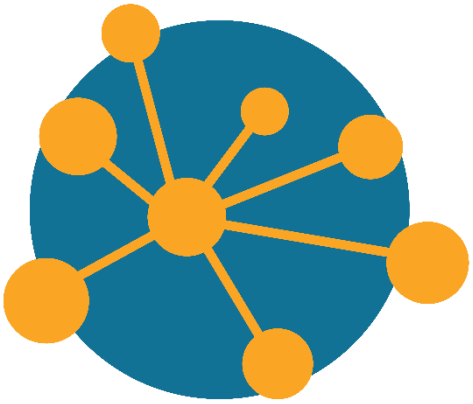
Digital Technologies in focus:

Supporting implementation of Digital Technologies



Scratch tutorial

acara AUSTRALIAN CURRICULUM, ASSESSMENT AND REPORTING AUTHORITY

Initiative of and funded by the Australian Government Department of Education and Training

Acknowledgements

The Curriculum group of ACARA has developed this professional learning program to support teachers in the 3-year National Innovation and Science Agenda (NISA) project known as Digital Technologies in focus: Supporting implementation of Digital Technologies.

For more information contact:

Julie King

Project Lead, Digital Technologies in focus

Curriculum Specialist, Technologies

julie.king@acara.edu.au

CONTENTS **Page**

ABOUT THIS TUTORIAL

There are three separate sections in this tutorial on Scratch:

- Section 1: Introductory tutorial
- Section 2: Creating a game
- Section 3: Extension exercises, resources and assessment.

Each section provides step-by-step instructions and images to support the learning of the visual programming language, Scratch. This language type is suitable for students demonstrating the Achievement Standards for Years 3 and 4 and Years 5 and 6 in the Australian Curriculum. If another curriculum is being used, please refer to the specific requirements regarding the use of a visual programming language.

Thought provoker

'Design and programming are human activities; forget that and all is lost.'

Source: Bjarne Stroustrup, Danish computer scientist and developer of the C++ programming language

Your challenge

Understand the basics of simple code – this works equally well for visual programming (block) and general-purpose coding – and how this applies to the Australian Curriculum: Digital Technologies.

Your mission

- understand coding practices and use coding tools effectively
- analyse the complexities and logic of code using block (or visual) code
- use the visual programming language, Scratch, in either its online or offline versions, to gain an understanding of input, output, sequences, looping, conditional branching and variables
- explore teaching and learning opportunities in other areas of learning
- create valid assessment opportunities for students to demonstrate the required standards and progress in their learning.

Setting the scene

Coding might look hard but it is only putting your thoughts into a form that a computer can follow in an efficient manner. Coding is the process of translating logic into a language that can be understood by a digital system. It is also important that the code can be understood by other coders, who might need to make modifications to it at later stages.

In this tutorial you will be learning to use the visual programming language, Scratch, which is available from the Massachusetts Institute of Technology (MIT) as a webapp or as an offline editor (free download for MacOS and Windows). The offline editor is really useful when you have a slow or intermittent internet connection. The theory is that all primary students coming into Year 7 will have experienced using a visual programming language such as Scratch; however, this might take some time to become a reality.

Alternatively, you can use Snap! (**http://snap.berkeley.edu**) from Berkeley University, especially if you are an iPad user. Scratch won't work on an iPad but ScratchJr will – **www.scratchjr.org/** MIT has released the beta version of Scratch 3.0 and the official version is available from 2 January 2019. This version will work on iPads and the offline editor.

Snap! is affectionately known as Scratch on steroids – in some ways it offers more coding opportunities but this is beyond the scope of this tutorial.

Before you start with this introductory tutorial you might like to initially watch the video *ScratchVariables* (**https://www.youtube.com/watch?v=1qAiiMJIc9E**) to see examples of perseverance, serendipity and collaborative learning.

SECTION 1: INTRODUCTORY TUTORIAL

Scratch is a visual programming language that uses graphic elements (or blocks) rather than just text to translate logic. Figure 1 is an annotated coding environment for Scratch that shows how blocks are used to make statements and control structures.

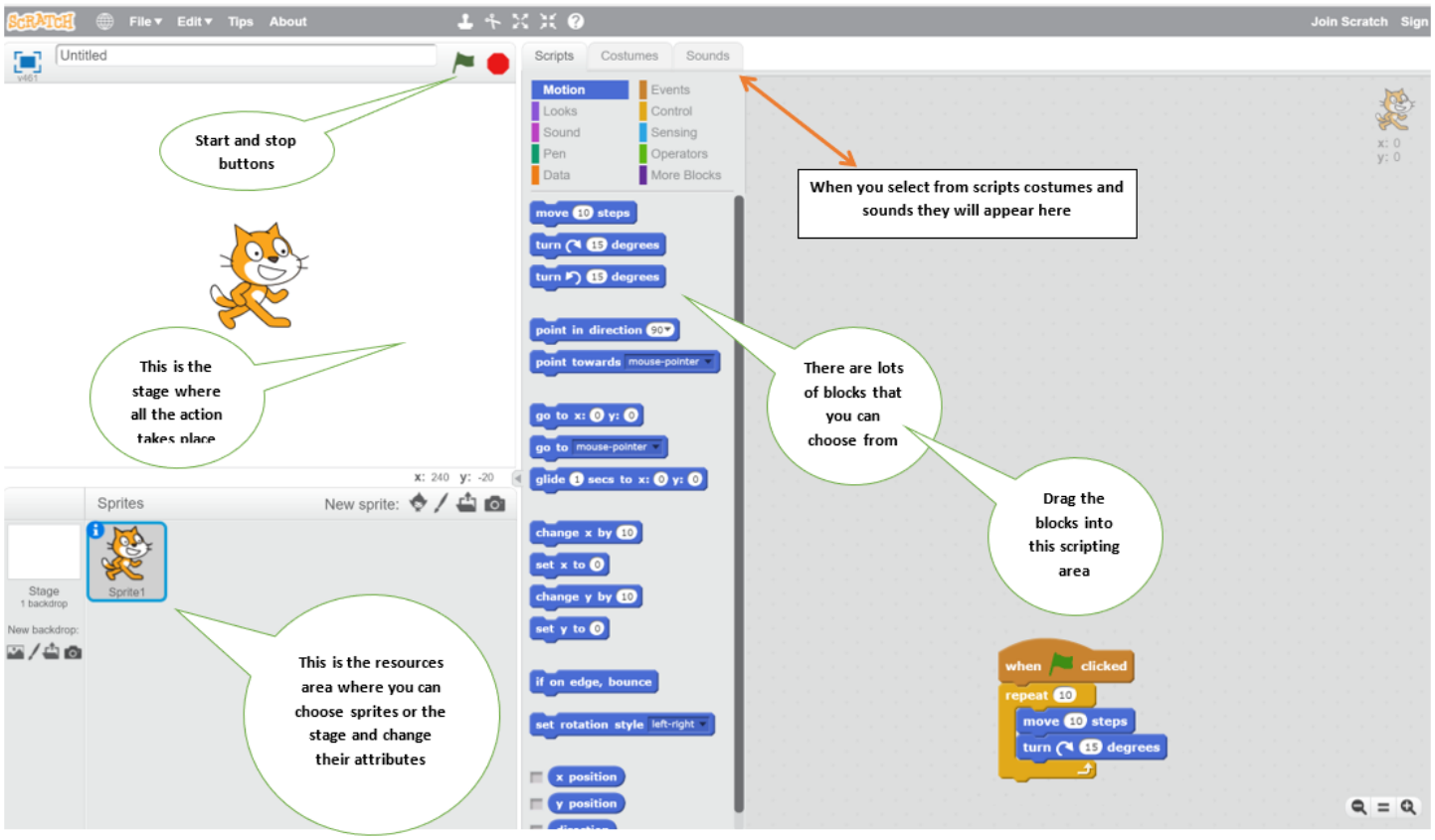


Figure 1: Scratch layout (there are slight variances, depending on your version)

To control your actor or sprite, click on it so Scratch knows you are writing scripts for it and then drag the script elements shown in Figure 2 (page 6), across to the scripting area on the right. Each script element has a characteristic shape, and not all shapes will fit into all slots. This is intentional (it's based on the behaviour of Lego™) as it disguises the syntax issues normally associated with general-purpose coding. Scratch is able to accept a variety of ways of inputting instructions, such as clicking and pressing the space bar, as shown in Figure 2.

Figure 2: Scratch will accept a variety of inputs

In each of these input cases, when the brown event is triggered, do what it says under it and observe the result. You can have more than one script or code fragment available, each responding to a different input event.

There are a variety of ways that you can start a program segment.

You can do the same thing to interact further with the user. Figure 3 (page 7) shows four different bits of code that become increasingly useful. Try them out.

<table>
<tr>
<td> 
</td>
<td>Before you try each of these, press the stop button so you are sure that any effect has been caused by your chosen bit of code and not remnant behaviour from a previous script.

Did these scripts do what you expected?

Did you notice that ‘answer’ has a variable value depending on what you enter in response to the question?

Computer language makes great use of this. You don’t need to know what the variable ‘answer’ (or output) contains – you only need to tell the computer what to do if it is a certain value.

The special nature of this last script is that it includes a conditional branch so that it does something different depending on a decision – in this case, whether your answer is equal to 40. These are often called IF THEN statements or more simply choice statements.</td>
</tr>
</table>

Figure 3: Scratch exercises for variables and output

The fourth script in Figure 3 is repeated in Figure 4 – it’s a bit wobbly, isn’t it? Pull the script apart and add a few new elements to give a better conditional branch.

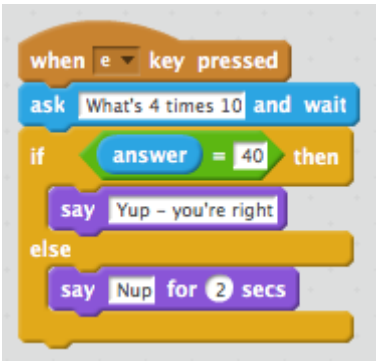


Figure 4: Scratch exercise using a conditional branch

This is better, but notice that a right answer plays up a bit – it just goes straight to the finish. You miss out on the recognition you deserve. Why is that? Because computers are so fast, you need to add a delay so you can see the result. This is done by putting a **loop** around the question so you get more than one opportunity to answer a little later on.

Let’s demonstrate the speed of the computer with some movement.

You are going to code a new project to get the sprite to move in a square.

But first your previous coding will be forgotten unless you save it somewhere – you can create a folder called Scratch on your desktop (right-click in any blank space on the desktop and choose new folder) and save it there. You then have two choices:

- If in Scratch Offline editor, select **File/Save as** and choose the desktop folder you created.
- If in Scratch Online, select **File/Download** to your computer and drill to the desktop folder you created.

You will be using this old code later on, so best to save it.

Starting a new project

To start a new project select: **File/New**. Let's start simply and then tweak our program to work smarter.

Think about how you would walk around the perimeter of a square. What are the movements that you need to make?

Hopefully you said something like:

Move forward 5 steps
Turn right 90 degrees
Move forward 5 steps
Turn right 90 degrees
Move forward 5 steps
Turn right 90 degrees
Move forward 5 steps
Turn right 90 degrees

This is an algorithm

You can code this in Scratch using some of the *Motion* (blue) blocks. Fifty steps are used to make the movement (output) more observable.

Click inside the white circle and type to change the current values.

You can right-click and choose **duplicate** to make the coding job easier, as shown in Figure 5.

Figure 5: The duplicate command makes coding faster and easier

What happens when you run the code? Nothing, right?

Well, no, that's not right. It's just that the computer is so fast that it does it all quicker than you can see it.

