Making an Interactive Quiz, with Scratch & MaKey MaKeys

Physical Computing

# Getting Started

In this activity we will create an interactive quiz, that combines Visual Programming and Physical Computing. This aim of this activity is not to give an in-depth introduction to these two concepts, but to give an example of what you can do in these areas with two beginner-friendly technologies: Scratch and MaKey MaKey. We have included some links to some more in-depth introduction to programming concepts in Scratch on the CS4S Primary School website.

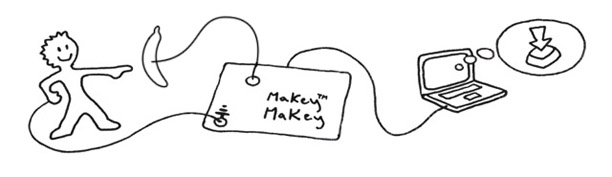
This activity also aims to be an example of the prototyping process. Although we cannot replicate this process exactly in this activity, we try and take an “incremental” approach. By an “incremental” approach, we mean that we get the essential features working first, then improve on these as we get further through the activity.

We recommend you work in pairs during this activity, working from one lab computer. You will need a bit of space on your desk as well, to make room for the MaKey MaKey, buzzers and wires. This activity and a copy of the finished Scratch program will be available on the CS4PS website after the workshop.

There are some sound effect files to use in this activity. These can be downloaded from the CS4SPS website. Download the resources from the “Making an Interactive Quiz Resources (Zip)” link on this session’s page. Once this has finished downloading, extract the files to the lab computer’s Downloads folder.

## MaKey MaKey

The illustration below shows a simple explanation of how to use a MaKey MaKey. When the person touches the banana, they complete a circuit. This triggers one of the actions on the MaKey MaKey (e.g. the Up arrow), which then happens on the connected computer.



Connect the grey USB cable from the MaKey MaKey to your PC. The lights on the MaKey MaKey will flash, to indicate that it has received power.

To test that the MaKey MaKey works correctly, put your thumb on the earth bar (the silver metal rectangle at the bottom) and touch one of the buttons (e.g. the SPACE or CLICK). When you do this, the green light near the button should light up and the action you tapped (e.g. the CLICK) will be performed on the connected computer.

You can clip alligator clips to whatever objects you can think of, and it will work with the Makey Makeys (as long as the object is conductive). For example, Makey Makeys have been used to make a piano with bananas, and game controllers with playdoh.

## Scratch

If you want to make a program that interacts with a Makey Makey, you can use any programming language you’d like. We’ll use Scratch in this session, because we don’t have to install anything and it’ll be straightforward for us to write code that reacts to different keys being pressed.

To get started in Scratch, create a new project (in Scratch, go to *File* > *New Project*).

# Numbers Quiz

In this section we will create a quiz game, where a random number will appear, and you will have to press the correct buzzer before the time runs out.

## A Random Number

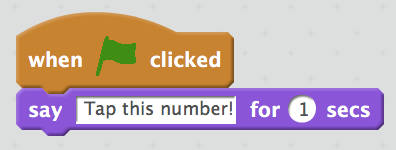
Firstly, we will make our Scratch program show an Introduction message and show a random number.

First, make sure the cat sprite (**Sprite1**) is selected. If a sprite is selected, there will be a blue rectangle around it in the Sprites area.

From the **Events** block section, drag the  block into the Scripts Editor.

Next, drag a  block from the **Looks** section under the block, so that it clicks in place.

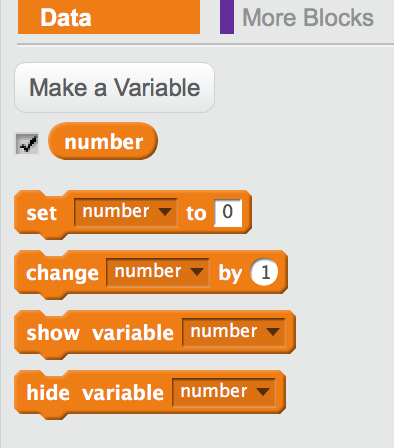
Change the *“Hello!”* text to say *“Tap this number!”* and change the *“2”* secs to *“1”*. The scripts should now look like this:



When you run the program, by clicking the green flag, the cat should say that message.

We need to make a variable for the random number that is generated. To make a variable, go to the **Data** section in Scripts, and click the **Make a variable** button. Type in *“number”* as the variable name, leave the **For all sprites** option checked and click OK.

