Australian **Computing** Academy

DT Mini Challenge

Intro to micro:bit

1. Displaying images and text
2. Buttons and gestures
3. Virtual Pet extensions

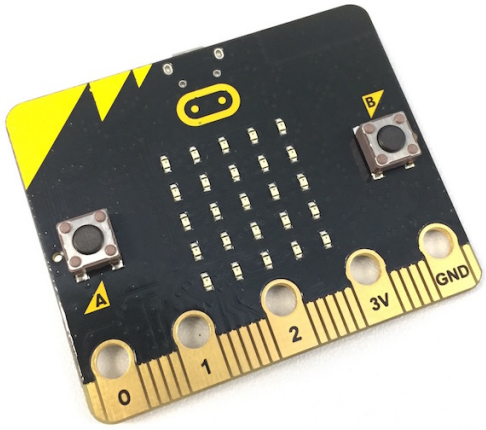
1

DISPLAYING IMAGES AND TEXT

1.1. Getting started

1.1.1. BBC micro:bit

The BBC micro:bit (https://www.microbit.co.uk/) is a tiny computer that runs the Python (https://microbit-micropython.readthedocs.io) programming language.



The micro:bit has:

- **a 5 x 5 display of LEDs** (light emitting diodes)
- **two buttons** (A and B)
- **an accelerometer** (to know which way is up)
- **a magnetometer** (like a compass)
- **a temperature sensor**
- **Bluetooth** (to talk to other micro:bits and phones)
- **pins** (gold pads along the bottom) to connect to other devices like screens, motors, buttons, lights, robots and more!

> **If you don't have a real micro:bit...**
>
> You can still do this course. It includes a full micro:bit simulator, so you'll be able to do everything you'd do on a real micro:bit!

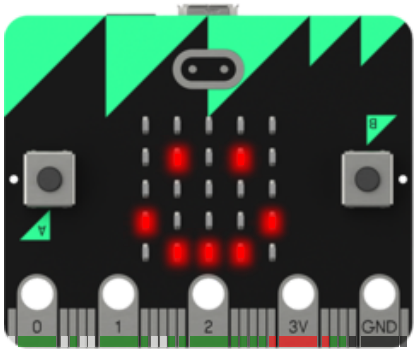
1.1.2. Hello, micro:bit!

Let's jump right in with a program...

Our first program is turning on a light (*light emitting diode* or LED). The micro:bit has 25 LEDs, so let's use lots of them!

Click run ▶ in the example below:

```
from microbit import *

display.show(Image.HAPPY)
```



Congratulations, you just programmed a micro:bit!

1.1.3. Importing microbit

Let's have a closer look at the first line (called a *statement*):

```
from microbit import *
```

`microbit` is a Python *module* (a library of useful code) that we use to control the BBC micro:bit.

The `*` (called an *asterisk*) means *everything*, so it imports all of the code from the `microbit` module.

This statement must be at the top of every micro:bit program. We'll include it for you, but don't delete it, or your programs won't work!

> **micro:bit vs. `microbit`**
>
> The BBC **micro:bit** is the device, and `microbit` is the Python module. Careful of the colon!

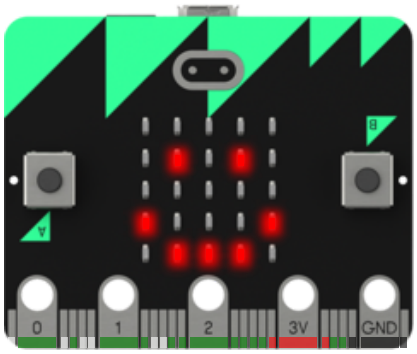
1.1.4. Using the microbit module

Everything we use in the second statement is imported from the `microbit` module:

- `display.show` is a *method* (like a command) of the `display` object, which controls the micro:bit's display.
- `Image.HAPPY` is a built-in image provided by the `Image` class.

We *call* (run) the `display.show` method by putting brackets after it. The image we want to display goes inside the brackets:

```
from microbit import *

display.show(Image.HAPPY)
```

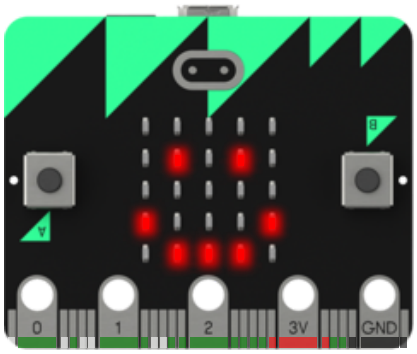


What is a statement?

A statement is the smallest stand-alone part of a program. It tells the computer to do something. Importing the `microbit` module and calling `display.show` are both examples of statements.

1.1.5. Problem: Happy micro:bit!

Write a program that shows the happy face on the micro:bit:



What, again? **Yes, now it's your turn to write it from scratch.**

If you're not sure how to start writing the program, go back a few pages and take another look at the notes.

How do I submit?

1. *Write* your program (in the `program.py` file) in the editor (large panel on the right);
2. *Run* your program by clicking Run in the top right-hand menu bar. The micro:bit will appear below, running your code. **Check that it works correctly!**
3. *Mark* your program by clicking Mark and we will automatically check if your program is correct, and if not, give you some hints to fix it up.

You'll need

program.py

```
from microbit import *
```

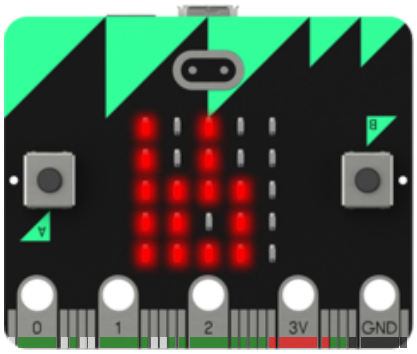
Testing

- [ ] Testing that the display is showing a happy face.
- [ ] **Congratulations, you've written your first micro:bit program!**

1.1.6. Problem: Your own Virtual Pet!

Now it's time to create your own virtual pet!

Write a program that shows a picture of a rabbit on the micro:bit:



To display the rabbit image you can write: `display.show(Image.RABBIT)`.

> **Microbit module**
>
> Don't forget to `import` the `microbit` module before you display the rabbit!

Testing

- [ ] Testing that the display is showing a rabbit face.
- [ ] **Congratulations, you've written your first micro:bit program!**

1.1.7. Downloading

If you have a real micro:bit you can download your code and run it in real life!

Click the Download button you will get a `.hex` file.

Take the `.hex` and drag it into the micro:bit – just like it's a USB drive.

It will take a few seconds, but once it's done you'll see your program running on the micro:bit!

You can also read our blog post with more detailed instructions (https://medium.com/p/b89fbbac2552).

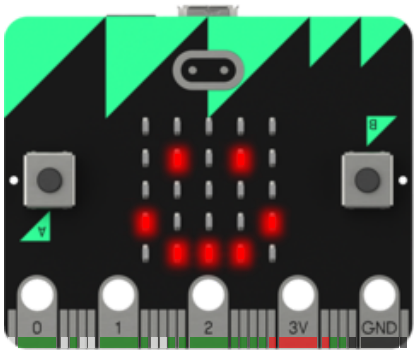
1.2. Writing micro:bit programs

1.2.1. HAPPY to SAD

We've seen that you can run the code examples in our notes. You can also edit them and play with the code!

Try changing the example below to `Image.SAD` and click run ▶ to see what it looks like:

```
from microbit import *

display.show(Image.HAPPY)
```



Click ↰ to swap to the original code. Click again to swap back to your modified version.

> **Play with the examples!**
>
> Try running and modifying (messing around with even!) **every** example in these notes to make sure you understand it.

1.2.2. More images!

There are lots of images included in `Image` for you to use. We've already seen `Image.HAPPY` and `Image.SAD`.

Here are some of our favourite images:

Name	Image
`Image.HAPPY`	
`Image.HEART`	
`Image.DUCK`	
`Image.PACMAN`	
`Image.ARROW_N`	
`Image.ARROW_E`	

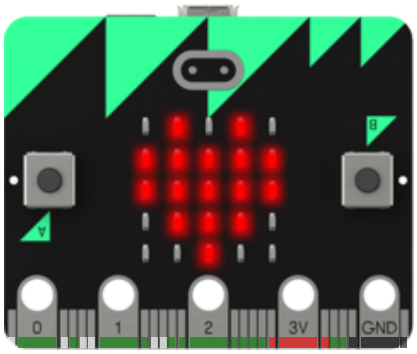
Name	Image
`Image.SAD`	
`Image.GIRAFFE`	
`Image.BUTTERFLY`	
`Image.GHOST`	
`Image.ARROW_S`	
`Image.ARROW_W`	

You can find the [full list here (https://microbit-micropython.readthedocs.io/en/latest/tutorials/images.html)](https://microbit-micropython.readthedocs.io/en/latest/tutorials/images.html).

1.2.3. Problem: From micro:bit with love

Do you have trouble thinking of a gift every year for Mother's Day? Let's use the micro:bit to create a Mother's Day card!

Write a program to display a picture of a heart using `Image.HEART`:



You'll need

program.py

```
from microbit import *
```

Testing

- [ ] Testing that the display is showing a heart.
- [ ] **Congratulations, what a great gift!!**

1.2.4. More pets!

The `microbit` module gives us a whole set of virtual pets we can use. We've already seen `Image.RABBIT`.