You can prove that easily by changing the last 'turn 90 degrees' to something like 'turn 30 degrees' and click the green flag again. Notice that something is happening but it is too fast for us to see it all.

So let's add in a wait time to see what is going on. **Wait** is under the orange control blocks. Use 'duplicate' again to increase the efficiency of coding.

Figures 6, 7 and 8 show the results of right-clicking on the top blue block of code and selecting 'duplicate' this line, and everything under it will be duplicated. Soon the code will be ready to run.

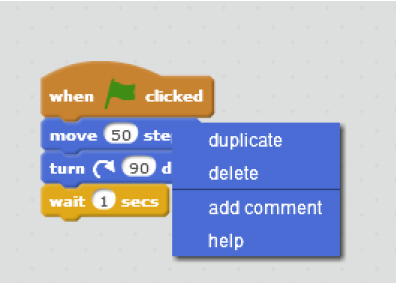


Figure 6: Scratch exercise using the duplicate command

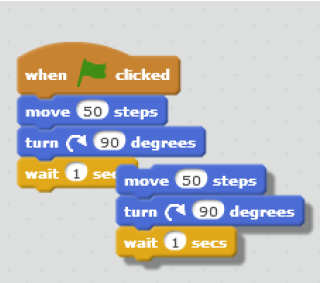


Figure 7: Scratch exercise showing duplication results

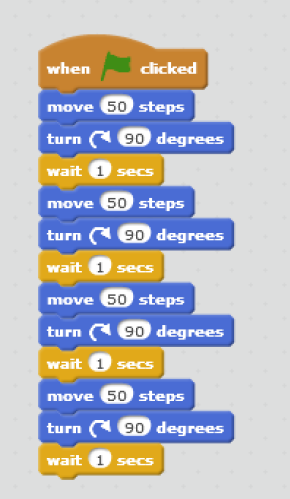


Figure 8: Scratch exercise showing code

Now when you run your code, you can see what is happening because of the 'wait' commands. You can also use the sprite to draw the results of your code.

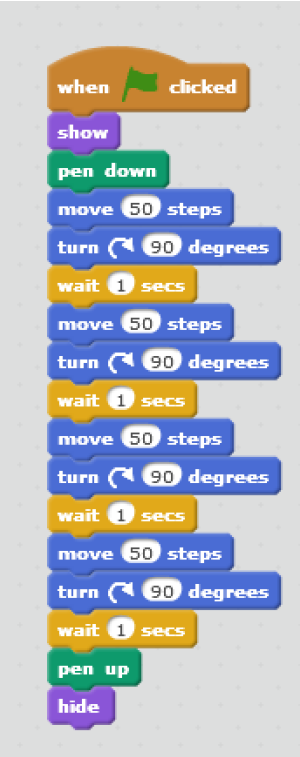


Figure 9: Scratch exercise using pen commands

You need to include two pen commands from the dark green blocks as shown in Figure 9. The *pen up* command at the end is not really needed, but it is good practice for the students.

The code in Figure 9 also includes a command to *hide* the sprite from the drawn image at the end. That also means you need a command to show the sprite at the start of the code, in case you run it again.

It's your turn. Try and get the square to clear each time the code is run so that the square is seen to be drawn each time.

Hint: Look in the *Pen* block for a command that might help you.

You can make the code work smarter by using a command that allows you to **not** type repetitive instructions.

In the orange control block there is a *repeat* command, as shown in Figure 10.

Figure 10: Scratch exercise using a repeat command

Note a couple of things about the *repeat* command:

- It has a slot where you can put other code inside it.
- It can be changed to a different number of repetitions.

Think about how you could use the *repeat loop* with the code you have just created. Think about what could sit inside the *repeat* command. Maybe you came up with something like the code shown in Figure 11 (page 11).

If you did, then great. If not, then you might have found another way that works – that's great too!

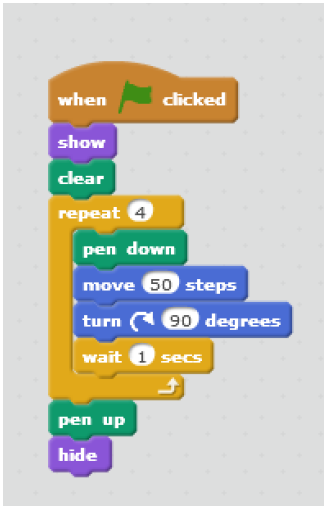


Figure 11: Scratch code using a repeat loop

Let's revise

You've learnt that there are different ways to start the code working, such as green flag and clicking on the sprite, as shown in Figure 12.

Figure 12: Scratch option to start code working

You've also learnt about a series of steps such as:

do this

then this

then this.

This is called a *sequence* and it is one of the big three things that algorithms can represent and code needs to do.

You've learnt how to repeat a sequence of steps a certain number of times, as shown in Figure 13 (page 12). This is called *repetition, looping or iteration* and it is another of the big three things that algorithms can represent and code needs to do.

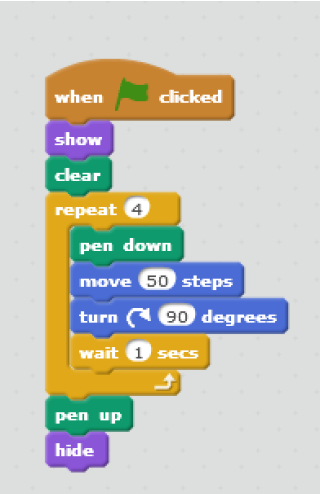


Figure 13: Scratch example of repeating a sequence of steps

You've also learnt about a *conditional* situation – if something is true, do this or else do something else, as shown in Figure 14. This is the third of the big three things that algorithms can represent and code needs to do.

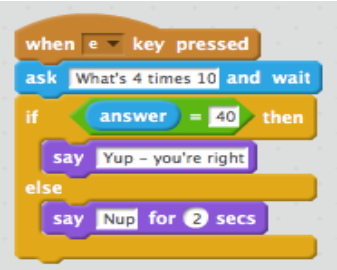


Figure 14: Scratch example of using conditions

There is just one more thing that you need to go through – getting feedback from the user in a useful way.

Go back to the original code that you saved in a folder on your desktop.

Opening a saved file (multiplier program)

To open a saved file, select **File/Open** and Scratch will ask if you want to save your current project. Once you do that, Scratch will let you drill to the project you want to load and it will load it onto the programming area for you to continue experimenting with it. It should look like the code in Figure 14.

The next step is to put a loop around the code so that it will continue to ask you for an answer more than one time. The *forever loop* achieves this, as shown in Figure 15. This loop will, as its name suggests, go on forever, even if you get the correct answer. There are lots of ways to fix this but an easy way is to press the red stop sign next to the green flag at the top of the stage window.

Figure 15: Scratch exercise of using the forever loop

Another problem here is that this code only works for one problem – '4 times 10' – this is referred to as being *hard coded*. This is great if you want to concentrate on just that problem but if you want to change the numbers you have to rewrite the program code every time. This is not an efficient way of coding.

So let's fix it so the code picks a random number for each multiplicand.

Variables

You'll need to create three *variables* (reddish data blocks) – one each to hold the value of the first and second numbers and one for the value of the result of the multiplication (product).

You'll have to create and name each variable needed first.

To do that, click on **Make a variable** and give it a suitable name. 'X' would be okay but 'first number' is much more descriptive and easier to remember if the code gets bigger and it is easier for another coder to interpret. You will use it for this sprite only in this example, as shown in Figure 16. That makes it a local variable – something that will come up as you and your students get more into coding. (In this example, it makes no difference.)

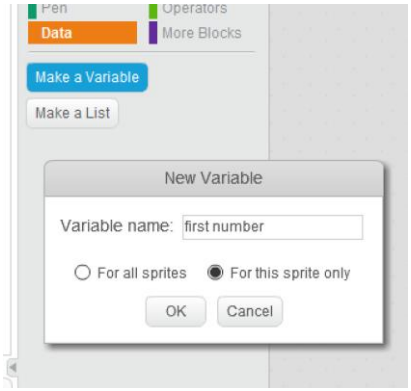


Figure 16: Scratch exercise of using the variable function

Do the same for ‘second number’ and ‘product’.

You should end up with a *Data* block of commands as shown in Figure 17:

Figure 17: Scratch exercise of a Data block of commands

Now you want to use mathematics to choose the random numbers.

The green *Operators* blocks have a large number of comparison operators, logical operators and mathematical operators available.

You will just get the code to choose the random numbers and give it some limits.

Choose the **pick random 1 to 10** block and leave the numbers 1 and 10, as shown in Figure 18.

Figure 18: Scratch allows random numbers to be used

If you want to test the code, you can put it in the coding area and click on it. A random number will appear each time you click it. Give it a go.

The next step is to set your variables ‘first number’ and ‘second number’ to random numbers between 1 and 10.

In the *Data* block of commands you will see a block ‘set product to 0’. Note that ‘product’ has a down arrow next to it.

If you click the down arrow next to product you get the choice of first number, second number and product, as shown in Figure 19. So, the block is happy to work with any of the variables that you created.

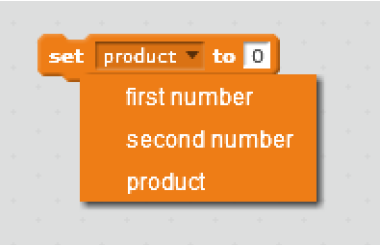


Figure 19: Scratch function for selecting variables for a product

Select **first number** and then add **pick random 1 to 10** to the little square after 'to' where the 0 is, as shown in Figure 20. If you have trouble placing things in the white circles or squares, try dragging with the mouse arrow at the left end of the block you are dragging.

Figure 20: Scratch function for setting first number to random within a range

Do the same thing for the variable **second number**.

Now think about what you want to set the product variable to. You want it to be the product of the two random numbers: In words that would look like this: Set 'product' to 'first number' times 'second number'.

You can code it with the four blocks, shown in Figure 21.

Figure 21: Scratch blocks needed to code multiplication

Have a go and see what you come up with.

If you cannot remember where to find a block, look at its colour first. Your result should look like Figure 22. Remember, if you have trouble placing things in the white circles or squares, try dragging with the arrow at the left end of the block you are dragging.

Figure 22: Scratch exercise of selecting blocks needed for multiplication

And the three commands you have created should look like Figure 23.

Figure 23: Scratch commands for multiplication with random selections for variables

Now you need to work out where to put them in the code. Good coding practice states that you should initialise variables first. Put the code straight after **when f key is pressed**.

Your code should now look like Figure 24.

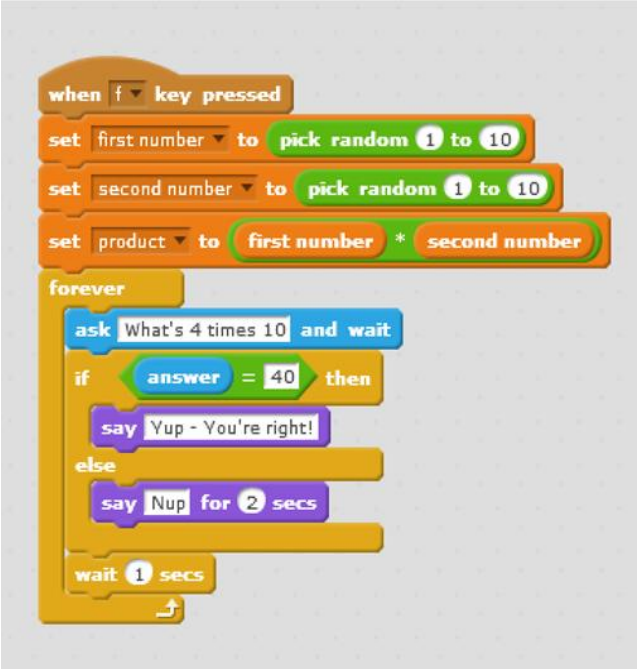


Figure 24: Scratch example of sequencing code

Let's recap

You have decided to make the code choose random numbers to multiply each time you press the 'f' key.