After doing this, you should see 5 orange blocks for the *“number”* variable, as seen in the image below:



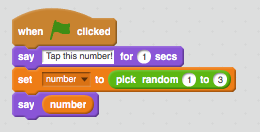
Drag the **set number to 0** block, and place it under the **say…** block, so that it clips into place.

Next, drag the **pick random from 1 to 10** from the **Operators** section into the **0** spot in the **set number to 0** block. Change the 10 to 3. This will mean that a random number between 1 and 3 will be picked.

Now drag a **say “Hello!”** block from the **Looks** section and place it under the other blocks.

Then drag the **number** block from the **Data** section into the *“Hello!”* part of the say block.

Your scripts should now look like the image below:



When you run the program, the cat will say a message *“Tap this number!”* and then say a random number. If you try this a few times, you’ll notice that this random number changes each time.

## Handling Input

The next step is to add sound effects that play when you press one of the buzzers. To upload these, click on the Sounds tab, highlighted in the image below:



Click the upload file button, which looks like this: , and a dialog for uploading the file will open.

Upload the *“correct\_answer.wav”* and *“wrong\_answer.wav”* files from the resources zip file you downloaded earlier. You should see these sound clips appear in the **Sounds** menu, under the *“meow”* sound clip.

Now, return to the **Scripts** tab. From the **Control** section, drag a **wait until** block and place it at the bottom of your blocks (after the **say number** block).

Place a **key “space” pressed?** in the **wait until** block’s slot. On the **key “space” pressed?,** select “any”from the dropdown (where “space” is). The bottom 2 blocks should now look like this:



We are now going to add blocks that will decide whether the player has pressed the correct buzzer. To do this, we need to use **if then else** blocks. These blocks are used to run different blocks, depending on some condition. In this quiz we want to check the random number that has been generated (for example, 1) and check whether the player has pressed the correct buzzer.

When the player presses the 1st buzzer, the computer will receive the same signal it would if you had pressed the Up Arrow. So, our first step is to check whether the number to be tapped is 1, and if the player has pressed the Up Arrow.

Firstly, drag an **if then else** block, from the **Control** section, and place it at the bottom of your other blocks in the Script Editor.

Next, drag a **=** block from the **Operators** section, and place it somewhere else on the Scripts Editor, so that it doesn’t clip into place into the other blocks. The **=** block has 2 slots. In the first one, we want to place a **number** block, from the **Data** section. In the 2nd slot you need to type 1.

Now, drag a key **“space” pressed?** block onto the Scripts Editor, away from the other blocks. From the dropdown that says “space”, select “up arrow” instead.

Then drag an **and** block from the **Operators** section, and place it away from the other blocks as well. Then drag the **=** block into the 1st slot of the **and** block, and the **key “up arrow” pressed?** into the 2nd slot of the **and** block. These steps are show in the below image:



Next, drag a **play sound** block, from the **Sound** section, onto the Scripts editor, away from the other blocks. From the dropdown that says **meow**, select **correct\_answer** instead.

Now drag the **and** block you created in the last step into the empty slot of the **if then else** block. Then drag the **play sound correct\_answer** block under the **if** part of the block, as shown below:

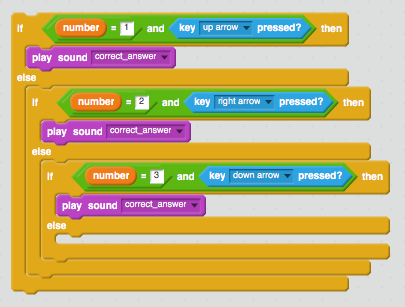


The following happens when you run the program and press a key on the keyboard, after the cat has said a random number. The **if** block checks if the number the cat has said is 1 and if you have pressed the up arrow button. If that is the case, then the blocks in the 1st section of the **if then else** will run (the “correct\_answer” sound will play). If the number isn’t 1 or you pressed a different button, the blocks in the 2nd section will run. At the moment, there are no blocks in the 2nd section of the **if then else** block. We will add some blocks to this section now, which will control what happens when we press the other arrow buttons.