Choose your own pet from this list:

Name	Image
`Image.RABBIT`	
`Image.COW`	
`Image.DUCK`	
`Image.TORTOISE`	

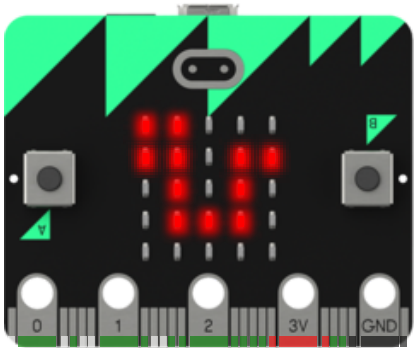
Name	Image
`Image.BUTTERFLY`	
`Image.GIRAFFE`	
`Image.SNAKE`	

1.2.5. Problem: Pick a pet!

Since you took such good care of your last pet, it's time to choose your own virtual pet!

Choose one of the pet pictures listed on the previous slide and write a program to display it.

For example, if you like Python, you could choose the pet snake:



You'll need

program.py

```
from microbit import *
```

Testing

- [ ] Testing that the display is showing one of the seven pets.
- [ ] **Well done, you've made your own virtual pet!**

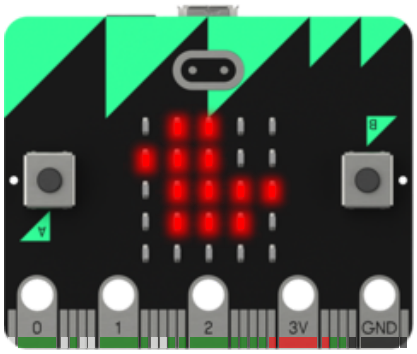
1.3. Animation

1.3.1. Ducks in a row

What if we want to show different images one after another?

Try running this:

```
from microbit import *

display.show(Image.GIRAFFE)
display.show(Image.DUCK)
```



We only see a picture of a duck!

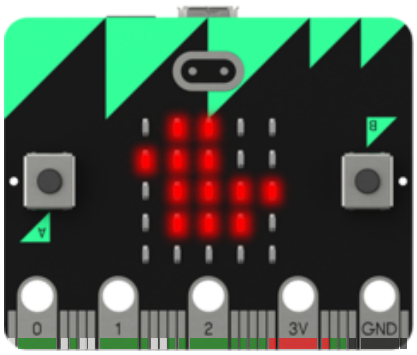
The code runs so fast that the giraffe doesn't stay on the display long enough for us to see it.

1.3.2. The `sleep` function

We can stop things going too fast using the **`sleep`** function:

```
from microbit import *

display.show(Image.GIRAFFE)
sleep(2000)
display.show(Image.DUCK)
```

Now it shows the giraffe, waits for 2 seconds, then shows the duck:



The `sleep` function makes the micro:bit wait for some time.

What is a function?

A *function* is a reusable piece of code that does a particular job.

1.3.3. Sleep for milliseconds

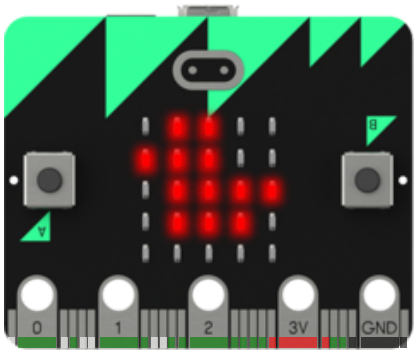
The `sleep` function needs information to do its job – it needs to know how long to wait for. We call this information an *argument*.

`sleep` expects the time in *milliseconds* (ms).

There are 1000 milliseconds in a second, so two seconds is 2 × 1000 = 2000 ms.

That's why when we *call* (or run) `sleep(2000)`, the giraffe stays on screen for two seconds:

```
from microbit import *

display.show(Image.GIRAFFE)
sleep(2000)
display.show(Image.DUCK)
```



1.3.4. To dot or not to dot

Why doesn't `sleep` have a dot like `display.show`?
Because `sleep` is a *function* that controls the whole micro:bit, while `display.show` is a *method* that controls the display.

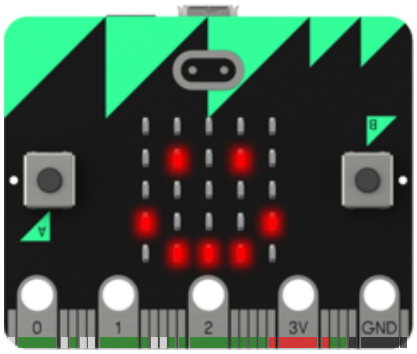
You call a *method* and a *function* the same way: by typing the name followed by round brackets. Any arguments go inside the brackets.

A method is just attached to something with a dot.

1.3.5. Problem: Pulling faces

Your micro:bit is feeling a bit meh today. Let's pull a silly face to make it feel happy again!

Write a program that shows a meh face for **one second**, followed by a silly face for **1.5 seconds**, before ending on a happy face.



Here are the built-in images for you to use:

Name	Image
`Image.MEH`	
`Image.SILLY`	
`Image.HAPPY`	

Sleep for milliseconds

Remember: the `sleep` function takes the time in milliseconds. You can convert seconds to milliseconds by multiplying by 1000.

You'll need

program.py

```
from microbit import *
```

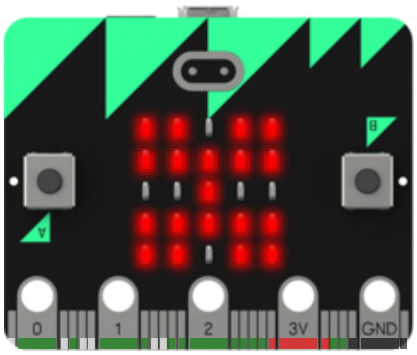
Testing

- [ ] Testing that the display starts with a meh face.
- [ ] Testing that the meh face is still on the screen less than 1 second later.
- [ ] Testing that the display changes to a silly face after 1 second.
- [ ] Testing that the silly face stays on the display for 1.5 seconds.
- [ ] Testing that the display changes to a happy face after 2.5 seconds.
- [ ] **Congratulations!!**

1.3.6. Problem: Virtual cocoon

Your virtual pet snake is jealous of all the virtual pet caterpillars! Help it transform into a butterfly like its fuzzy little friends.

Write a program that shows a snake for **1.5 seconds**, followed by a small diamond for **2 seconds**, followed by a large diamond for **0.5 seconds**, before ending on a butterfly.



Here are the built-in images for you to use:

Name	Image
`Image.SNAKE`	
`Image.DIAMOND_SMALL`	
`Image.DIAMOND`	
`Image.BUTTERFLY`	

Sleep for milliseconds

Remember: the `sleep` function takes the time in milliseconds. You can convert seconds to milliseconds by multiplying by 1000.

You'll need

program.py

```
from microbit import *
```

Testing

- [ ] Testing that the display starts with a snake.
- [ ] Testing that the snake is still on the screen less than 1.5 seconds later.
- [ ] Testing that the display changes to a small diamond after 1.5 seconds.
- [ ] Testing that the small diamond stays on the display for 2 seconds.
- [ ] Testing that the display changes to a large diamond after 2 seconds.
- [ ] Testing that the small diamond stays on the display for 2 seconds.

- [ ] Testing that the display changes to a butterfly after 0.5 seconds.
- [ ] **Congratulations! You turned the snake into a butterfly!**

1.4. Fixing errors

1.4.1. When things go wrong

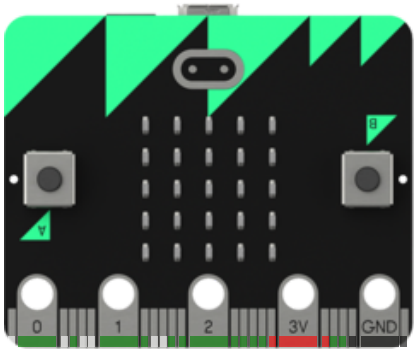
When you talk, you need to follow certain rules to be understood, called the *grammar* or *syntax* of a language.

Like English, Python has its own *syntax*. However, unlike people, computers can't understand bad grammar at all!

Run the following example with an image that doesn't exist. **Then click ■ to stop it running**:

```
from microbit import *

display.show(Image.EXCITED)
```

```
Traceback (most recent call last):
  File "__main__", line 3, in <module>
AttributeError: type object 'MicroBitImage' has no attribute 'EXCITED'
MicroPython v1.7-9-gbe020eb on 2016-09-14; micro:bit with nRF51822
Type "help()" for more information.
>>>
soft reboot
```



The BBC micro:bit scrolls the error on the display, which is hard to read. We'll also print the error message in another box for you.

Don't panic!

Errors happen all the time, but don't worry, you can learn to fix them. **Try to read and understand the error message**.

Errors on a real micro:bit

To see the printed error message for a real micro:bit, you'll need to use the serial console over USB (https://www.microbit.co.uk/td/serial-library).