You have set up variables for the two random numbers 'first number' and 'second number'.

You have set up a variable to store the result of the computation: 'product = first number x second number'.

You have placed all those variables at the start so later code can use them and know what they are.

Next step: Joining the blocks

Now all you really need to do is change the *ask question* to reflect the new variables and wait. You will also need to change your loop and add a second. Before you do the last bit, think about what would happen if you put the set statements inside the *forever loop*.

Next you need to join together the ‘first number’, the word ‘times’ and the ‘second number’ in one line.

Let’s assume the first number is 5 and the second number is 7 (randomly chosen, remember).

You want the following to appear as the question: ‘What is 5 times 7?’ That should be pretty easy, but in Scratch and many other languages you need to **join** different types of data in special ways. This is called concatenation.

The different types of data are:

- text or string – ‘what is’, ‘times’
- integers – 5, 7.

So you need several **‘**joins’. You will still use the same *ask and wait* block and add the ‘joins’ to it.

There are four bits of data – ‘what is’, ‘5’, ‘times’, ‘7’ – so you need three joins. You will build the statement backwards to help work it out.

You will use the three joins shown in Figure 25. Note that the blank squares come first in blocks 2 and 3 – you will see why in a minute.

Working backwards, the statement would read: ‘Second number’ will be joined by ‘times’, which will be joined by ‘first number’ and then joined to ‘what is’. In your 5 and 7 example, the statement would be – 7 needs to be joined to ‘times’, which needs to be joined to 5, which needs to be joined to ‘what is’.

Figure 25: Scratch example of join blocks

Joining it all together

So the first part of joining this all together is as follows: 'Second number needs to be joined by times', as shown in Figure 26.

Figure 26: Scratch example of joining two blocks

Next add the other **join** in the space that is left, as shown in Figure 27.

Figure 27: Scratch example of joining three blocks

Now, if you read the command (without the join words and left to right) it reads how we want it: 'What is first number times second number'. This is something you will need to teach to students as a great way to think about joins.

Next you need to put the complete join statement into the *Ask* block, as shown in Figure 28.

Figure 28: Scratch example of joining statements into the Ask block

If you haven't moved your command blocks around too much the whole code will look like Figure 29.

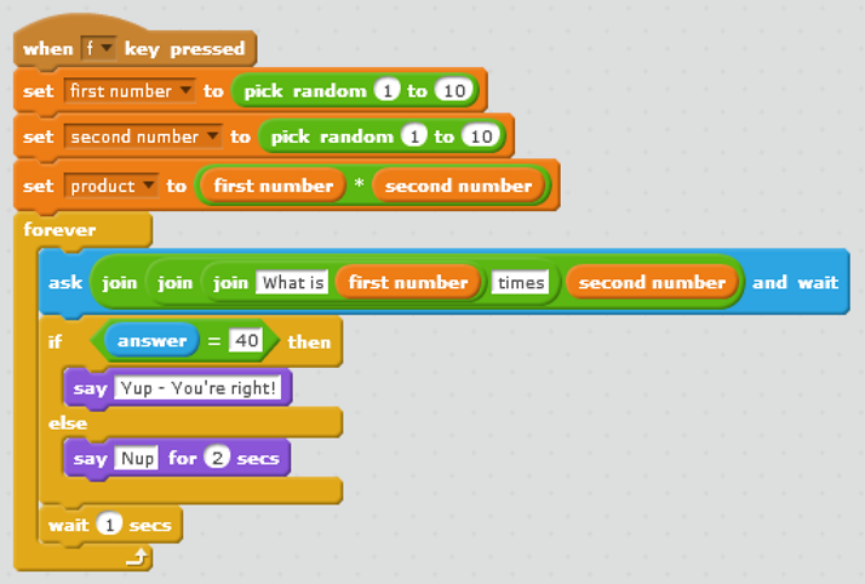


Figure 29: Scratch example of using variables, joins, ask and wait, loop commands

Can you see the last bit of code that needs to change? There is still something to do. Run your code a couple of times and see what it does.

What needs to change is: **if answer = 40 then**.

You want the answer to equal 'first number times second number' and that will rarely be 40. As luck would have it, you have created a variable already to hold the result of 'first number times second number'. It is called *product*. So, no matter whatever is the 'first number' and whatever is the 'second number', *product* will always be the result of multiplying those two numbers. Why? Because that is what we told the computer to do with this line, as shown in Figure 30.

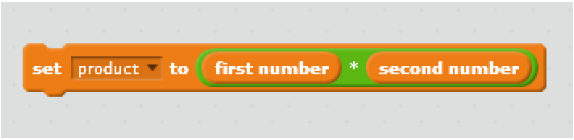


Figure 30: Scratch example of selecting blocks needed for multiplication

The next step is to go into the reddish data blocks and drop the **product** variable into the box that currently says 40, as shown in Figure 31.

Figure 31: Scratch example of selecting the product variable

Testing the code

Once you think you are finished, it is always important to test the code to see if it works. Test it a few times, trying with wrong answers as well.

Did the program work as expected? If you answered 'no' to these questions, then you know there is still a little more we have to do.

The main problem is the *forever loop* because it will keep asking the same question each time even if you get the right answer. Think about what you actually want.

If you said you want a program that continually comes up with new problems to solve, but only goes on to a new problem when the current one is solved correctly, then you are on the right track.

So let's think about that.

What the code is doing right now is basically the same as the initial problem you are trying to solve. It will ask a question and wait until it gets the correct answer, but then ask the same question over and over again.

To address this problem the first thing you need to do is change the *forever loop* to one that will repeat until you get the right answer.

There is such a loop in Scratch and it is shown in Figure 32.

Figure 32: Scratch example of a loop that repeats until the right answer is input

Note that it has a hexagonal space after 'until', where something can be placed.

Question: How can the program know when the correct answer has been supplied?

Answer: The same way that the *if statement* knows.

You can get the code to repeat until **answer = product** (see how it is in a hexagonal type box as well).

So our *repeat statement* looks like that shown in Figure 33.

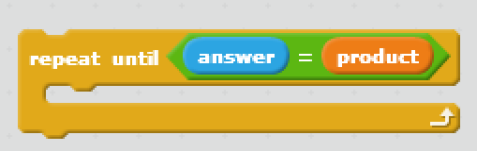


Figure 33: Scratch example of using a repeat statement

The whole loop is shown in Figure 34.

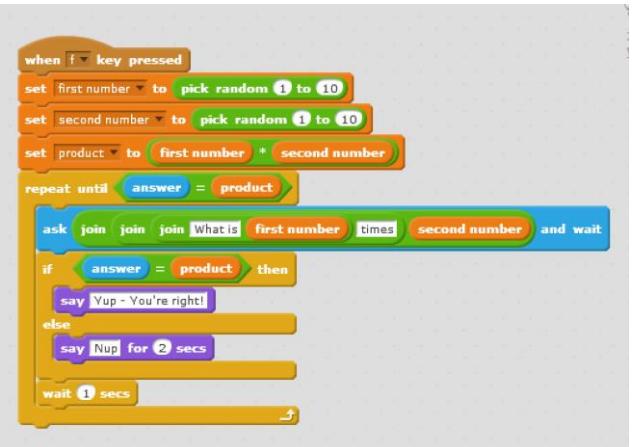


Figure 34: Scratch example of using a whole loop

If you run the code now it will work brilliantly, but only once, and then you have to press **f** again each time you want to run it.

That is okay, and you might want to stop there, but if you want the code to continually come up with new questions, then you need to add another loop.

This time it can be a *forever loop*. If you put all the code in a *forever loop* then after the correct answer is given, new random numbers will be chosen.

The code is shown in Figure 35. Try running it now.

Does it do what you wanted it to do? Hopefully you can answer ‘yes’.

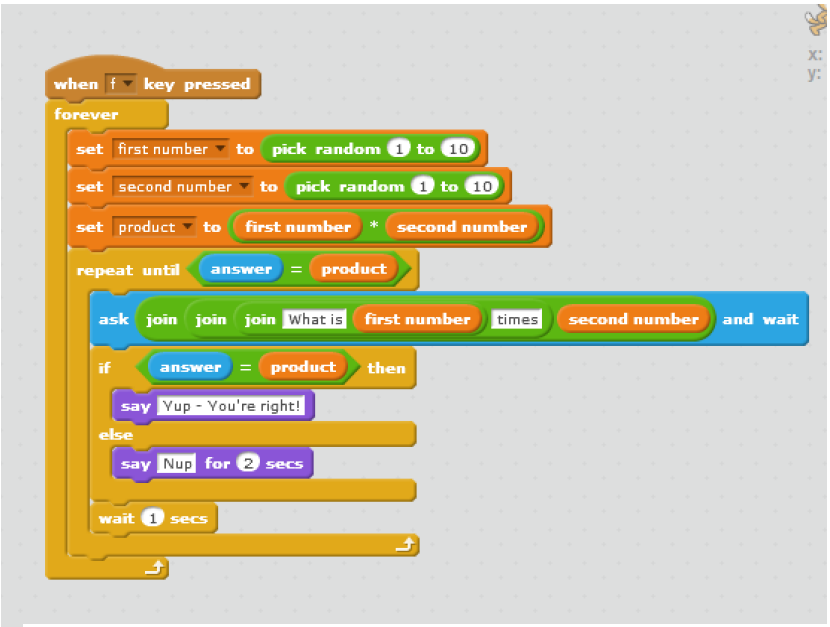


Figure 35: Scratch code for generating a new question once a correct answer is given

Are you happy with the result? What happens with this code is when the question is asked, it looks like what is shown in Figure 36. The ‘6times6’ format is neither conventional nor attractive. It can be easily fixed by adding some white spaces (a space bar) to either side of ‘what is’ and ‘times’, as shown in Figure 37.

Figure 36: Scratch example showing if spaces are not added to separate instructions

Figure 37: Scratch example of adding spaces to separate commands

The result of adding the spaces is shown in Figure 38 – more attractive and conventional.

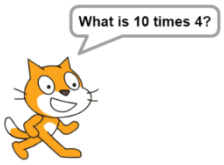


Figure 38: Scratch example of output
when spaces are added

There is another part of this solution that needs fixing. While you see nicely spaced writing on the stage, you also see the variables being displayed and one of them is the answer, as shown in Figure 39.

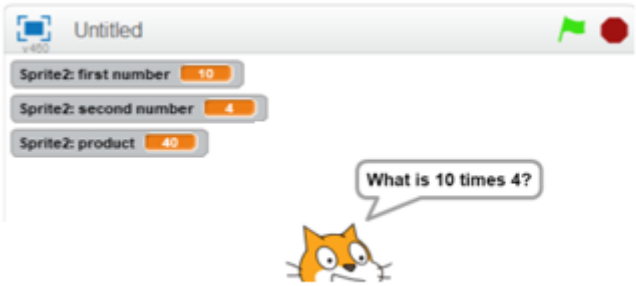


Figure 39: Scratch example of code displaying variables
on the stage

There are times when variables on the stage are great, for example, for timers and scores, but not for this project. You can hide the variables by going into the reddish data blocks and **unticking** the little boxes, as shown in Figure 40. They will disappear from the stage as you do this.

Figure 40: Scratch example of hiding
variables from the stage by unticking

You can also hide the variables by right-clicking on the variable on the actual stage and choosing **hide**, as shown in Figure 41.