We need to repeat the steps followed above for the other correct “pairs”, for example when “2” is the generated number and the player has pressed the “right arrow” button, and so on. To do this, we can duplicate the **if then else** block, by right clicking on it and selecting duplicate. We then change the “1” to “2”, and the “up arrow” to “right arrow”. This duplicated block is then placed in the 2nd section of the **if then else** block, as shown below:



The duplicated block is surrounded by a blue rectangle in the image above. We repeat this to check when the random number is 3 and the down arrow button is pressed. After duplicating the **if then else** block twice and changing it for these numbers, and placing them in the correct spots, the blocks will look like this:



The blocks placed in the last else section will run if all of the **if** blocks’ conditions are false. So, in this example that will be when the player has pressed the wrong key. A noise indicating that the answer was incorrect should play when this happens.

To make this sound play, drag a **play sound** block into the 2nd section of the last **if then else** block, and select “wrong\_answer” from the dropdown. The last **if then else** block should look like this now:



Now when you press the incorrect key, for example when you press the Up Arrow after the cat says to tap 2, the “wrong\_answer” noise will play. Try this out and see what happens when you press the different arrow keys. Note that each time you would like a new number to appear, you will have to click the  block or click the green flag.

Next, we’ll make the buzzers for interacting with the quiz.

## Making the Buzzers

To interact with our quiz, we’ll build 3 simple buzzers using cardboard and aluminium foil. To start with, we will write a number (1, 2, and 3) on each of the buzzers. The image below shows some example buzzers we have made:



To make the buzzers, cut out some rectangular pieces of cardboard, cover them in aluminium foil, and draw the appropriate number on them. You may have a more creative approach to make them than this, so feel free to make them in a different way to above.

Next, connect up the buzzers to the MaKey MaKey, using the wires. We want to connect each buzzer up to the following buttons on the MaKey MaKey:

* **1** to the **Up Arrow**
* **2** to the **Right**
* **3** to the **Down Arrow**

Then attach one of the ends of another wire to one of the slots on the MaKey MaKey’s earth bar. The person that plays the quiz will have to hold the other end of this wire (to complete the circuit).

Now, try touching each of the buzzers, to make sure that the correct arrow button lights up. Note that the person pressing the buzzers will also have to be holding the end of the alligator clip connected to the earth bar on the MaKey MaKey.

If the buttons don’t light up, double check that the clips are clipped to foil on the buzzers and that the alligator clips are clipped into place on the MaKey MaKey.

Now, run the program and see what happens when your press the correct and incorrect answers. When the cat says a number (for example: 1) and you press the appropriate buzzer, you should hear the “correct\_answer” sound play.

In the next section we will add a time limit to the game.

## Adding a Timer and Messages

In Scratch, you can show the timer by going to the **Sensing** section, and ticking the checkbox next to the **timer** block, as shown below:



When this is ticked, you will see a box on the top left of the canvas, that looks like this: 

The count on the timer will increase as time goes on, even if you are not running your program. Because the timer works this way, we will have to reset it (set it back to 0) every time the program runs. To do this, drag a **reset timer** block, from the **Sensing** section, and place it between the **say number** and **wait until…** blocks. Now when you run the program the timer should reset back to 0.

We now need to add a time limit to the game. We’ll use 1.5 seconds, but you can adjust this to any number you like to make it less or more challenging. To do this, we’ll change the **wait until…** block.

Replace the **key “any” pressed?** in the **wait until…** block with an **or** block, from the **Operators** section. The **key “any” pressed?** will move away from the other blocks, you can leave that where it is for now.

Drag a **>** block from the **Operators** section onto the Scripts Editor, but don’t connect it to the other blocks. Put a **timer** block, from the **Sensing** section, into the 1st slot of the **>** block. In the 2nd slot, type in 1.5 (or however long in seconds you want the timer to last).

Now, place the **key “any” pressed?** into the 1st slot of the **or** block. Then place the **>** block into the 2nd slot of the or block. These steps are shown in the image below:



This means that the blocks below the **wait until** block will not run until the player presses a buzzer or 1.5 seconds elapse. If you run the program and wait 1.5 seconds, you should hear the wrong answer sound effect play.

At the moment, the wrong answer sound effect plays whenever the player runs out of time or when the player presses the wrong buzzer. We will now add messages to tell the player which of these has happened, and to congratulate them when they press the right buzzer.

To add a message that tells the player they have pressed the wrong buzzer, do the following steps. Drag a **say “Hello” block for 2** secs block from the **Looks** section, and place it under the **play sound “wrong\_answer” block**. Change the *“Hello!”* to something like this: *“You pressed the wrong buzzer :(“* and the *“2”* to a *“1”.*

Now, do the same as the previous steps above, but place the **say…** blocks under the 4 **play sound “correct\_answer”** blocks**,** and change the text to *“Good work!”.*

Next, we need to add blocks to check when the timer has run out. To do this, we first drag an **if then else** block onto the Scripts Editor, away from the other blocks. Duplicate the **>** block you used in the **wait until** block and place this in the slot in the **if then else** block.