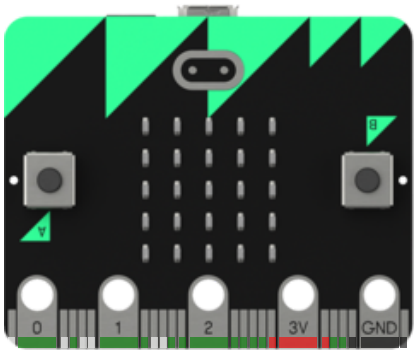
1.4.2. Help! I have a Syntax Error

A `SyntaxError` just means that you haven't followed the grammar or *syntax* of the programming language, so the computer can't understand you!

Let's practise fixing a `SyntaxError` together. Run this program:

```
from microbit import *

sleep(2000
display.show(Image.DUCK)
```

```
Traceback (most recent call last):
  File "__main__", line 4
SyntaxError: invalid syntax
MicroPython v1.7-9-gbe020eb on 2016-09-14; micro:bit with nRF51822
Type "help()" for more information.
>>>
soft reboot
Traceback (most recent call last):
  File "__main__", line 4
SyntaxError: invalid syntax
```



Where's the error?

The error message tries to help by printing which line it thinks the error is on. In this example, it says `line 4`.

If it doesn't seem like there's an error where the error message says, **try looking on the previous line**.

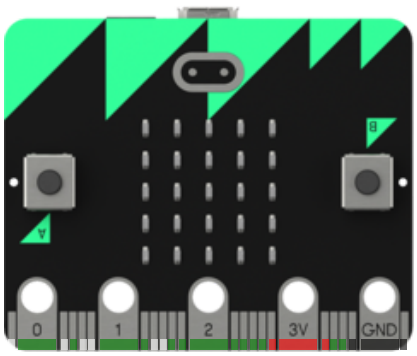
Hover me for answer

1.4.3. Scrolling letters and words

We can scroll our own message on the micro:bit display using the `display.scroll` method.

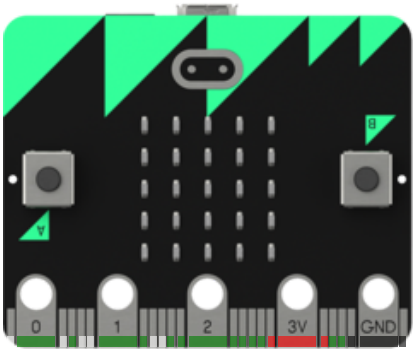
`display.scroll` takes a *string* as an argument.

```
from microbit import *

display.scroll('Hello')
```



A string can contain any letters, digits, punctuation and spaces that you want. We use the *single quote* to start and end the string.

```
from microbit import *

display.scroll('abc ABC 123 @!?.#')
```



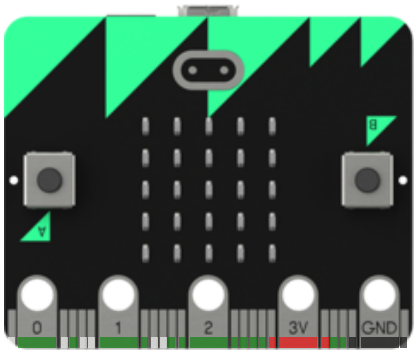
A string of characters

The individual letters, digits, symbols and spaces are called *characters* and the word string is short for *string of characters*.

1.4.4. Problem: I ❤ micro:bit

Let's put everything we've learned so far together. Write a program to animate the message: I ❤ micro:bit!

Your program should scroll `I`, then show the heart image `Image.HEART` for one second, then scroll `micro:bit!`



How to read the error message

If you get an error, don't panic. When solving problems, you can click the <> button on the left of your micro:bit to read the error message.

Pay attention to punctuation, and double check if the letter should be upper or lower case. Computers are very picky; even a single character can make a difference.

You'll need

program.py

```
from microbit import *
```

Testing

- [ ] Testing that an `I` scrolls past.
- [ ] Testing that a heart appears after the `I`.
- [ ] Testing that the heart stays on the display for 1 second.
- [ ] Testing that an `m` scrolls past.
- [ ] Testing that `micro:bit!` scrolls past.

1.4.5. More faces!

Here are a list of faces you can use to give your pet some personality:

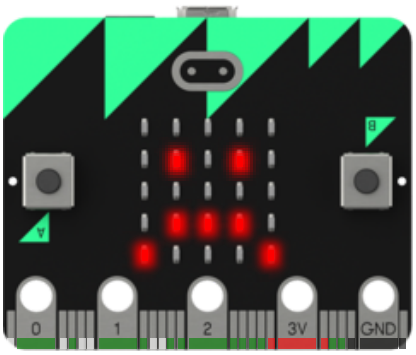
Name	Image
`Image.HAPPY`	
`Image.SAD`	
`Image.MEH`	
`Image.SILLY`	
`Image.ANGRY`	

Name	Image
`Image.FABULOUS`	
`Image.ASLEEP`	
`Image.CONFUSED`	
`Image.SURPRISED`	

1.4.6. Problem: My duck is sad

Now let's used what we've learned to give our pet some personality! Write a program to animate a message like: My 🦆 is ☹️

Your program should scroll `My`, then show an image of your pet (for example `Image.DUCK`) for **one second**, then scroll `is` then show a face (for example `Image.SAD`).



You can use *any* animal or face in the correct order.

You'll need

program.py

```
from microbit import *
```

Testing

- [ ] Testing that a `My` scrolls past.
- [ ] Testing that a pet appears after the `My`.
- [ ] Testing that the pet stays on the display for 1 second.
- [ ] Testing that `is` scrolls past.
- [ ] Testing that a face appears after the `is`.
- [ ] Testing that the face stays on the display.

1.5. Summary

1.5.1. Congratulations!

You've reached the end of Module 1.

We learned about:

- what's in the BBC micro:bit
- displaying images on the micro:bit
- built-in images
- making the micro:bit wait with `sleep`
- the difference between methods and functions
- syntax errors and how to fix them
- scrolling text on the display

Click » to continue on to Module 2: Making decisions with buttons and gestures.

2

BUTTONS AND GESTURES

2.1. Looping forever

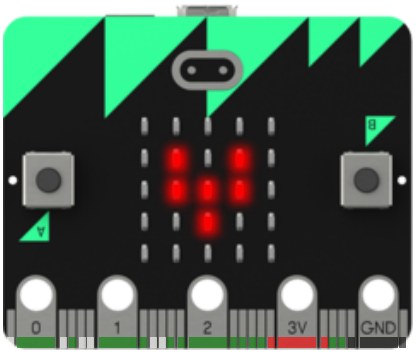
2.1.1. Introducing loops

So far, our programs make the micro:bit do something and then stop. What if we want the program to keep running forever?

We can use a `while` *loop* to run some code again (and again)!

For example, this heartbeat will keep running until you stop it:

```
from microbit import *

while True:
  display.show(Image.HEART)
  sleep(500)
  display.show(Image.HEART_SMALL)
  sleep(500)
```



You have to click stop!

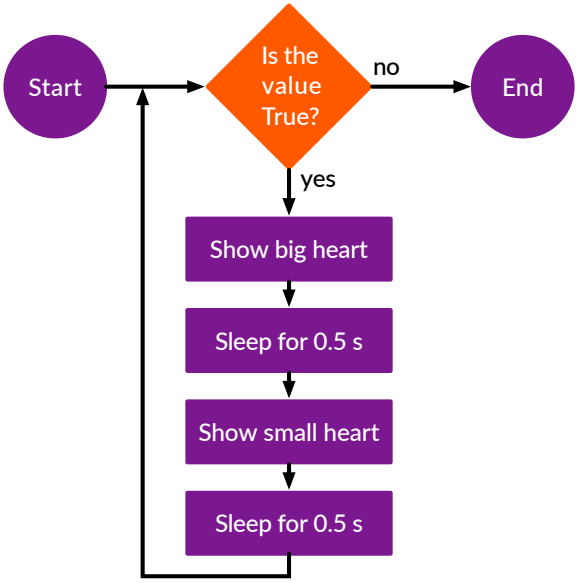
Don't forget to use the ■ button to stop the program, otherwise it will keep running forever.

2.1.2. Visualising a loop

Let's look at the heartbeat program as a *flowchart*.

Follow the arrows to see how the program runs. The loop only finishes when the answer to “is the value `True`?” is no.

To make the loop run forever, we give it the value `True`, so it's stuck going around the "yes" part of the flowchart. This kind of loop is called an *infinite loop*.



2.1.3. Writing a `while` loop

A `while` loop keeps repeating *while* the *condition* is `True`. To create an *infinite loop*, we give a `while` loop the condition `True` by typing:

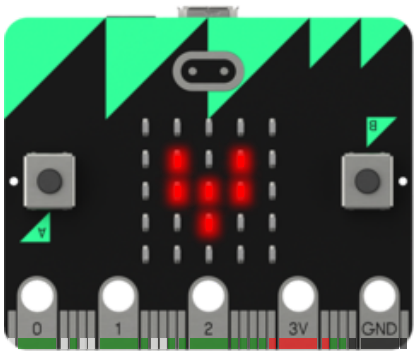
```
while True:
```

Don't forget the colon!