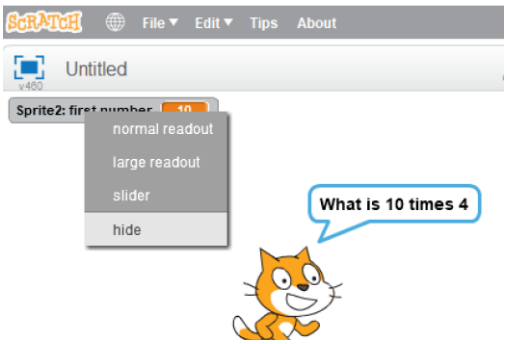


Figure 41: Scratch example of hiding variables from the stage by right-clicking

The solution is complete!

Defining key terms

By now you'll have learnt a lot, particularly if you are new to coding. Reflect on what you have learnt by writing your own definitions in Table 1. You might like to add additional terms.

TABLE 1: KEY TERMS

Term	Meaning
Input	
Output	
Looping	
Variables	
Conditional statements	

How can this look in your classroom?

From the Scratch work you have done, you are now able to do all sorts of things.

In your classroom, the **movement** exercise could result in:

- students moving the sprite through a maze
- using the operators to automatically draw shapes of various angles just by getting user input and then dividing by 360 to get the angles that need to be turned
- adding the 'xy' background, as shown in Section 3 and working with coordinates
- animating a story like 'We're going on a bear hunt'.

In your classroom, the **multiplier** program could result in:

- times tables drill games that the students create – either specific, by reducing the random number or general, by including random numbers up to 12
- games for younger students that require some simple mathematics – it doesn't have to be just multiplication
- a game being created where the cat (sprite) could move forward every time a correct answer is given and you win when you get to the other end of the screen. You could add a timer to see the fastest in the class.

SECTION 2: CREATING A GAME

Scratch can be used to create games, and it's great for games that represent a concept, event or idea from another learning area. One such example is the wire loop circuit game, where the objective is to move an object along a curved length of 'wire' without touching it. It draws on Years 5 and 6 students' knowledge of how '…electrical energy can control movement, sound or light in a designed product or system …' (*Source: Australian Curriculum F–10, Design and Technologies, Content description ACTDEK020, ACARA, 2015*)

From a Digital Technologies perspective, creating a game with a visual programming language, such as Scratch, is an effective way for Years 5 and 6 students to develop and apply their knowledge and skills associated with the Content Description 'implement digital solutions as simple visual programs involving branching, iteration and user input'. (*Source: Australian Curriculum F–10, Digital Technologies Content Description ACTDIP020, ACARA, 2015)*

Figure 42 shows an image of a sample wire loop circuit and its underlying principles, which form the basis of the game. School programs that take an integrated approach to this project would need to initially focus on content related to electrical energy circuits before starting to create this game. This resource focuses only on the programming aspect of the game.

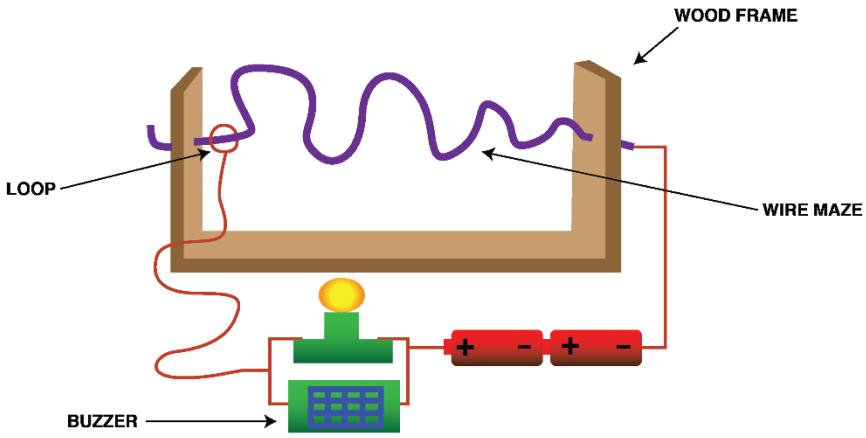


Figure 42: Model of a circuit that shows how electrical energy (generated by batteries) can control movement

Students make a game that focuses on the curved loop part of a circuit, with the user having to negotiate the loop or 'track' to get from left to right. Each time a user touches the wall of the track he or she will get a 'zap' (indicated by a buzzing noise) and lose a life.

There are many permutations on determining a winner of this game, such as the fastest time to move from left to right, the highest score based on number of hits divided by fastest time the least lives lost.

Setting up the game

The default size of the sprite, (the cat, referred to Cat1), is too big for your purposes (getting through a track) so the first thing to do is shrink Cat1. You can choose another sprite if you want, but this resource uses the default cat. Another cat (downloaded from the internet) will be added later for a special effect.

The format of this tutorial is that after an explanation of different commands and codes needed to create this game, it will be followed by a '*To do'* segment – this is where you carry out the instruction.

Resizing a sprite

To **shrink** Cat1, choose the icon that is circled in red, as shown in Figure 43, and then move to Cat1 and click. Note that the icon to the left of the shrinker will enlarge the cat.

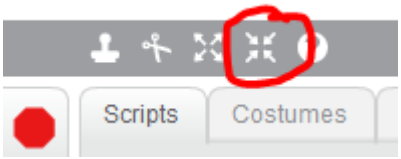


Figure 43: Scratch icon for shrinking a sprite

To do: Resize your own sprite.

Editing the backdrop

The next task is to draw the track. Single click on the **Stage 1 Backdrop**, (marked in red) as shown in Figure 44.

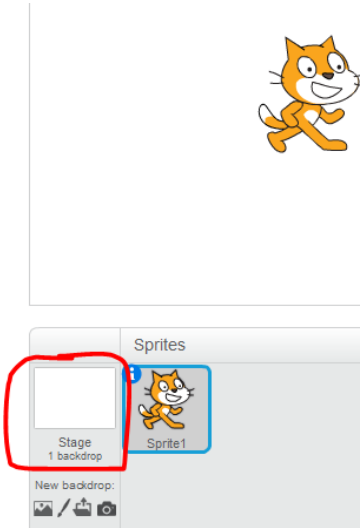


Figure 44: Scratch example of creating a backdrop

Then click the **Backdrops** tab, which suddenly appears up the top of the blocks area between ‘Scripts’ and ‘Sounds’, as shown in Figure 45. Choose the **brush** tool so you can draw freehand. It is the one highlighted in blue.

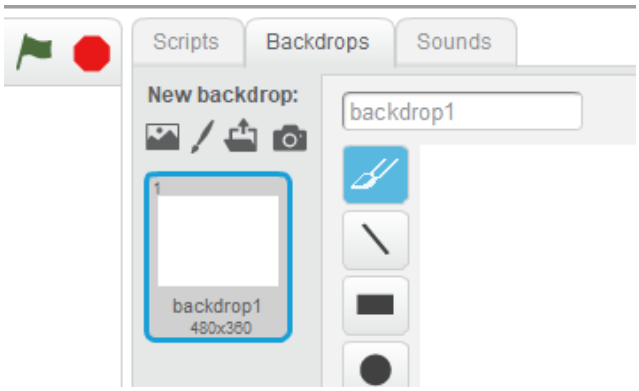


Figure 45: Scratch example of selecting Backdrops tab and brush tool

The next step is to **draw** a track that Cat1 follows. The track must be wide enough to let Cat1 move through it without hitting the sides. It will have a left and right side, like a road. A sample track is shown in Figure 46.

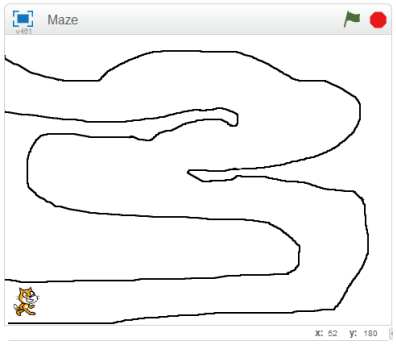


Figure 46: A track drawn in backdrops, using the brush tool

Note the size of the track compared to Cat1 – it is small enough to fit all the way around the track and this is important. Of course, you can reduce or enlarge the size of Cat1 to suit your track, as shown in Figure 43 (page 26).

If you make a mistake there is an eraser tool within the Backdrops edit screen as well as a text tool, which will be discussed later.

To do: Draw your own track on the backdrop. Make sure it is just one colour!

Adding another sprite

In this game it's useful to use a second sprite, as shown in Figure 47.

Figure 47: Scratch allows more than one sprite in a solution

Cat1 will be used until it hits the side of the track. When this happens, the second cat (referred to as Shocked cat) will be used because it has a shocked expression, in keeping with the wire loop game theme. Cat1 will then reappear. It should be effective, especially if you add some other special effects. See 'Adding a sound effect' (page 37).

You can find a range of sprites on the Scratch Wikia website at: **http://scratch.wikia.com/wiki/File:Cat1_sprites_(game).png** It is suggested that you save a selected sprite in the file where you keep all your Scratch programs. In this instance the Shocked cat was saved as **shocked cat.png** and then this allows you to retrieve it from the 'Upload sprite from file' function, as shown in Figure 48. This Shocked cat will also appear in your sprites area.

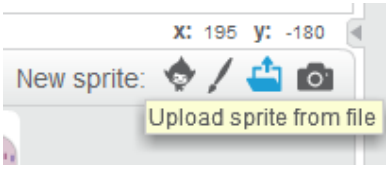


Figure 48: Scratch function to upload sprite from file

In Figure 49 you can also see a donut sprite that was chosen from the *sprite* library. You do this by clicking on the **sprite icon** (little figure with big hair) to the left of the brush. The donut sprite will be added to the game later.

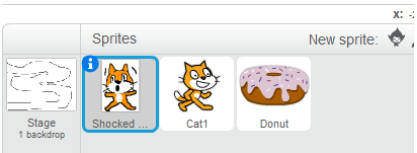


Figure 49: Additional sprite (donut) chosen from sprite library

To do: Get another sprite to use for when Cat1 touches the edge. There is a lightning bolt within the built-in library if you don't want to use a 'shocked' cat.

Now you are almost ready to start coding. But first you need to think about, and document, what you want Cat1 and the game to do, the decisions that need to be made and the order of events. This is where you write an *algorithm* to help capture your logic and drive your coding. **Note:** At Years 5 and 6 the Achievement Standards state that students should be able to write algorithms.

The logic (an algorithm)

1. I want Cat1 to start at the bottom left and end at the top left.
2. I want Cat1 to move smoothly when I press the arrow keys. When the keys are not pressed I want Cat1 to stay still.
3. If Cat1 hits any track edges (in black) I want it to do three things:
    a. Swap to the Shocked cat and play a sound. The Shocked cat needs to be in exactly the same spot as Cat1. When it is being shocked, the cat should hide.
    b. After being shocked, Cat1 should return to its normal self and the Shocked cat should hide.
    c. After being shocked, Cat1 should return a couple of steps back from where it hit the track edge.
4. It is important to have Cat1 return a couple of steps back so that it doesn't just continue getting shocked. It serves another function as well – it means that Cat1 cannot just cross over the track edge; rather Cat1 is forced to follow it. The track will restrict Cat1 within its edges.
5. I want a variable that will count the number of times the track edge is touched.
6. I want the game to end when the donut (finishing incentive) is touched.

Making the cat move smoothly

Make sure before you start coding that Cat1 has been selected. It will have a blue box around it if it has, as shown in Figure 50. Nearly all the code in this program needs to happen when the cat object does something, so the code needs to be put in the cat sprite coding area.