Next, add a **play sound “wrong\_answer”** and a **say “You ran out of time :(“ for 1 secs”** block in the 1st section of the **if then else** block, as shown below:

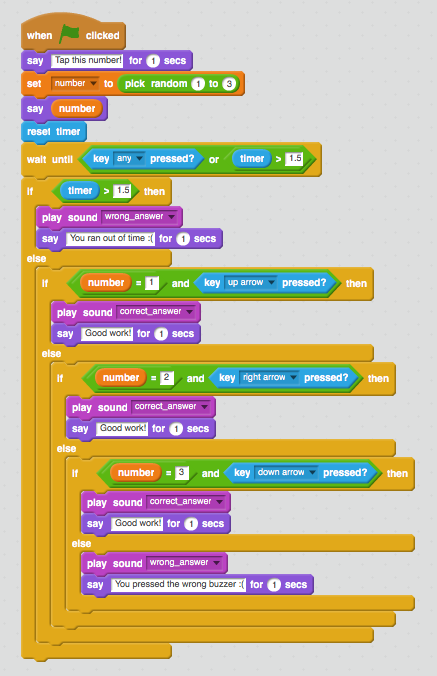


We now move all of the **if then else** block that is after the **wait until** block into the 2nd section of the **if then else** block above, as shown below:



Finally, move the **if then else** block we just added, placing it after the **wait until** block.

The scripts should now look like this:



When you run the program a sound effect should play and the cat should say a message whenever you:

* Run out of time or
* Press the right buzzer or
* Press the wrong buzzer

In the next section, we will add a score and a game loop, so that we don’t have to keep running the program over and over again to get a new number.

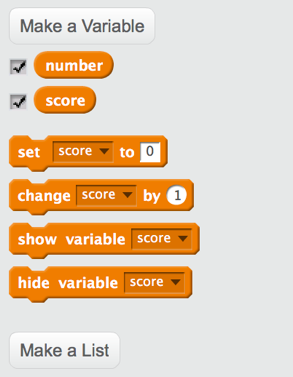
## Adding a Game Loop and Score

We will now add a loop that will run the quiz 5 times before the program stops. From the Control section, drag a **repeat 10** block towards the blocks and place it under the  block. The **repeat** block should wrap around all the other blocks that are there. Then change the *“10”* to *“5”*, this will make all the blocks inside the **repeat** block run 5 times before the program ends. Your first few top blocks should look something like this:



Now when you run the program, you will be asked to tap a number 5 times before the program stops.

Next, we will add a score. To keep track of the score, we will have to use a variable. To add this variable, go to the **Data** section, then click the **Make a Variable** button. Type in *“score”,* keeping the **For all sprites** checked and click OK. The **Data** section should now look like this:



Every time we start the game, we want to reset the score back to 0. To do this, drag the **set score to** **0** block from the **Data** section and place it after the  block.

Next, we need to place a **change score by 1** block after each of the say **“Good work!” for 1 secs** blocks. This will increase the player’s score every time you press the right buzzer before the time runs out.

When you run the program, you should see the score variable, which will be on the top left of the canvas, increasing by 1 every time you press the right buzzer.

Now we will add a message that will display the player’s score after the 5 rounds have finished. To do this, put a **say “Hello” for 2 secs** block at the bottom of your blocks.

We will use a **join** block, from the **Operators** section, to make this message. Drag a **join** block onto the Scripts Editor, and type *“Your final score was: “* in the 1st slot. Drag a **score** block, from the **Data** section, into the 2nd slot of the **join** block. Then, drag this **join** block into the **say “Hello” for 1 secs** block. These steps are shown in the images below:



Good work! You have successfully made the Numbers quiz. In the next section, we will change the quiz to use colours instead of numbers.

# Choosing the Right Colour

In this section we change our quiz to show a random colour. A coloured circle will appear behind the cat, as shown below:



The player then has to press the buzzer with the correct colour (blue in the example above), before the timer runs out.