Put the code to repeat inside the loop by **indenting** it with spaces:

```
from microbit import *

while True:
  display.show(Image.HEART)
  sleep(500)
  display.show(Image.HEART_SMALL)
  sleep(500)
```

The last 4 statements are indented, so they are all repeated:



Indentation

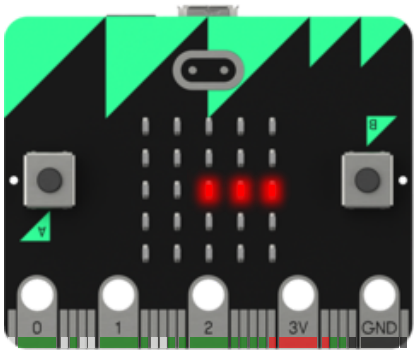
We *indent* by putting spaces at the beginning of the line. Indentation tells Python that the code is **inside the loop**.

Make sure you use the same number of spaces for indenting each line inside the loop.

2.1.4. Problem: Tick tock

Time is ticking! Write a program to move a clock hand continuously around the display.

It should start at 12 o'clock, go to 3 o'clock, then 6 o'clock, and then 9 o'clock, **staying in each position for one second**.



Here are the images for you to use:

Name	Image
`Image.CLOCK12`	
`Image.CLOCK3`	
`Image.CLOCK6`	
`Image.CLOCK9`	

You have to click stop!

When you run your program, you'll need to click the ■ button to stop it running before you'll be able to submit.

You'll need

program.py

```
from microbit import *
```

Testing

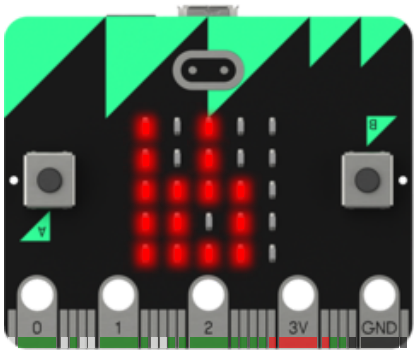
- [ ] Testing that the display starts with 12 o'clock.
- [ ] Testing that the display is still showing 12 o'clock after less than a second.
- [ ] Testing that the display shows 3 o'clock for a second.
- [ ] Testing that the display shows 6 o'clock for a second.
- [ ] Testing that the display shows 9 o'clock for a second.
- [ ] Checking that your code contains an infinite loop.
- [ ] Testing that the display goes back to 12 o'clock for a second.

- [ ] Testing that the animation loops continuously.

2.1.5. Problem: Pet shop

How much is that turtle in the window? Or that Giraffe? Show me all of your pets!

Make a program to show each of the animals on the micro:bit for **1 second each**. And then repeat forever!



Here are the pet images for you to use. **Make sure you show them in order!**

Name	Image
`Image.RABBIT`	
`Image.COW`	
`Image.DUCK`	
`Image.TORTOISE`	
`Image.BUTTERFLY`	
`Image.GIRAFFE`	
`Image.SNAKE`	

You'll need

program.py

```
from microbit import *
```

Testing

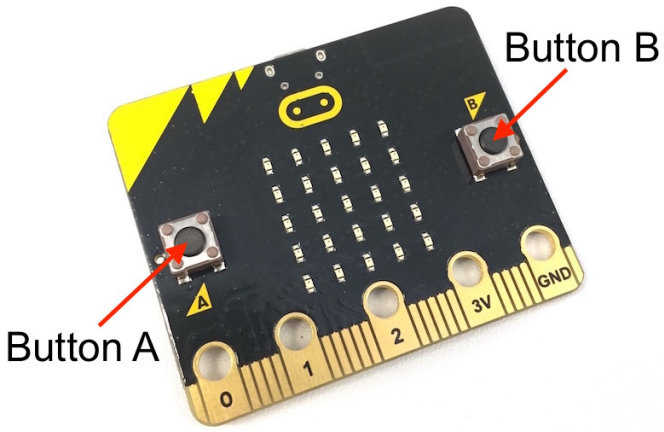
- [ ] Testing that the display starts with a rabbit.
- [ ] Testing that the display is still showing a rabbit after less than a second.
- [ ] Testing that the display shows a cow for a second.
- [ ] Testing that the display shows a duck for a second.
- [ ] Testing that the display shows a tortoise for a second.

- [ ] Testing that the display shows a butterfly for a second.
- [ ] Testing that the display shows a giraffe for a second.
- [ ] Testing that the display shows a snake for a second.
- [ ] Checking that your code contains an infinite loop.
- [ ] Testing that the display goes back to a rabbit for a second.
- [ ] Testing that the animation loops continuously.
- [ ] **Well done! You can loop forever!**

2.2. Making decisions with buttons

2.2.1. Button A and Button B

The BBC micro:bit has two buttons, labelled A and B.



The `microbit` module represents these buttons as two objects: `button_a` and `button_b`.

These are just like the `display` object you've been using, but they have their own *methods*. For example, we can call the `is_pressed` method:

- `button_a.is_pressed()` to check if Button A is being held down
- `button_b.is_pressed()` to check if Button B is being held down

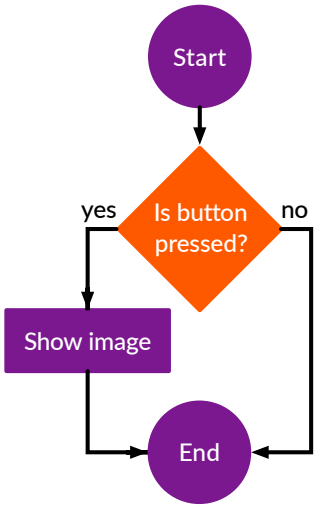
> **Methods are attached**
>
> We need to attach the `is_pressed` method to `button_a` or `button_b`, otherwise we wouldn't know which button to check.

2.2.2. Making decisions

So far our programs have changed the micro:bit display (produced *output*). These programs have run the same way every time.

We want our programs to react to things in the world (respond to *input*), like someone pressing a micro:bit button.

This flowchart describes a process (or *algorithm*) that makes the program run differently if a button is pressed:

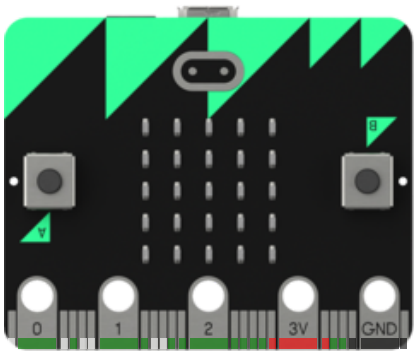


The diamond requires a **yes** or **no** decision. The answer determines which line we follow. If the answer is **yes**, we do the extra step of showing an image. If the answer is no **no**, we skip it.

2.2.3. Writing an `if` statement

We can write an `if` statement to make the decision in the orange diamond of the flowchart.

```
from microbit import *

if button_a.is_pressed():
  display.show(Image.DUCK)
```



Try running this program and then pressing Button A.

Why doesn't it work? Because the code runs too fast — the program ends before we can press the button!

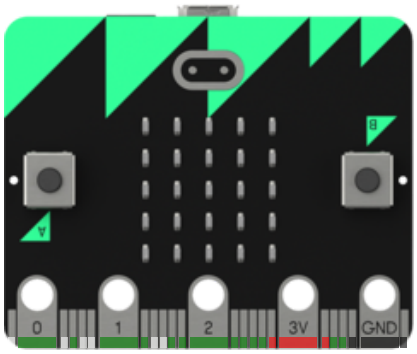
We need to keep checking whether the button is pressed...

> **Use your mouse or keyboard**
>
> Press the buttons in the examples by clicking with your mouse or pressing A or B on your keyboard.

2.2.4. Decisions inside a loop

We fix this by putting the `if` statement *inside* an infinite loop:

```
from microbit import *

while True:
  if button_a.is_pressed():
    display.show(Image.DUCK)
```



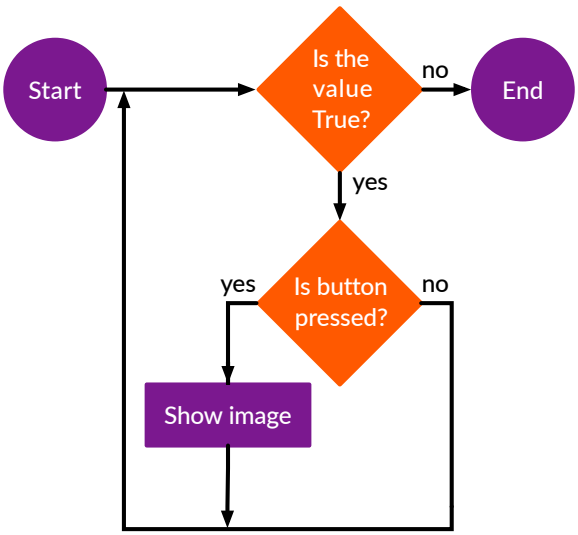
The `if` statement only runs `display.show(Image.DUCK)` *if* Button A is being pressed.

Because we don't clear the display, the image will stay after the first time we press the button.

> **Two levels of indentation**
>
> We put code *inside* the `if` statement by *indenting* it 2 spaces.
>
> We put the whole `if` statement (including the code inside it) inside the `while` loop by *indenting another level* (4 spaces).

2.2.5. Visualising control structures

Here's the program as a flowchart. Follow it and see how it will keep looping, whether or not the button is pressed. The decision is *inside* the loop.