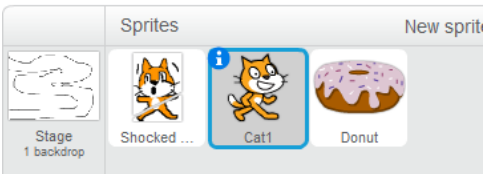


Figure 50: The selected sprite is indicated by a blue border

You make Cat1 move by including some *if statements* inside a loop that checks forever if the condition is true. Let's start with **when green flag clicked**.

To start, let's get Cat1 to move to the right when the green flag is clicked and the right arrow key is then pressed. The code to achieve this is shown in Figure 51. Create that code yourself and try it out.

Figure 51: Code to start Cat1 moving to the right of the track

Hopefully you have a smooth-moving cat.

To do: Try changing 'change x by 2' to some other number and see if there is still smooth movement.

You want a low number like '2' so that Cat1 can be controlled in small increments.

Now you need three other *if statements* to cover the other three arrow keys (left, up, down). See if you can code it yourself. The answer is shown in Figure 53 (over page) but you should try it to learn how to do it for yourself.

Testing the cat's movements

You should have a program that will move Cat1 around the stage wherever you want it to go. What is happening is the *forever loop* makes the program constantly check to see if one of the four arrow keys is being pressed. If so, it will move two steps in whichever direction is indicated by the arrow key.

You should be able to follow the track you made, but notice there is nothing stopping you from crossing the lines. As an incentive to Cat1, you can add the donut to the end of your track, as shown in Figure 52. At the moment; however, if you want to get to the donut you can go straight up. You want to force Cat1 to stay within the lines.

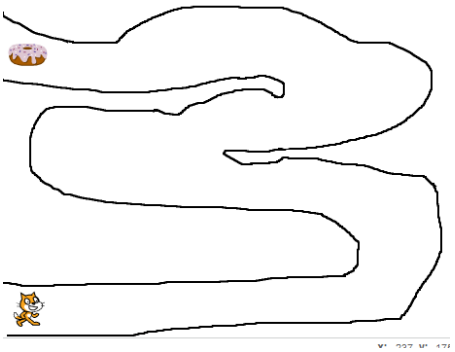


Figure 52: An additional sprite (a donut) is added at the end of the track

If you are successfully moving Cat1 around the stage, your code should look very similar to that shown in Figure 53.

Figure 53: Code showing if statements for moving Cat1 around the stage

You are going to finish with the code you have created (like that shown in Figure 53) and begin a new set of blocks, which will also start working when the green flag is clicked. You can put this new set of instructions right beside the one you have already created. At the start it will look like Figure 54.

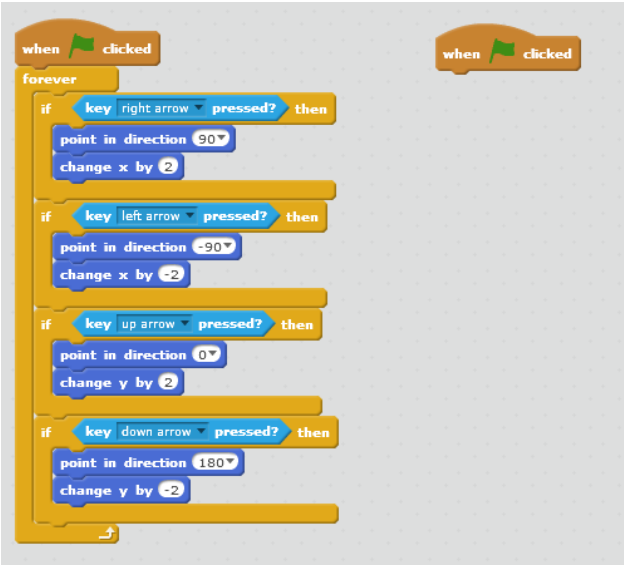


Figure 54: New instructions are placed next to existing code

You will build this new block by using another *forever loop* to make the program check constantly if the edge of the track has been touched, and some *if statements* to work out how to move back a couple of steps. After that you will make the Shocked cat appear.

Keeping the cat within the track

One way to tell if Cat1 is touching the edge of the track is to use the *Sensing* block. Notice that the colour is currently orange. Drag the **touching color** block**,** as shown in Figure 55, into the coding area and click in the box. The arrow should turn into a hand. Move the hand around the screen and you will see that the colour of the little box will change to suit what the hand is currently hovering over.

Figure 55: Touching colour block is used to match an existing colour, such as the colour of the track

To do: Try this!

Now that you can choose a colour, move the hand over the track and when the little colour square is the same colour as the track edge, click to lock in that colour. Hopefully you made your track only one colour, as suggested earlier.

You want a *forever loop* to constantly check and you want two levels of *if statements* (nested conditional statements). The first *if statement* will check if the edge is being touched – if it isn't being touched you don't want it to do anything. The second-level *if statement* will determine what key is being pressed, so you know which way to go back a couple of steps. The first *nested loop* looks like Figure 56.

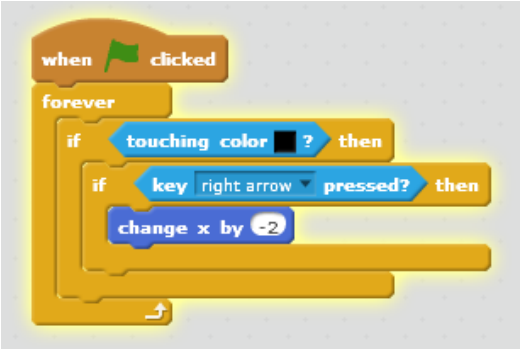


Figure 56: Scratch example of nested conditional statements

The *forever loop* checks forever if Cat1 is touching an edge (touching something black, the colour of the track shown in Figure 52). If this is the situation, then if the right arrow key is being pressed, you need to change the x axis by –2 to move Cat1 away from the edge.

To do (maybe): You could talk to your students about Cartesian maths and x and y coordinates at this point.

If the left arrow is being pressed, then instead of it returning –2 steps, you want it to return 2 steps, so the new code will look that shown in Figure 57.

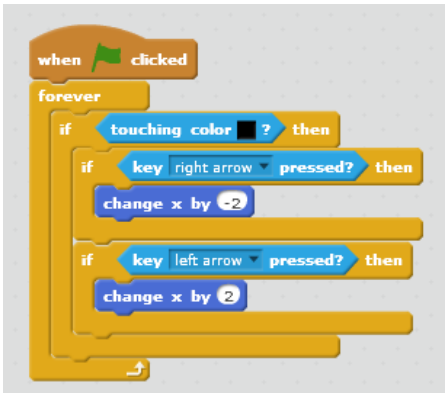


Figure 57: Code for moving away from an edge if right or left arrows were pressed

To do: Code this much and then try it out in your program.

See if you can complete the code to deal with the situations when Cat1 touches the track edge when moving up or down. The finished code is shown in Figure 58 (page 34), but again it would be good for you to try it first.

Check your progress

If you have been successful up to this point, you should have a cat that moves smoothly (logic point 1) and moves back a couple of steps when it hits an edge (logic point 3c), as stated in the algorithm. Cat1 should also be contained within the track, so it cannot escape and take a shortcut to the donut (logic point 4).

To do: Test your program to see that it does the above.

Now you will get the rest of the code working, starting with logic point 3**,** which has three parts to it:

If Cat1 hits any track edges (in black) I want it to do three things:

a. Swap to the Shocked cat and play a sound. The Shocked cat needs to be in exactly the same spot as Cat1. When it is being shocked the cat should hide.
b. After being shocked, Cat1 should return to its normal elf and the Shocked cat should hide.
c. After being shocked, Cat1 should return a couple of steps back from where it hit the track edge.

Before you start coding, just check that your code is like that shown in Figure 58 (page 34).

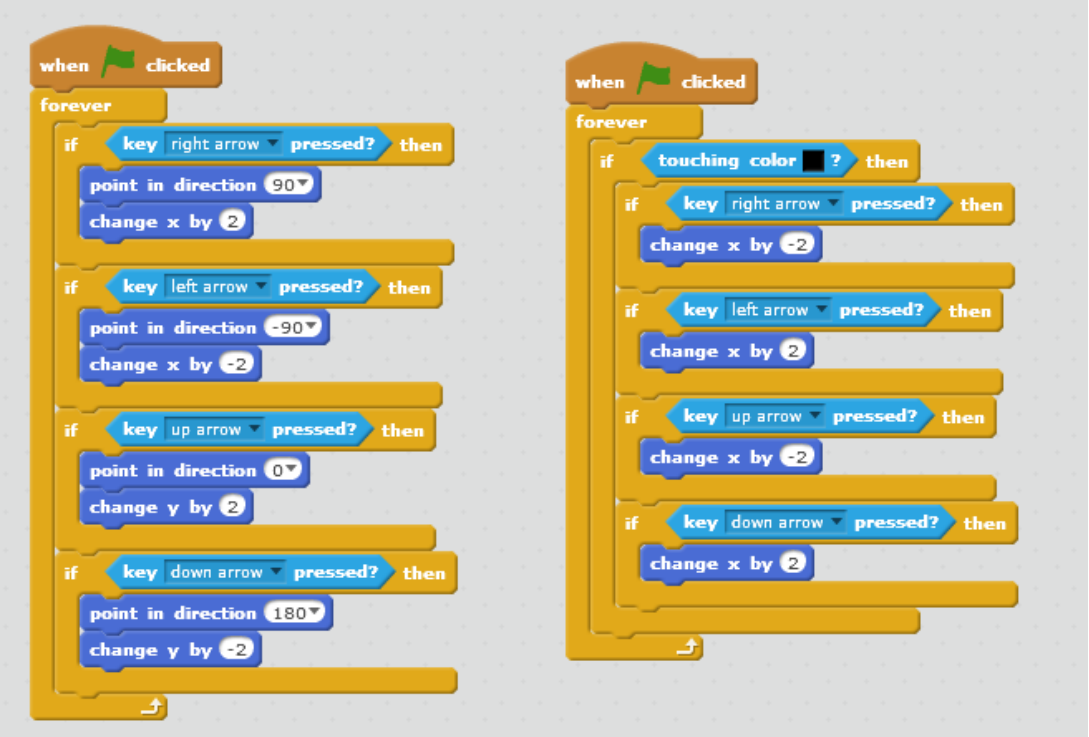


Figure 58: Two blocks of code with the right one showing if Cat1 hits the edge of the track

Broadcasting a message

Scratch has an interesting way of letting other parts of the program know what is happening. It is called *broadcasting* and it means letting other code know when something is happening that it is waiting for.

You are going to use *broadcasting* to make the Shocked cat appear, make Cat1 disappear, play a sound and then get Cat1 to reappear. It should look pretty good if you do it correctly.

Start by choosing the **Event block** that looks like Figure 59.

Figure 59: Event block used for broadcasting

Drag the block into the coding area for Cat1 (where all the code is at this stage). Click the down arrow on the block and choose **new message.** Call the new message ‘Zapped’. Drag another similar ‘Event block’ into the coding area and create another new message called ‘zapped_over’. You should have two blocks that look like Figure 60.

Figure 60: Two event blocks for different messages

The logic is that when ‘Zapped’ is broadcast, Cat1 should disappear. When ‘zapped_over’ is broadcast, Cat1 should reappear.

To do this, add **hide** and **show** to the respective *Event* blocks, as shown in Figure 61.

Figure 61: Functions to hide and show respective Event blocks

Next you need to work out when to broadcast the message. If you do it whenever Cat1 is touching the edge, (black colour) then this might cause a constant loop to occur. You want it to broadcast only after Cat1 has moved back away from the edge. Therefore you need four broadcast messages.