We’ll use the following colours for this game:

1. Blue
2. Yellow
3. Orange

## Adding Colours to the Buzzers

Using the post-it notes we have given you, add colour to the buzzers. Note that because the post-it notes aren’t conductive, you will not be able to press on the post-it note to activate a key press. You will have to press on the foil part on the buzzer for the correct keypress to be sent to the computer. This is how we made our buzzers, note that the colours match up with the numbers above:



## Drawing a Random Colour

Before we start drawing the coloured circles, delete the **say number** block, by right clicking on it and selecting *delete*. If you accidentally delete other blocks, you can use *Edit > Undelete* to bring these back.

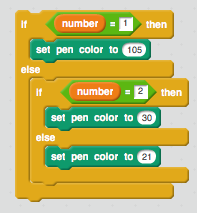
You should also change the message in the first **say** block, so that it says *“Tap this colour!”* instead of *“Tap this number!”.*

We are going to use blocks from the **Pen** section to draw the different coloured circles.

Firstly, drag a **clear** block and place it above the **say “Tap this colour!” for 1 secs***.* Each time the loop runs, this will remove the circle that was previously drawn, before drawing the next one.

Under this **clear** block, place a **set pen size to 1** block and change the “1” to “200”.

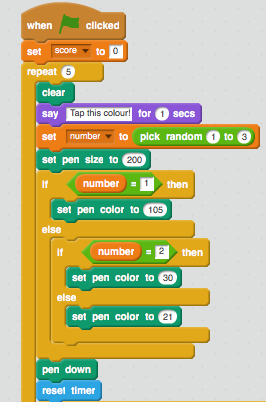
Now, we need to change the pen colour depending on which random number is generated. To do this, we will use **if then else** blocks. We will use them in a similar way to the way we checked if the player pressed the right buzzer.



The image above shows the way we will decide which colour circle to draw. Add these blocks underneath the **set pen size to 200** block.

Finally, place a **pen down** block after this **if then else** block, which will draw the circle.

The top part of these blocks should now look like the image below:



Click the green flag and try pressing the buzzers for the different colours. Is the timer too fast or too slow, how could you change the time the player is given to press the right buzzer?

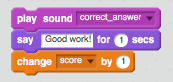
Good work! You have finished this activity. In the next section, we have included an extension activity and then some suggestions for extra features you can add to your game.

# Extension Activity: Code Smells and Procedures

We have included an extension activity in this tutorial, which you may like to complete if you finish the rest of the activity quickly. This activity explores *code smells*, gives an example of one of these *smells* (code duplication) and provides a solution to removing this *smell*.

## Code Smells and Duplication of Code

A *code smell* is a term that is occasionally used by Software Engineers to describe code that works as expected but that could cause problems in the future. An example of a *code smell* in Scratch is duplication of code blocks in several spots, which can make it easier to make mistakes when changing what a Scratch program does. Have you noticed any duplicated code in the quiz game’s code? These three blocks are repeated three times in our scripts and will run every time the player presses the right buzzer:



A recent study (available at this URL: [goo.gl/jzcbgY](https://goo.gl/jzcbgY)) of 12 year old students coding in Scratch explored how different types of *code smells* affected the students’ performance in coding exercises. One of the interesting findings of the study was that when students needed to modify scripts with duplicated code blocks, they made more mistakes than those that had to modify scripts without duplicated code blocks.

For example, say that you decided to change the message when a player gets a correct answer to *“You pressed the correct buzzer!”*, instead of *“Good work!”*.To make this change you would have to change the text in all three **say** blocks to this different text. It would be easy to accidentally miss changing one of these **say** blocks, and consequently there would be a mistake in the script. However, could we reduce the duplication of code and make it easier to change the script in the future?

## Creating a Procedure to Reduce Duplication

A rule of thumb when coding is that if you have the same blocks of code more than twice in your script, then you should put this code in a *procedure*, instead of duplicating the code. A *procedure* is a sequence of code, that can be run multiple times, that are commonly used to cut down on repetition of code. *Procedures* can be madein Scratch using *“Custom Blocks”*. We will create our own *“Custom Block”* called *“correct\_buzzer\_pressed”* that will be used whenever a player presses the correct buzzer, in order to remove the duplicated code and consequently remove a *code smell*.

In the *Scripts* tab, go to the *More Blocks* section. Click the **Make a Block** button, write *“correct\_buzzer\_pressed”* in the purple block and click *OK*, as illustrated below:



After doing this, a **define correct\_buzzer\_pressed** block will appear in your Scripts.