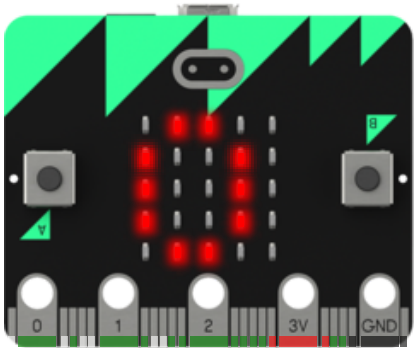
Both `while` and `if` are called *control structures*, because they *control* the flow of the program. Our flowcharts show them as orange diamonds where you go one of two ways based on the answer to a question.

2.2.6. Problem: 3... 2... 1... GO!

Make a count down timer for starting races (quitely). On your marks, get set, GO!

Write a program that will scroll `3 2 1 GO!` across the display when the A button is pressed.

Here's an example interaction with the program (you can't actually press the buttons yourself).



You'll need

program.py

```
from microbit import *
```

Testing

- [ ] Checking that your code contains an infinite loop.
- [ ] Testing that the display starts off being blank.
- [ ] Testing that the display counts down when the A button is pressed.
- [ ] Testing that it went back to a blank screen afterwards.
- [ ] Testing that it continues to work multiple times.

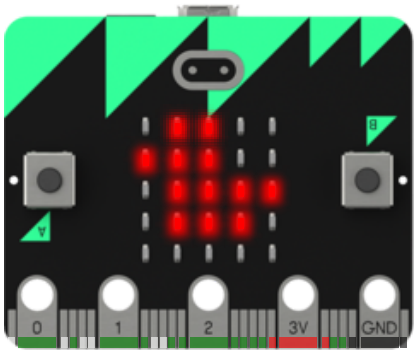
2.2.7. Problem: Duck pet vs Rabbit pet

Is it a duck pet or a rabbit pet?

Write a program that if the A button is pressed, shows a duck (`Image.DUCK`). And if the B button is pressed, shows a rabbit (`Image.RABBIT`).

You'll have to use two `if` statements for this one!

Here's a demo:



You'll need

program.py

```
from microbit import *
```

Testing

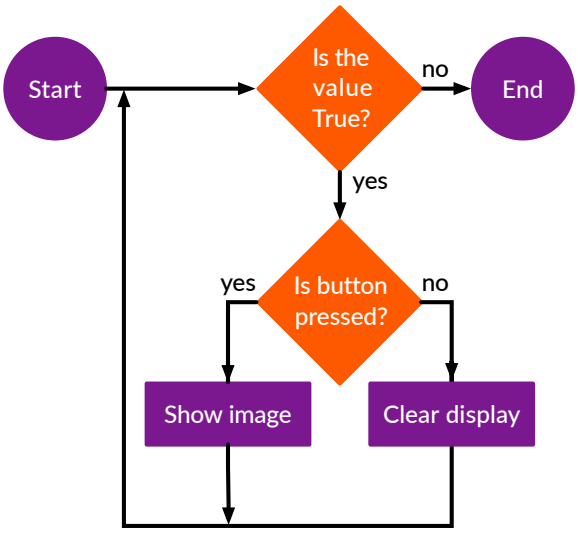
- [ ] Checking that your code contains an infinite loop.
- [ ] Testing that the display starts off blank.
- [ ] Testing that it shows a duck when the A button is pressed.
- [ ] Testing that it shows a rabbit when the B button is pressed.
- [ ] Testing that it shows a duck then a rabbit when the A button then the B button is pressed.
- [ ] Testing that it continues to work multiple times.

2.3. Decisions with two options

2.3.1. Decisions with two options

Often when we make a decision, we care about both answers.

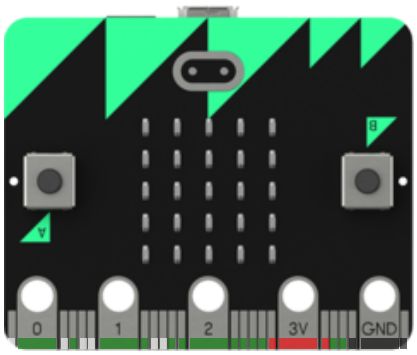
When we ask "Is the button pressed?", we might want to show an image if the answer is yes, and clear the display if the answer is no.



2.3.2. Writing an `if-else` statement

We handle decisions with two options by adding an `else` clause to our `if` statement.

```
from microbit import *

while True:
  if button_a.is_pressed():
    display.show(Image.DUCK)
  else:
    display.clear()
```



Indentation strikes again!

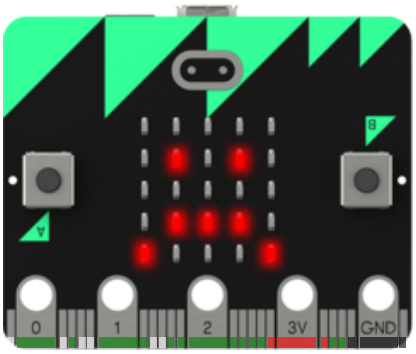
Just like with the `if` statement, we *indent* the code we want to put *inside* the `else` clause.

`if` and `else` keywords are on the **same level** of indentation, because they're two outcomes of the **same decision**.

2.3.3. Problem: Smile for the camera!

Smile for the camera! Write a program that shows a happy face (`Image.HAPPY`) while Button A is pressed (and the photo is being taken), and a sad face (`Image.SAD`) while the button is released.

Here's an example interaction with the program (you can't actually press the buttons yourself).



You'll need

program.py

```
from microbit import *
```

Testing

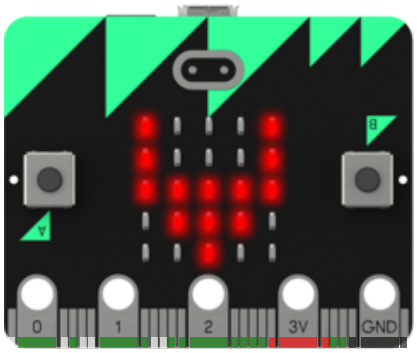
- [ ] Checking that your code contains an infinite loop.
- [ ] Testing that the display starts off showing a sad face.
- [ ] Testing that it becomes happy when the button is pressed.
- [ ] Testing that it went back to a sad face after the button was released.
- [ ] Testing that holding down the button keeps the happy face on the screen.
- [ ] Testing that it continues to work multiple times.

2.3.4. Problem: Feed me!

Write a program that shows an open mouth (using `Image.SURPRISED`) if you "feed" the pet with the A button.

You can use *any* animal stored in the micro:bit, for example the cow (`Image.COW`), but it will also work on other animals.

Here's an example interaction with the program (you can't actually press the buttons yourself).



You'll need

program.py

```
from microbit import *
```

Testing

- [ ] Checking that your code contains an infinite loop.
- [ ] Testing that the display starts off showing a pet.
- [ ] Testing that it opens its mouth when the button is pressed.
- [ ] Testing that it went back to a pet after the button was released.
- [ ] Testing that holding down the button keeps the mouth open on the screen.
- [ ] Testing that it continues to work multiple times.
- [ ] **Nice work, you fed the pet!**

2.4. Making decisions with gestures

2.4.1. Accelerometer

The micro:bit has a built-in accelerometer (https://en.wikipedia.org/wiki/Accelerometer) that measures *acceleration*.

Lots of other devices you use contain accelerometers, including your phone, fitness tracker, and some game controllers.

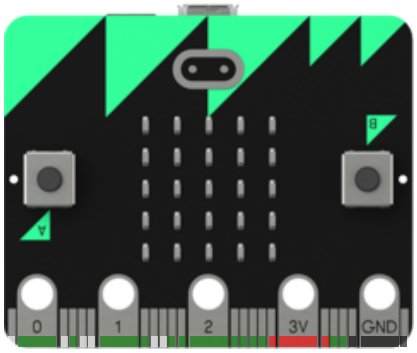
Using an accelerometer, you can detect which way the device is facing (e.g. screen orientation on your phone). You can also detect movement where acceleration changes (such as vibration, shock, and falls).

2.4.2. Shake gesture

The `microbit` module represents the accelerometer as the object `accelerometer`. We can detect gestures made with your micro:bit using the `was_gesture` method.

Run this program. Press the "Shake" button to shake the simulated micro:bit:

```
from microbit import *

while True:
  if accelerometer.was_gesture("shake"):
    display.show(Image.ANGRY)
    sleep(500)
    display.clear()
```



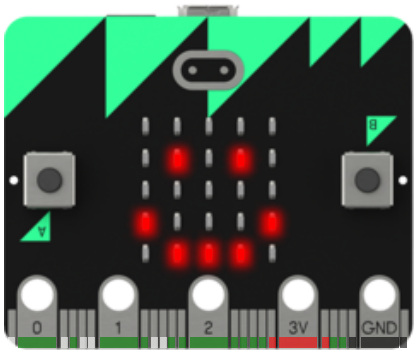
Shake

2.4.3. Problem: Shake it off

Haters gonna hate! Shake off those angry feelings!

Show an angry face (`Image.ANGRY`) if nothing is happening. But if the the micro:bit detects a shake gesture then show a happy face (`Image.HAPPY`) for 2 seconds.

Here's an example interaction:




```
Shake
```

Shake it!