The broadcast message looks like Figure 62. You want to broadcast ‘Zapped’ regardless of which arrow key made Cat1 touch the edge.

Figure 62: Broadcast block for Zapped

To do: Make sure your code looks like Figure 63.

The logic is: Check to see if Cat1 is touching the edge. If it is, then work out which key is being pressed and move back two steps. Then **broadcast** ‘Zapped’. If ‘Zapped’ is received, then hide Cat1.

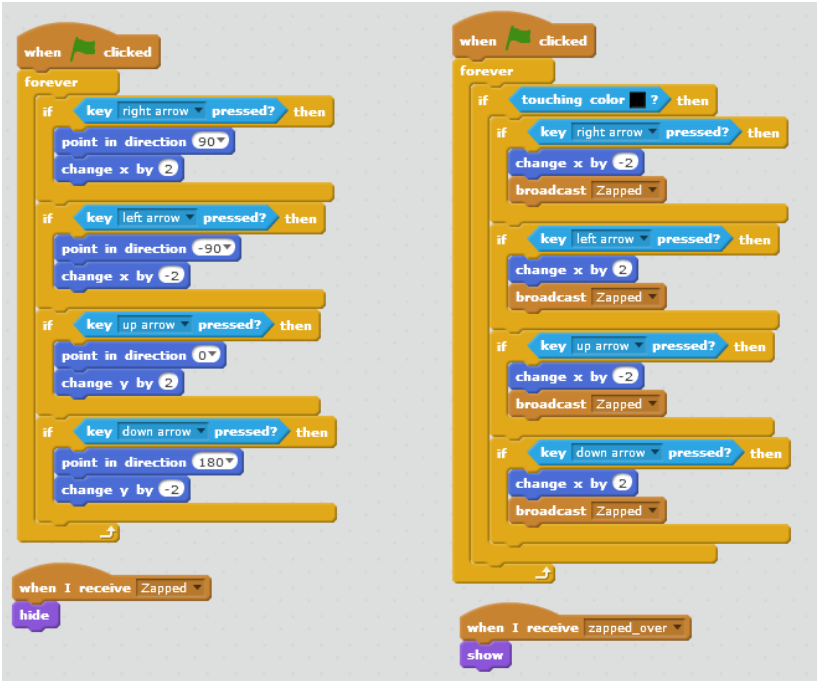


Figure 63: Code for Cat1, including broadcasting

Next you will make the Shocked cat do several things: appear when **'**Zapped' is broadcast, flash a couple of times, play a sound and then disappear. Also, you need to broadcast another 'zapped_over' message to let Cat1 know when to reappear.

You are finished with Cat1 for now but will return to do a couple of quick things at the end.

To do: Just to be safe, let's save your code now.

Remember: If using the offline editor, go to **File** on the top ribbon, choose **Save as** and pick a location you will remember.

If in Scratch Online, choose **File/Download** to your computer and drill to somewhere you will remember.

Coding a second sprite

To code the Shocked cat make sure it is selected – it will have a blue box around it (as shown in Figure 64) and all the code you have written will have disappeared from the coding area. Don't worry; it is still there. **Toggle** between Shocked cat and Cat1 and you will see your code still exists.

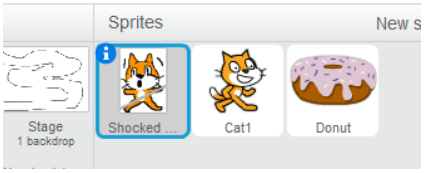


Figure 64: The selected sprite is indicated by a blue border

If the Shocked cat cannot be seen anywhere on the stage area (anywhere near your track or donut or Cat1), then drag it in from the sprites area.

In the Shocked cat coding area, choose another **when I receive Zapped** *Event* block (as shown in Figure 65). Also choose a **when green flag clicked** block. The first thing you want to do is when the green flag is clicked you need to make sure the Shocked cat doesn't appear. When the program starts, it needs to hide.

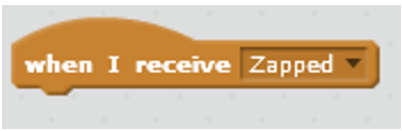


Figure 65: Event block for the Shocked cat

Your code for Shocked cat should look like Figure 66.

Figure 66: Code to date for the Shocked cat

You are going to add a *repeat loop***,** which will make the Shocked cat appear and disappear twice with a 0.3 second wait time in between. You will also broadcast 'zapped_over' after the loop is finished, so that Cat1 will know to reappear.

To do: Try doing this before looking at the code shown in Figure 67.

The logic: When the green flag is clicked, Shocked cat will hide and Cat1 will be able to move around and crash into the edge of the track. If it does, a message called 'Zapped' will be broadcast. Shocked cat is waiting for this message. When it comes, it will flash twice and then hide. It will also broadcast 'zapped_over'. Cat1 is waiting for this message. It will alert Cat1 to reappear (show).

The code for this is shown in Figure 67.

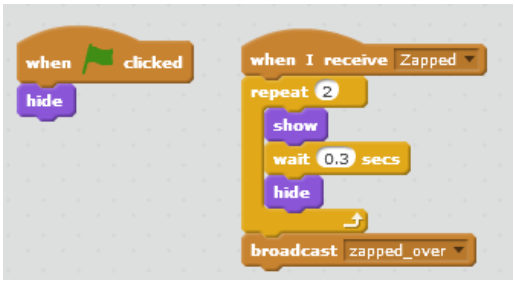


Figure 67: Repeat loop used to make Shocked cat appear and disappear twice

Remember, the code is in the Shocked cat coding area. You've probably tried it out and have noticed a big problem. Can you identify the problem?

Before you tackle the problem, you are going to add a sound effect.

Adding a sound effect

You want a sound effect that is pretty close to a lightning strike sound. Scratch doesn't seem to have one but you can choose something else that is close. Go to the pink sound blocks. There is a block called *play sound meow until done***,** as shown in Figure 68.

Figure 68: Sound block used to issue sound instruction

Notice the block has a dropdown arrow. When you click it there are no more presets but you get the choice to record. You are not going to do that. Instead, go up to the 'Scripts, Costumes, Sounds' tabs and click on 'Sounds', as shown in Figure 69. Choose **New sound** from library (as shown in Figure 70), go to **Effects**, click on **rattle** and choose **OK** at the bottom right of the window.

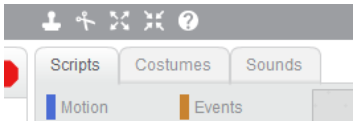


Figure 69: Use Scripts, Costumes and Sounds tab to access sounds

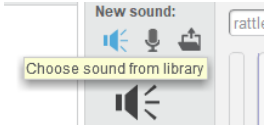


Figure 70: Function to select New sound from a sound library

Click the Scripts tab again and when you go to **play sound meow until done,** click the down arrow and 'rattle' will come up as a choice – select it, as shown in Figure 71. You want to place the **play sound** block after the Shocked cat appears. It will play until the loop is complete (until done).

Figure 71: Play sound block includes a choice of sounds

Your code should look like Figure 72.

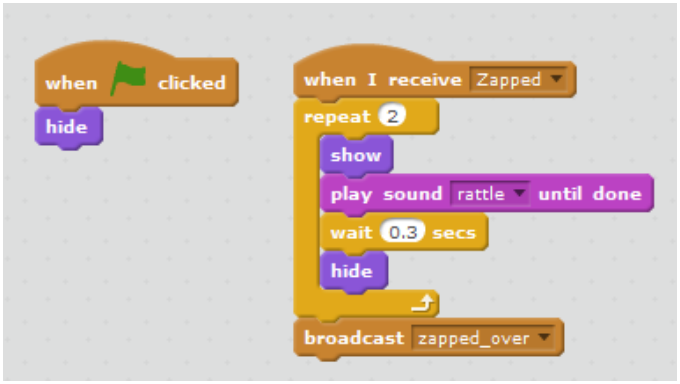


Figure 72: Scratch code for sound effect for Shocked cat

To do: Try this code out now. What needs fixing?

Placing a second sprite exactly where a first sprite is located

Did you notice that the Shocked cat does not appear where you want it to be? This is because you have not told it where to go, so it just goes where you originally placed it. You can fix this quite easily but it takes a few *variables*, as shown in Figure 73.

Your variables are going to store the current x and y coordinates of Cat1. If you tell the Shocked cat to go to the same coordinates, then the program should look much better.

If you also need to make Shocked cat about the same size as Cat1, do that now.

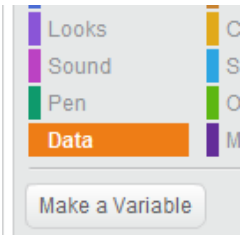


Figure 73: Scratch options for making a variable

Creating some variables

Make sure you are in the Cat1 code area. Select **Cat1** in the Sprites area (blue box around it) and all your other code should appear.

Go to the *Data* blocks. This is where you can create a variable. You will make three variables. Click on **Make a variable** and name it 'counter'. Make sure that **For all sprites** has the filled in circle and click **OK**, as shown in Figure 74. A new variable will appear within the *Data* blocks called 'counter'. You will use this one very soon.

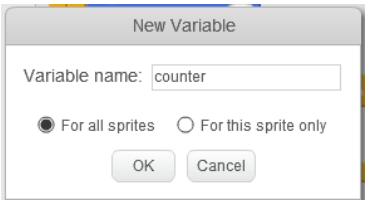


Figure 74: Scratch example of creating a variable for all sprites

Click on 'Make a Variable**'** again and this time call it 'xpos' (for x position). Again, make it **For all sprites**. Do the same a third time, this time calling it 'ypos' and making it 'For all sprites'**.**

Your three *variables* should look like Figure 75. Notice that each has a little grey box next to it. This is how variables are depicted in Scratch.

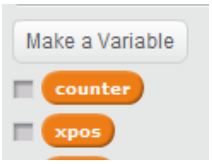


Figure 75: Scratch example of creating three variables

Setting the variable to a known location

Luckily for you, Scratch always knows the location (x and y positions) of a sprite. All you need to do is set your variables ('xpos' and 'ypos') to the current x position and y position of Cat1**.** In the *Data* block, choose the block that says **set counter to 0.** Note there is a drop arrow next to counter. You need two of these blocks, so you can drag one into the Cat1 coding area and then right click to duplicate it.

So far, your code should look like Figure 76.

Figure 76: Scratch example of setting x and y variables to current positions

Go into the *Motion* blocks and look at the very bottom. What do you notice?

There are three items that also look like variables – called ‘x position’, ‘y position’ and ‘direction’, as shown in Figure 77. These will give you the current x and y coordinates for the sprite you are coding. Drag the blue **x position** block to where the 0 currently appears in the **set xpos to 0,** as shown in Figure 78. You can do the same with the **set ypos to 0** block.

Figure 77: Scratch example of motion blocks for position and direction

Now your blocks look like Figure 78.

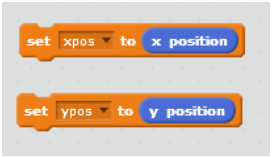


Figure 78: Scratch example of setting variables to known locations

You are now going to add these blocks to the bottom of the first block of code you created, making it look like Figure 79.