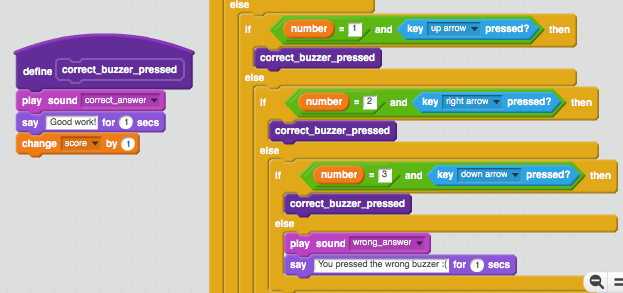
Drag the 3 blocks (play **sound correct\_answer**, say **Good work! for 1 secs** and **change score by 1**) into the **define correct\_buzzer\_pressed** block. This is illustrated below:



This means that every time the **correct\_buzzer\_pressed** block is run in the Scripts, the following actions will be taken:

* The *“correct\_answer”* sound will play
* The cat will say *“Good work!”*
* The player’s score will increase by 1

Now that you have created this *“Custom Block”*, you should replace all times those above three blocks are used with the **correct\_buzzer\_pressed** block, by dragging this from the *“More Blocks”* section. The section of blocks where this *“Custom Block”* is used and the **define correct\_buzzer\_pressed** block should now look like this:

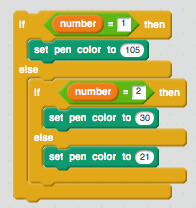


Now, if you needed to make a change, such as changing the text shown when the player gets a correct answer to *“You pressed the correct buzzer!”*, you would only need to do this in one place. Another example of a change you could make easier now than previously would be to increase the player’s score by 2 whenever they press the correct buzzer.

## Why Worry about Smelly Code?

It is important that students can create code that is clear and that does not contain many *code smells*, much like it is ideal that students are able to write stories and essays that are clear and that can be easily understood by others. Writing code without *smells* is not only important because it is good practice, particularly for those students who use coding in their future careers, but also because *non-smelly code* will be easier for the students’ future selves, their peers and their teacher/s to read and understand. For example, if a student is encountering an issue with their program and a teacher is trying to help them fix it, it will be easier for the teacher to guide the student if there is a limited amount of duplicated code.

Duplication of code is one example of a *code smell* but it is not the only type of *code smell*. Another example of a *code smell* that has been studied in Scratch programs in the previously mentioned study (available at this URL: [goo.gl/jzcbgY](https://goo.gl/jzcbgY)) is called *long method*. A *long method* is a way of referring to a long group of blocks. The main stack of blocks that is used in our quiz game could be considered a *long method.* To reduce the size of this stack we could move some parts of the stack into separate “Custom Blocks”. For example, the blocks that are used to determine the colour of the circle, which is shown below, could be moved into a “Custom Block” called *“choose\_pen\_colour”:*



Try and add the *“choose\_pen\_colour”* block to the quiz game. Do you think that this makes the code easier to understand?

You may identify *code smells*, or stacks of blocks that are difficult to understand, in your students’ Scratch programs. This is not necessarily a problem, and the identification and removal of a *code smell* is likely to create a valuable learning opportunity for you and your students. The process of rewriting code, which is called *refactoring*, is very common in Software Engineering and an important skill for anyone that writes code. Coding is often compared to writing, and refactoring could be seen as similar to editing an essay to improve sentence structure or grammar without changing the essay’s main arguments or overall structure.

# Extras

If you complete the main and extension activities, you may want to add some extra features to the game. A few examples of these features could be:

* Adding a counter for the current round, then using this in one of the cat’s messages. For example, at the start of each round the cat could say “Round N: Tap this colour!” where N is the number of the current round
* Giving the player 3 “lives”. After the player has answered 3 questions wrong, the quiz should end, rather than lasting 5 rounds.
* Make the program call out the numbers, colours or symbols the player should press, by using the play sound block. For this, you could use Scratch to record your own sounds or use sounds from the Scratch library.
* Adding another buzzer and colour so that you have 4 buzzers instead of 3

If you need any help adding of the above features, or would like to ask how you could implement your own ideas, please let us know.

# Acknowledgements

The following images and sound effects have been used as part of this activity:

* The MaKey MaKey Diagram is from: <https://wiki.scratch.mit.edu/wiki/How_to_Connect_to_the_Physical_World>
* The wrong answer noise is from: <http://freesound.org/people/themusicalnomad/sounds/253886/>
* The correct answer noise is from: <https://www.freesound.org/people/rhodesmas/sounds/320652/>