Like detecting a button press with `button_a.is_pressed()`, you can detect a shake of the micro:bit by using `accelerometer.was_gesture("shake")` inside of an `if` statement

You'll need

program.py

```
from microbit import *
```

Testing

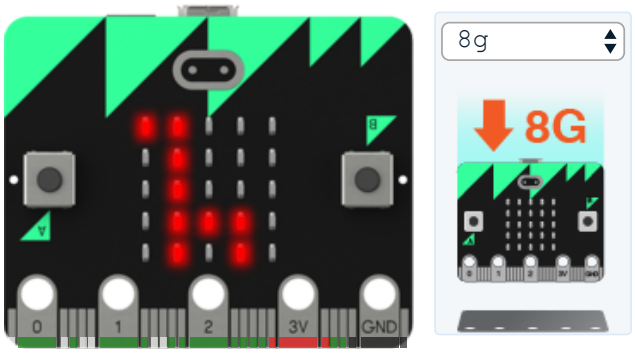
- [ ] Testing that the display starts with an angry face.
- [ ] Testing that a happy face appears after shaking.
- [ ] Testing that an angry face appears after two seconds shaking.

2.4.4. Problem: Giraffe sleeps standing

In order to avoid predators, griaffes sleep standing up. Your pet giraffe doesn't need to worry about predators, because she sleeps standing!

Write a program that shows a giraffe (`Image.GIRAFFE`) normally, you make the `up` gesture shows (`Image.ASLEEP`) for 3 seconds.

Here's an example interaction:



You'll need

program.py

```
from microbit import *
```

Testing

- [ ] Testing that the display starts with a giraffe.
- [ ] Testing that the image is the giraffe after an `up` gesture.
- [ ] Testing that giraffe appears again after 3 seconds.
- [ ] Testing that it works repeatedly.

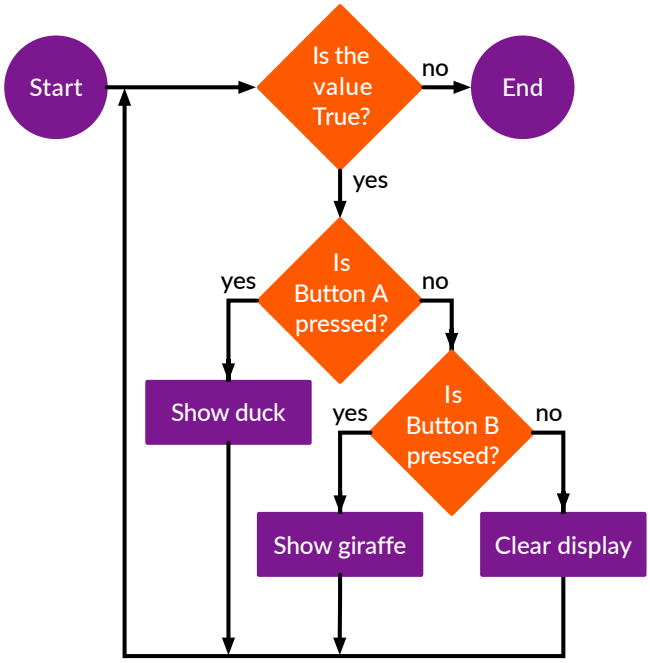
2.5. More complex decisions

2.5.1. Decisions with multiple options

Decisions with multiple options need to ask more than one question. This flowchart covers three options:

1. Button A is pressed
2. Button B is pressed
3. Neither button is pressed

If the answer to "Is Button A pressed?" is no, we need to ask another question, "Is Button B pressed?"

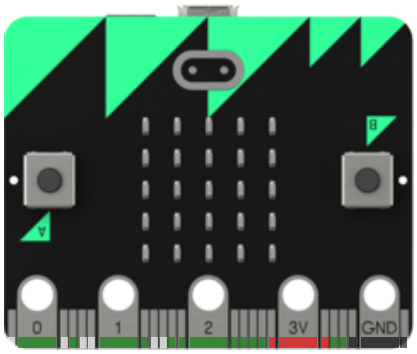


2.5.2. Writing an `if-elif-else` statement

We can add an `elif` (abbreviation of *else if*) clause to make the extra decision in the flowchart.

`elif` clauses go after the `if` statement and before the `else` clause, at the same level of indentation.

```
from microbit import *

while True:
 if button_a.is_pressed():
   display.show(Image.DUCK)
 elif button_b.is_pressed():
   display.show(Image.GIRAFFE)
 else:
   display.clear()
```



Order matters!

What do you expect to happen when you press both buttons at the same time? **Have a guess before you try it out.**

Python checks each statement from top to bottom and runs only the first statement where the answer is yes.

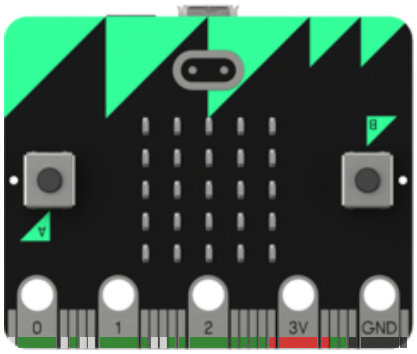
This program checks whether Button A is pressed before it checks Button B, so it shows a duck when both are pressed!

2.5.3. What if both buttons are pressed at once?

We can make a single decision based on multiple questions by using the `and` operator to join them.

This program only shows an image when *both* buttons are pressed at once. **Use the `A` and `B` keys on your keyboard to press both buttons at the same time.**

```
from microbit import *

while True:
 if button_a.is_pressed() and button_b.is_pressed():
   display.show(Image.BUTTERFLY)
 else:
   display.clear()
```

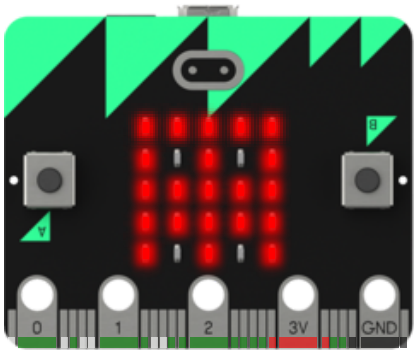


2.5.4. Even more options

We can add multiple `elif` statements to cover even more options. The example below shows a different image for:

- both buttons pressed
- only Button A pressed
- only Button B pressed
- neither button pressed

```
from microbit import *

while True:
  if button_a.is_pressed() and button_b.is_pressed():
    display.show(Image.BUTTERFLY)
  elif button_a.is_pressed():
    display.show(Image.DUCK)
  elif button_b.is_pressed():
    display.show(Image.GIRAFFE)
  else:
    display.show(Image.GHOST)
```



Order matters even more!

What happens if you swap the order of the statements so you check whether both buttons are pressed after checking Button A by itself?

The program would show a duck even when both buttons are pressed.

Remember, Python checks each statement from top to bottom and runs only the first statement where the answer is yes.

2.5.5. Problem: ATV controller

Let's build a controller for an all-terrain vehicle (ATV) that uses caterpillar treads (https://en.wikipedia.org/wiki/Continuous_track) instead of wheels.

Bulldozer with caterpillar treads.

When both buttons are pressed, it drives forwards. If Button A is pressed, it turns left. If Button B is pressed, it turns right. Otherwise, the vehicle doesn't move.

Write a program to draw an arrow on the display indicating the direction. The display should be blank when not moving.

Here are the arrow images:

Name	Direction	Image
`Image.ARROW_N`	forward	
`Image.ARROW_E`	right	
`Image.ARROW_W`	left	

Remember!

Use the `A` and `B` keys on your keyboard to press both buttons at the same time.

You'll need

program.py

```
from microbit import *
```

Testing

- [ ] Checking that your code contains an infinite loop.
- [ ] Testing that the display starts blank.
- [ ] Testing that the display shows the left arrow when the A button is pressed.
- [ ] Testing that the display goes blank again when A button is released.

- [ ] Testing that the display shows the right arrow when the B button is pressed.
- [ ] Testing that the display goes blank again when the B button is released.
- [ ] Testing that the display shows the up arrow when both buttons are pressed.
- [ ] Testing that the display goes blank again when the buttons are released.
- [ ] Testing none → A → A+B → B → none.

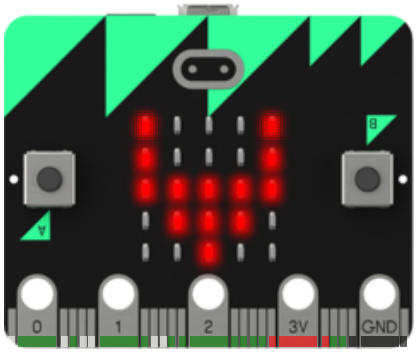
2.5.6. Problem: Feed me or pet me!

Let's add more interactions with our pet!

Write a program that shows an open mouth (using `Image.SURPRISED`) if you "feed" the pet with the A button. Shows a smile face (with `Image.HAPPY`) if you "pat" the pet with the B button. And if you try to fed and pat your pet at the same time it will get angry (and show a `Image.ANGRY`) face!

Otherwise your program should just show a picture of your pet.