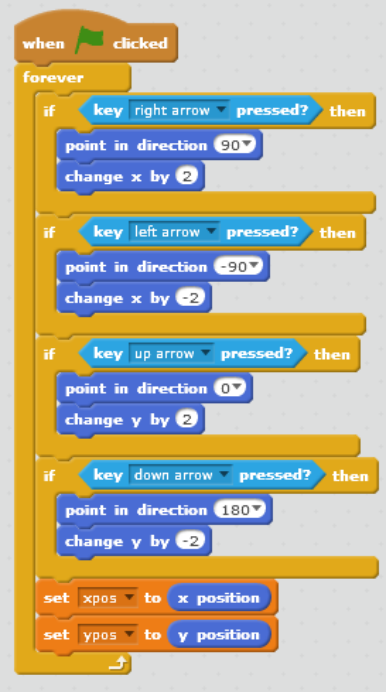


Figure 79: Scratch example adding variables to existing code

All you need to do now is go into the Shocked cat coding area (choose it from the sprites so it has a blue box around it and its code appears). From the *Motion* block set (dark blue) choose the block that looks like Figure 80.

Figure 80: Scratch example of Motion block with coordinates

Remember you have created two variables that are holding the current x and y coordinates of Cat1. You are going to use them now.

Go into **Data block** and choose the variable **xpos**. Drag it into the blue **go to x:** block Figure 81.

Figure 81: Scratch example of Data block including a variable

Do the same thing with the **ypos** variable. You should have a block of code that now looks like Figure 82.

Figure 82: Scratch example of Data block including two variables

Now think about where it should go in the Shocked cat code. It should go to that location as soon as the 'Zapped' message is broadcast. It will still be hidden at this stage, but that is okay. It will become visible very soon after it gets there. Your code for the Shocked cat now looks like Figure 83.

Figure 83: Scratch example of code for shocked cat

Note: If you made your 'xpos' and 'ypos' variables 'For this sprite only', then you would not see them in the Shocked cat sprite coding area.

To do: Try out your program now and see where Shocked cat appears.

You might see the two variables ‘xpos’ and ‘ypos’ appear at the top of the stage area, like that shown in Figure 84. You don’t want this, so you can simply right-click on them and choose **hide** and they will disappear.

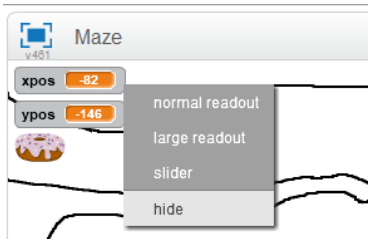


Figure 84: Right-click and select ‘hide’ to remove this function

The other variable you created, ‘counter’, is going to be used to count up how many times the track edge is touched. In this case you do want the variable to appear at the top of the stage area.

The best place to put the code for this is when ‘Zapped’ is broadcast, as it is broadcast each time the edge is touched. So still in the Shocked cat coding area, go to **Data blocks** and choose the **change counter by 1 block**. Place this block after (or just before) the broadcast ‘zapped_over’ command at the end of the code. Your code should now look like Figure 85.

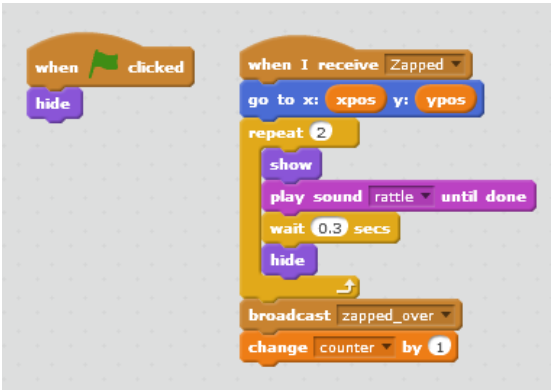


Figure 85: Code for Shocked cat incorporating the counter variable from Data blocks and broadcast command

We are now just about finished.

To do: Try out your program now and see if the counter works accurately.

There are only a couple more quick things to do to make the code complete. You have finished with the Shocked cat code, so go back to the Cat1 coding area by choosing its sprite in the sprites area.

Setting up a constant start

You want to make sure that the program starts exactly how you want – Cat1 at the bottom, facing right and the counter set to 0. As you want these things to occur right when the program starts, you need to put the code before the *forever loop* in the first block of code. From the *Data* blocks you can choose **set counter to 0** and just place it at the top of the code just like Figure 86 (page 43).

Figure 86: Placement of 'set counter to 0'
to start program where required

Now drag Cat1 to its starting position. You will notice that at the bottom of the stage area is an x and y coordinate readout, as shown in Figure 87.

Figure 87: Location of x and y coordinates for Cat1

This tracks the mouse location but you can take a good guess about Cat1's actual location. When you drag Cat1 to its starting position the **x** readout is about **–220** and the **y** readout is about **–150**. You will use those numbers to set the initial (home) position of Cat1. In the blue *Motion* blocks you need to choose the block that says **go to x: (number) y: (number)**. Replace the numbers in the ovals with –220 and –150. You also need to set up the direction by choosing the **point in direction 90** block and just place it without change into the first block of code. That block of code now looks like that shown in Figure 88 and is finished.

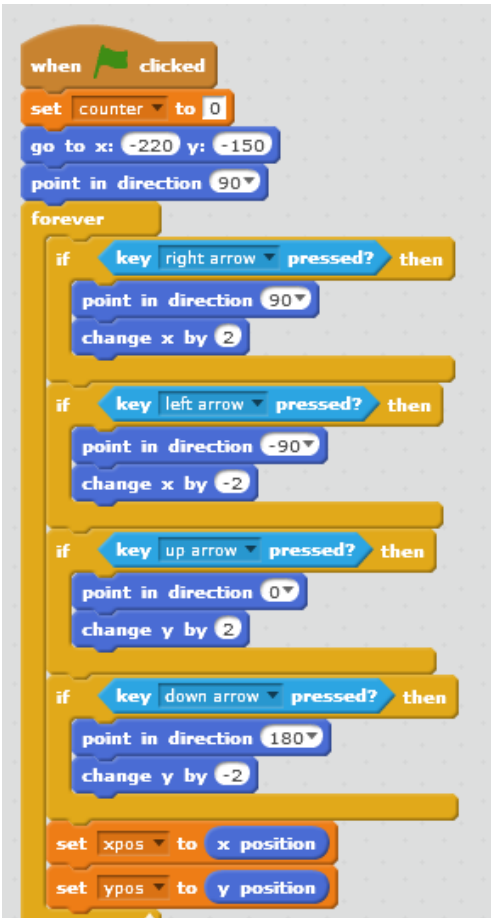


Figure 88: Completed set of code
for Cat1

To finish off the game, have Cat1 say something when it reaches the donut. Then you will stop the program completely.

You will use the second block to house this little bit of code. It could probably go in the first block, but this way the blocks will be about the same length and you can get Cat1 to do something if it is not touching the edge but is touching the donut.

To do: Try working out how to do this last little bit. The code is in Figure 89 if you get stuck.

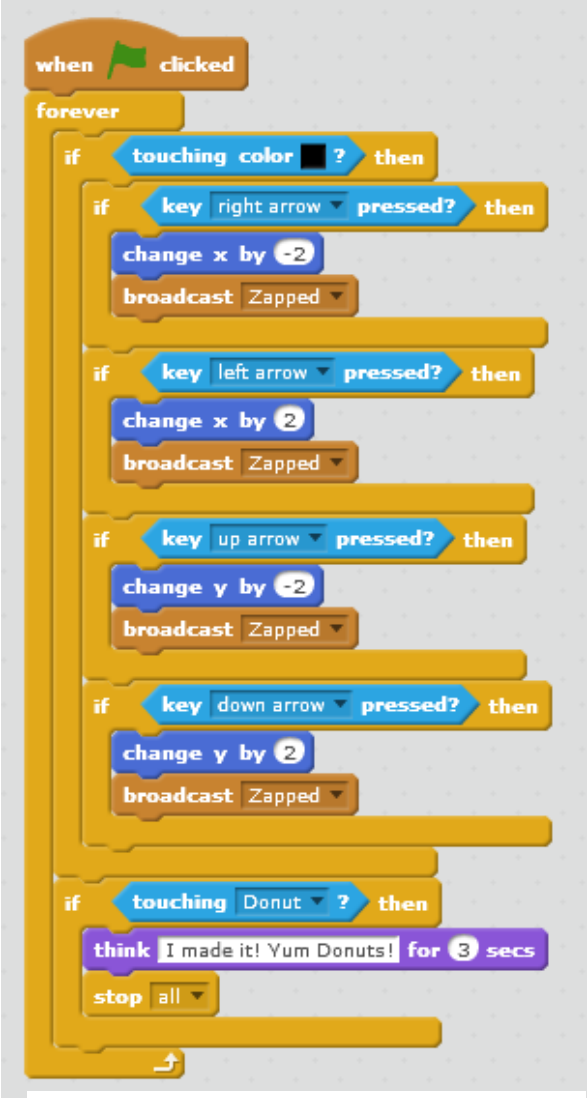


Figure 89: Completed set of code including dialogue when Cat1 touches the donut

Note where the new *if touching Donut then* block is located. It is at the same level as the *if touching color then* block. It cannot be inside that block as that scenario (touching both the edge and the donut) might not happen.

Run the code to see if it works. Obviously, you could add quite a bit more to this game to make it more challenging – more levels, limited lives, timers. Try some!

You have learnt quite a deal more about Scratch with this final fun exercise. Hopefully you can see that it could be used to extend the wire loop materials (or engineering) project that is commonly done by students at this level. If you are not using the Australian Curriculum, please refer to the specific engineering systems requirements for your curriculum.

SECTION 3: EXTENSION EXERCISES, RESOURCES AND ASSESSMENT

This section provides two short exercises that extend advice provided in Section 2, a list of resources, including projects and online coding resources that feature other visual programming languages, as well as assessment advice.

Extension exercises

The following two extension exercises enhance your skills in controlling movement and playing sounds.

Controlling movement

This exercise extends what is covered in Section 2: Creating a game. Controlling movement has great applications in learning mathematics, designing games and animation, as it can be very precise.

Choose **File/New** to start a new coding session. Select **Stage** and click on the left-most icon underneath to select a **new backdrop** from the library, as shown in Figure 90.

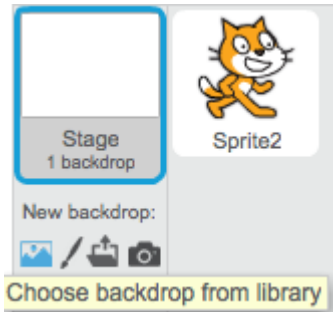


Figure 90: Scratch has a range of resources, backdrops, sounds and sprites in its library

Choose **xy-grid**. Figure 91 shows the results of that backdrop.

Figure 91: The xy-grid is useful for planning moves and for mathematics

Now your stage looks like a Cartesian plane with horizontal and vertical number lines.

Click on any of the *Motion* (blue) commands to see what they do.

You can always get back to where you started (the origin), the right way up, by clicking on ‘point in direction 90’ and ‘go to x:0 y:0’, as shown in Figure 92.

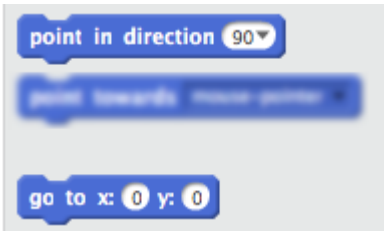


Figure 92: Scratch uses 0,0 as the origin (centre of the stage) and 0°, 90°, 180° and 270° for north, east, south and west, respectively. Just like a compass, 0° is the same direction as 360°

Playing sounds

Have fun exploring sounds (use headphones) and costumes. Combine these to make something interesting happen. Also try using the *Broadcast* blocks – they add a whole new dimension to what you can do with block coding.

Resources

You don’t have to look far in Scratch to find help. It has lots of tips that come up when you click on ‘Tips’ on the menu bar, as shown in Figure 93. These help you go through projects step by step – a great way to challenge students if you are not so sure yourself! There are other tips as well, such as creating effects, animation and games; telling stories, and working with music.