Here's an example interaction with the program, if your pet was a cow (`Image.COW`):



You'll need

program.py

```
from microbit import *
```

Testing

- [ ] Checking that your code contains an infinite loop.
- [ ] Testing that the display starts off showing a pet.
- [ ] Testing that the display shows an open mouth when the A button is pressed.
- [ ] Testing that the display goes back to the pet again when A button is released.
- [ ] Testing that the display shows a happy face when the B button is pressed.
- [ ] Testing that the display goes back to the pet again when B button is released.
- [ ] Testing that the display shows an angry face when both buttons are pressed.
- [ ] Testing that the display goes back to the pet again when the buttons are released.
- [ ] Testing none → A → A+B → B → none.
- [ ] **Well done! You can pet, feed and make your pet angry! Like a real one!**

2.6. Summary

2.6.1. Congratulations!

Fantastic work! You've just finished Module 2.

We learned about:

- visualising programs as flowcharts
- infinite `while` loops
- buttons on the micro:bit
- the accelerometer and gestures on the micro:bit
- the difference between input and output
- making simple decisions with `if` statements
- making decisions with two options with `if-else` statements
- making decisions with multiple options `if-elif-else` statements
- making complex decisions with the `and` operator

VIRTUAL PET EXTENSIONS

3.1. More micro:bit

3.1.1. More micro:bit

You've already learned the main parts of programming a micro:bit.

- Output (the micro:bit display)
- Input (buttons and gestures)
- `if`/`elif`/`else` statements
- `while` loops

With these tools you can already build lots and lots of interesting projects!

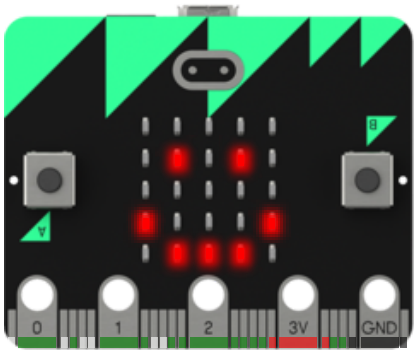
But if you'd like to learn about more things you can do then you can try out this module for a quick sample of some extra cool micro:bit features!

3.2. DIY images

3.2.1. Image strings

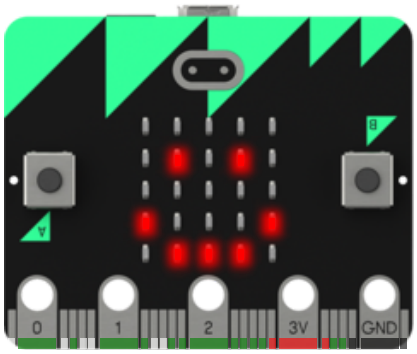
So far, we've only used the built-in images from the `Image` class:

```
from microbit import *

display.show(Image.HAPPY)
```



Images can also described by *strings*. Here's a do it yourself `HAPPY`:

```
from microbit import *

SMILE = Image('00000:'
              '09090:'
              '00000:'
              '90009:'
              '09990:')
display.show(SMILE)
```



`SMILE = Image()` creates an image and stores it in the constant variable `SMILE`. It can then be used like any builtin image.

The strings represent the brightness of each pixel in the image:

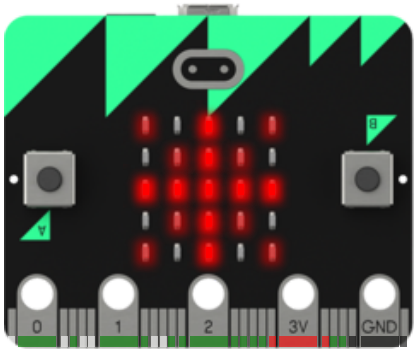
- Each line is a row of the image and ends with a colon (`:`).
- Each number is the pixel brightness from `0` (off) to `9` (fully on).

If create an image without passing a string, e.g. `SMILE = Image()`, it creates a blank image, with all off the pixels off.

3.2.2. DIY images

We can now create our own images (with varying brightness):

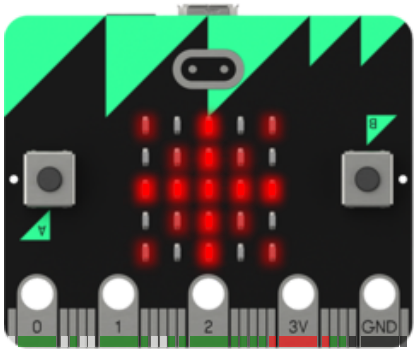
```
from microbit import *

FLAG = Image('50905:'
             '05950:'
             '99999:'
             '05950:'
             '50905:')
display.show(FLAG)
```



Try making your own image!

We split the 5x5 image over five lines to make it easier to read, but you can combine them into one string. The colon separates each row, so you can also leave the last one off:

```
from microbit import *

FLAG = Image('50905:05950:99999:05950:50905')
display.show(FLAG)
```



3.2.3. Problem: I choose you!

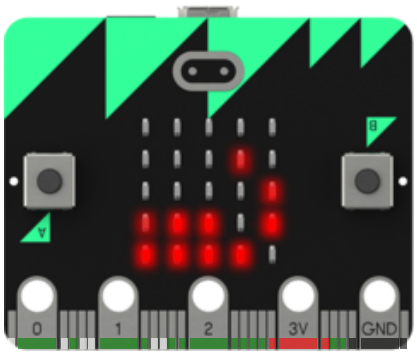
Nintendo's Pokemon games often start off with a choice between 3 different starting Pokemon.

Let's create three new pets to choose from for our virtual pet game! **You must create your own pets**, they can be *anything* but they must all be different.

Make a program that shows a different custom made pet when the A+B buttons, A button, or B button is pressed.

> **How do I use buttons again?**
>
> You can go back to the previous module if you've forgotten about how buttons and `if`/`elif`/`else` statments work.

Here's an example where you can choose between a mouse, cat or elephant:



If you wanted an example of a pet, the code for the mouse would look like this:

```
MOUSE = Image('00000:'
              '00060:'
              '00007:'
              '69909:'
              '99990')
```

You'll need

program.py

```
from microbit import *

PET1 = Image('00000:'
             '00000:'
             '00000:'
             '00000:'
             '00000')
PET2 = Image('00000:'
             '00000:'
             '00000:'
             '00000:'
             '00000')
PET3 = Image('00000:'
             '00000:'
             '00000:'
             '00000:'
             '00000')
```

Testing

- [ ] Checking that your code contains an infinite loop.
- [ ] Testing that the display starts blank.
- [ ] Testing that the display shows a custom pet when the A button is pressed.
- [ ] Testing that the display shows a custom pet when the B button is pressed.
- [ ] Testing that the display shows a custom pet when the A and B buttons are pressed.
- [ ] Checking that all three of the pets have different images.
- [ ] Testing none → A → A+B → B.

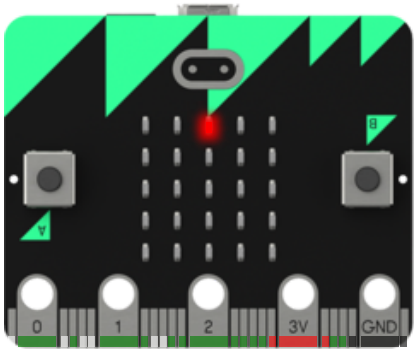
3.3. Shifting images

3.3.1. Shifting images

You can move images around on the screen by shifting them up, down, left and right.

Let's start with an image that's a single pixel in the middle of the screen and move it up to the top:

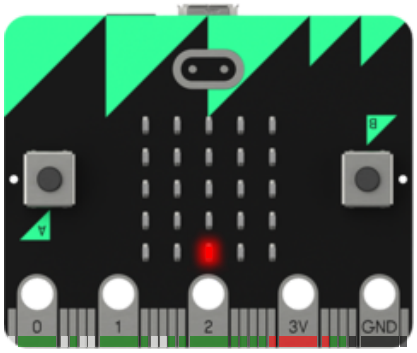
```
from microbit import *

START = Image('00000:00000:00900:00000:00000')
display.show(START)
sleep(1000)
display.show(START.shift_up(2))
```



`shift_up` takes the original image and returns a new one that's been shifted *up* by the number of rows given as the argument.

`shift_down`, `shift_left` and `shift_right` work the same way:

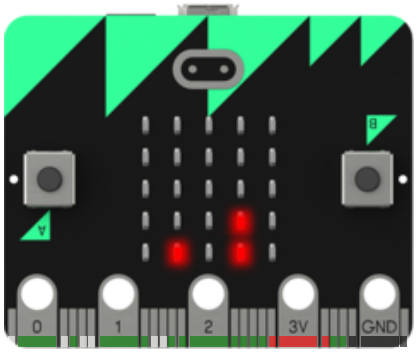
```
from microbit import *

START = Image('00000:00000:00900:00000:00000')
display.show(START)
sleep(1000)
display.show(START.shift_left(2))
sleep(1000)
display.show(START.shift_right(2))
sleep(1000)
display.show(START.shift_down(2))
```



3.3.2. Shifting with a `for` loop

We can use a `for` loop to shift images one row/column at a time, to create an animation! For example, to animate a ship sinking:

```
from microbit import *

SHIP = Image('00090:'
             '09090:'
             '09090:'
             '99999:'
             '09990:')

for i in range(6):
  display.show(SHIP.shift_down(i))
  sleep(500)
```



Remember, `range(6)` counts from `0` to `5`.