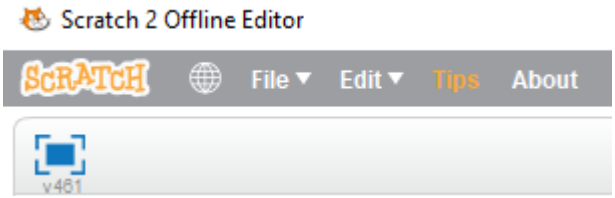


Figure 93: You can access lots of tips from the menu bar

There is also a tab that provides a lot of information on every block in Scratch, as shown in Figure 94 (page 48). If students come to you asking about how, for example, *broadcast* works, you can send them here in the first instance.

The tips are really useful in encouraging student-led activities as well as student self-sufficiency.

They are also useful for teachers to brush up on their skills.

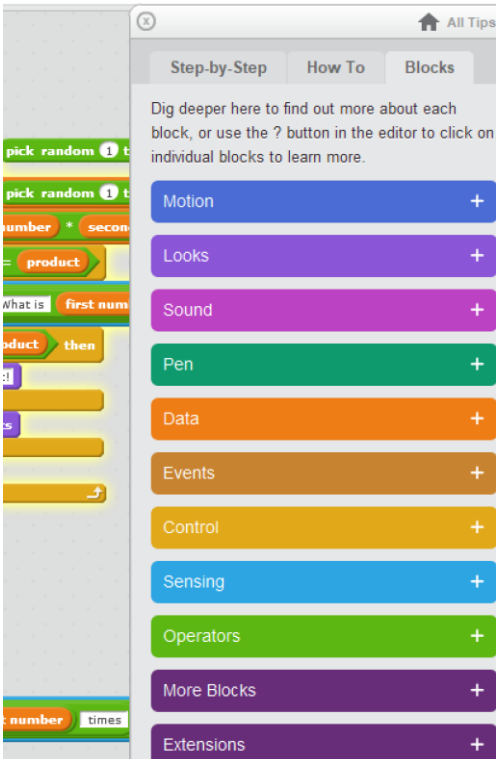


Figure 94: Scratch provides extensive information about each block

Apart from tips, there are thousands of projects available online where students can see how other programmers got things working.

These are available in the online version only under ‘Explore’, as shown in Figure 95 (page 49).

In Explore, students can load and play the programs that others have created.

Students can also click on 'See inside’ (the blue button) to see the code, as shown in Figure 96 (page 49).

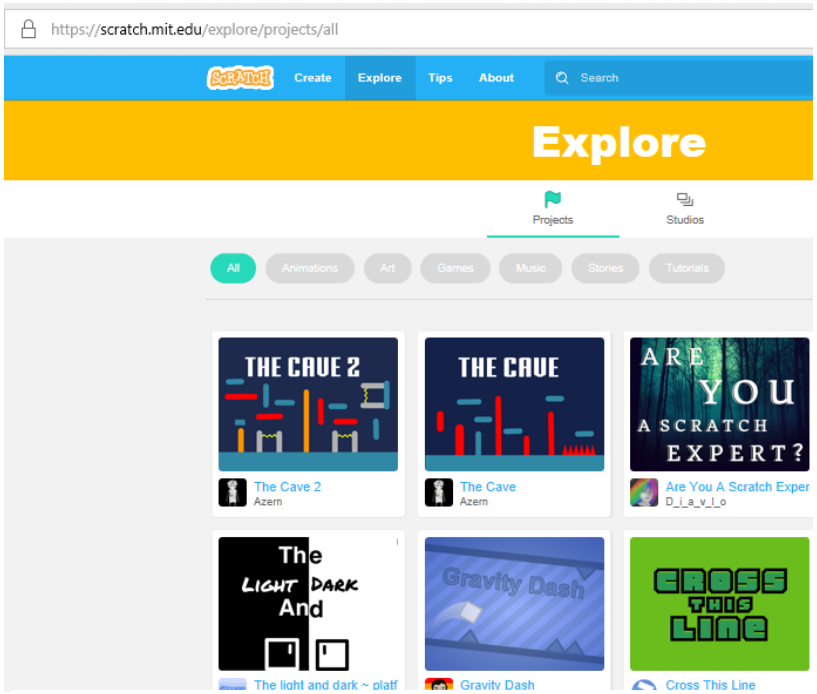


Figure 95: Select 'Explore' to access a significant bank of projects

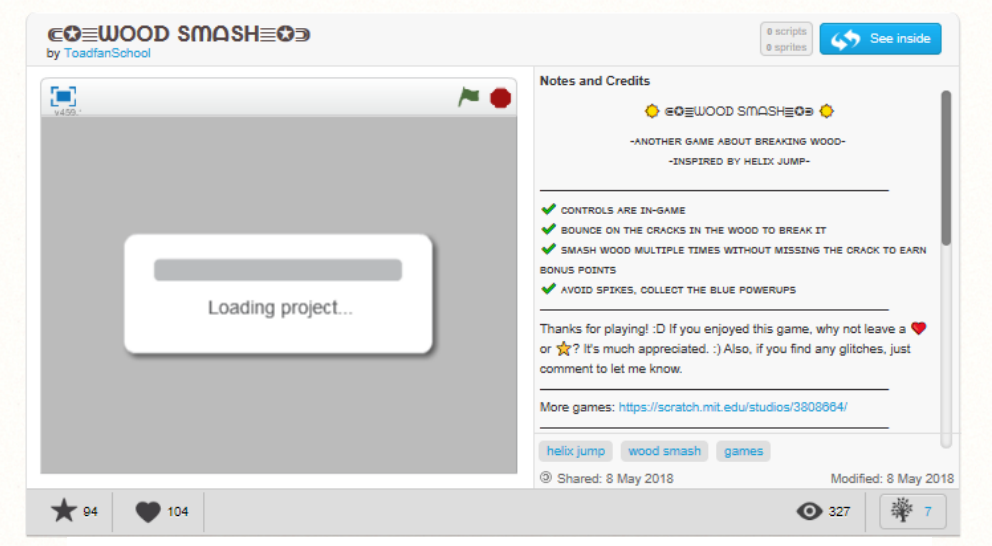


Figure 96: Select 'See inside' to reveal the code of projects

In Scratch, students are also encouraged to change the code of others (as long as credit is given). This is called *remixing* and it helps students to learn more about coding. Students can upload their own code or remixed code for other users to play and use. It is very collaborative and many students really enjoy the site. You might want to think about getting all your students to sign up. They can save their work online and continue working on it anywhere they have internet access and a computer.

Other visual programming languages

In addition to Scratch and Snap! (see page 4), Blockly is another suitable visual programming language for students. Grok Learning (**https://groklearning.com**) and Blockly (**https://blockly-games.appspot.com**) are two examples of sites that provide challenges and courses that guide the learner, showing the links between visual (block-based) and general-purpose (text-based) languages.

These resources are a bit more particular about variables than this resource; distinguishing between numerical and text (or *string*) types, as shown in Figures 97 and 98. Give them a go and make sure you try Snap! (**https://snap.berkeley.edu/**) It can be very useful when coding with external devices.

In Figures 97 and 98 you should be able to recognise the use of variables – note the use of 'print' rather than 'Say' in Scratch – and recognise a repeat loop.

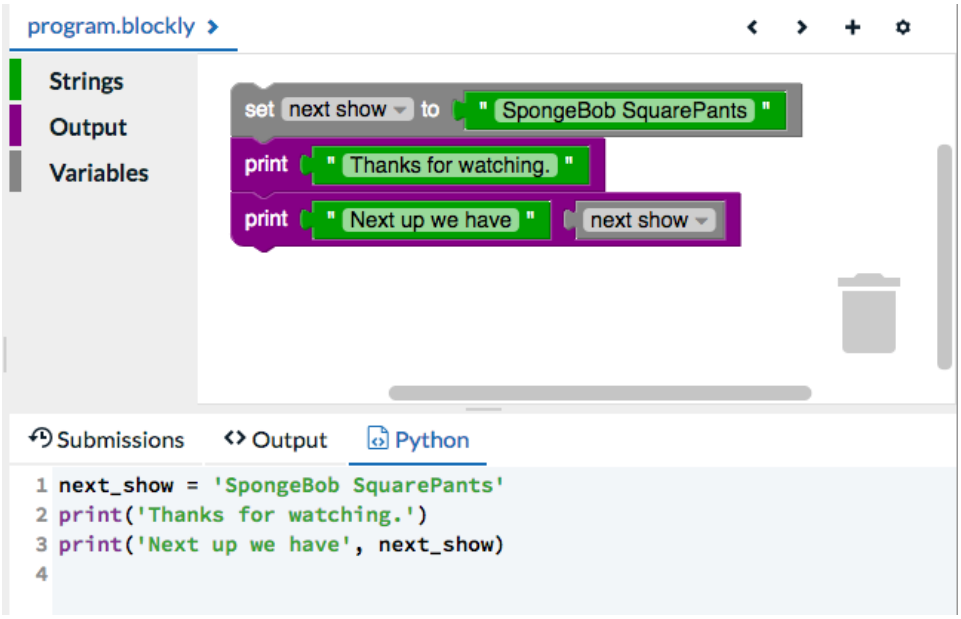


Figure 97: Example of Blockly code using variables

Figure 98: Example of Blockly code using repeat loop

Assessment

The Achievement Standards in the Australian Curriculum describe the expected learning of students at designated points. For Digital Technologies this is the end of a band of years. The Achievement Standards are benchmarks upon which you make judgments about students' progress and the results of such assessments should help frame your teaching and learning programs in order to progress students from one level of achievement to the next.

To see the Digital Technologies Achievement Standards for Years 3 and 4, Years 5 and 6 and Years 7 and 8 visit **www.australiancurriculum.edu.au** > F-10 > Technologies > Digital Technologies > Years 3 and 4 (or Years 5 and 6 or Years 7 and 8) > Achievement Standards.

The following extracts from the Digital Technologies Achievement Standards focus on the expected learning by students related to designing solutions and using a visual programming language to implement digital solutions. **Note:** While students at Years 7 and 8 should be studying a general-purpose programming language, they can still use a visual programming language to create digital solutions.

It should be noted that most of the content of this resource focuses on using a visual programming language to implement a solution – this forms just one part of students becoming innovative creators of digital solutions.

Years 3 and 4 Achievement Standards

'Students define simple problems, design and implement digital solutions using algorithms that involve decision-making and user input.'

Years 5 and 6 Achievement Standards

'They incorporate decision-making, repetition and user interface design into their designs and implement their digital solutions, including a visual program.'

Years 7 and 8 Achievement Standards

'Students design user experiences and algorithms incorporating branching and iterations, and test, modify and implement digital solutions.'

Assessment opportunities

When creating assessment opportunities, it is important that there are direct connections between what is taught (as stated in the Content Descriptions) and what knowledge and skills are being assessed. The Content Descriptions form the boundaries, in terms of breadth and depth, of the Achievement Standards, and they should be foremost when determining what evidence of student performance needs to be captured.

Listed are some assessment suggestions to help you measure student progress.

- Judicious use of Dr. Scratch (**http://www.drscratch.org**), focusing on intrinsic documentation, use of meaningful variable names, use of loops rather than repetitive statements. Remember, loops minimise the number of statements, hence decrease the chance of errors.
- A screen recording where the coder shows how the code works and the contribution made by some parts.

- Get Year 5 students to design a game for Year 2 students and then have the Year 2 students provide feedback to the coders. Guidance on the scope of feedback should be provided to the Grade 2 students.
- Journal entries of the steps taken and changes made in developing codes as well as algorithms. This might be a digital journal using screen shots, where appropriate.