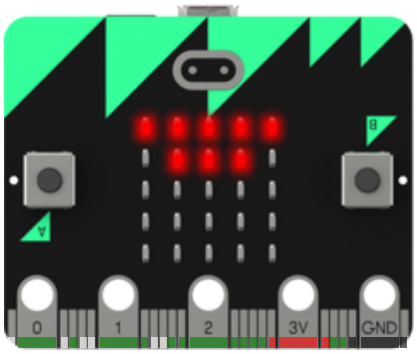
Using a `for` loop, we first show the original image shifted down by 0 pixels, then 1 pixel, then 2 pixels, up to 5 pixels, where the ship is no longer on the display.

3.3.3. Unshifting with a `for` loop

Instead of shifting an image *from* its original position downwards, how do we shift an image upwards *to* its original position?

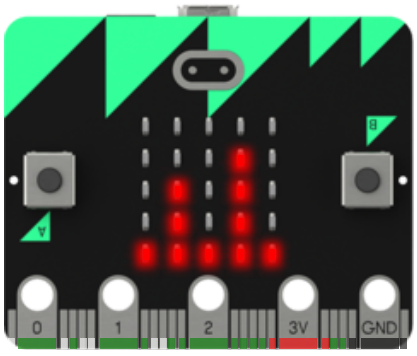
This **doesn't** work the way we would like it to because it just moves the boat up off the display:

```
from microbit import *

SHIP = Image('00090:'
             '09090:'
             '09090:'
             '99999:'
             '09990:')

for i in range(6):
  display.show(SHIP.shift_up(i))
  sleep(500)
```



This code **does** work the way we want it to because it starts shifting down a large amount and finishes shifting down 0 places

```
from microbit import *

SHIP = Image('00090:'
             '09090:'
             '09090:'
             '99999:'
             '09990:')

for i in reversed(range(6)):
  display.show(SHIP.shift_down(i))
  sleep(500)
```



3.3.4. Problem: Jump!

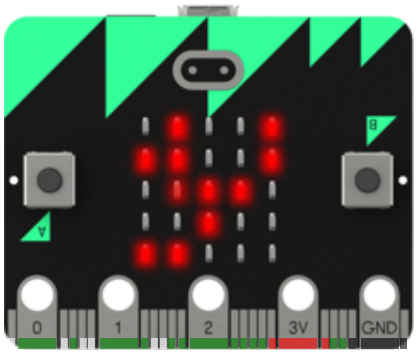
Sometimes smiley faces are not enough to show how happy you are. So you might as well jump!

Write a program that first shows the kangaroo image we've provided. And then makes the kangaroo jump when the A button is pressed.

To make the kangaroo jump:

- `shift_up` until it disappears off the display, with a pause of **50ms** between each shift.
- And then do the reverse until the kangaroo is back where it started. See the example of `reversed` on the previous slide.

Here's an example of how it should work:



You'll need

program.py

```
from microbit import *

KANGAROO = Image('09009:'
                 '99009:'
                 '06990:'
                 '00900:'
                 '99000:')

display.show(KANGAROO)
```

Testing

- [ ] Testing that you showed the kangaroo.
- [ ] Testing that your display shows the first frame of the jump.
- [ ] Testing that the first frame stays on the display for 50ms.
- [ ] Testing that the kangaroo jumps off the display.
- [ ] Testing that the kangaroo falls back down.
- [ ] Testing that the kangaroo stays on the display after jumping.
- [ ] Testing that the kangaroo can jump two times in a row.
- [ ] **Congrats! You can bounce better than the rest of them!**

3.4. Music

3.4.1. Connecting some headphones

The micro:bit has lots of built-in components like the LEDs, but it can do even more when we connect it to other components.

Here we'll learn how to connect headphones to play music!

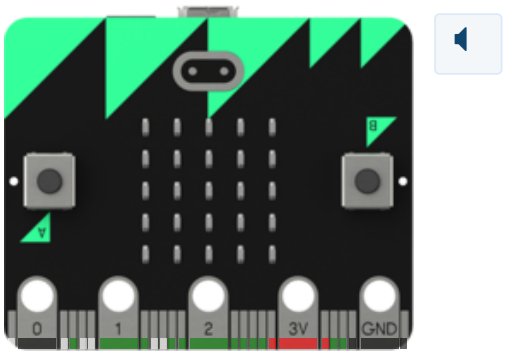
When you see this button next to the micro:bit in our simulator, it means that the headphones (or speaker) are connected and playing.

> **Hack your headphones!**
>
> If you have a real micro:bit, follow these instructions (https://www.microbit.co.uk/blocks/lessons/hack-your-headphones/activity) to play sound through your headphones!

3.4.2. Playing music

Let's play a song with the micro:bit. Run this example (remember to turn your system sound on!):

```
from microbit import *
import music

music.play(music.BIRTHDAY)
```



3.4.3. The `music` module

In order to play music, we need to import the **`music`** module by adding the line:

```
import music
```

The **`music`** module gives us:

- **`music.play`** to play sound;
- built-in tunes like **`music.BIRTHDAY`** which we can pass as an argument to **`music.play`**.

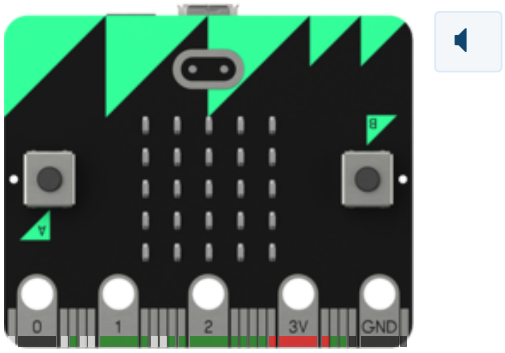
We'll provide the `import` **`music`** statement for problems that need it, but don't delete it!

> **Two separate imports!**
>
> Unfortunately, **`music`** is not part of the **`microbit`** module, and so it needs to be imported in a different way.

3.4.4. Play a whole album

Here's the same program, but with different music:

```
from microbit import *
import music

music.play(music.ODE)
```



The `music` module provides lots of other built-in tunes you can use! **Try them out by editing the example above.**

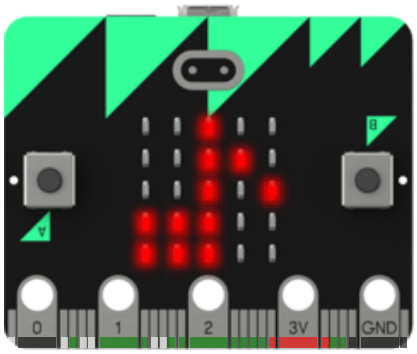
- `music.DADADADUM`
- `music.ENTERTAINER`
- `music.PRELUDE`
- `music.ODE`
- `music.NYAN`
- `music.RINGTONE`
- `music.FUNK`
- `music.BLUES`
- `music.BIRTHDAY`
- `music.WEDDING`
- `music.FUNERAL`
- `music.PUNCHLINE`
- `music.PYTHON`
- `music.BADDY`
- `music.CHASE`
- `music.BA_DING`
- `music.WAWAWAWAA`
- `music.JUMP_UP`
- `music.JUMP_DOWN`
- `music.POWER_UP`
- `music.POWER_DOWN`

3.4.5. Music takes time

Playing music makes the rest of the program wait. It's a bit like sleeping for the length of the song (except that music is playing!)

In this example, the note (a quaver) won't show on the screen until after the whole song has played:

```
from microbit import *
import music

music.play(music.ENTERTAINER)
display.show(Image.MUSIC_QUAVER)
```

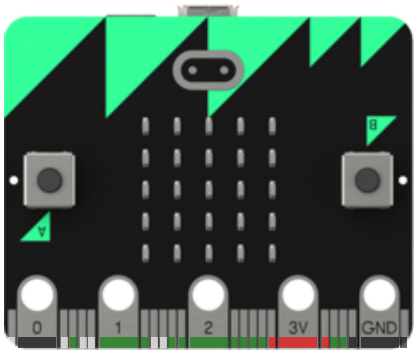


Put the `display.show` before the `music.play`, and run it again.

3.4.6. Problem: Power up, power down

Write a program to interact with your virtual pet using sound.

The program should play `music.POWER_UP`, then show a happy face (`Image.HAPPY`) for two seconds, before clearing the display with `display.clear()` and playing `music.POWER_DOWN`:



You'll need

program.py

```
from microbit import *
import music
```

Testing

- [ ] Testing that the micro:bit starts playing music.
- [ ] Testing that it plays `music.POWER_UP`.
- [ ] Testing that it then shows the happy face.
- [ ] Testing that it shows the happy face for two seconds.
- [ ] Testing that the display then goes blank.
- [ ] Testing that it plays `music.POWER_DOWN` at the end.

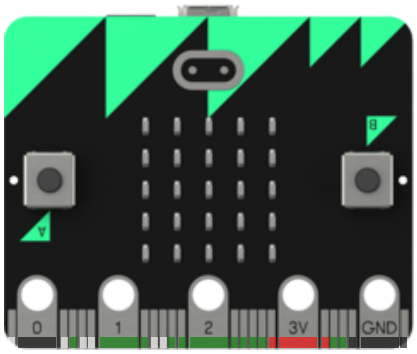
3.4.7. Problem: Sad trombone

A common sound effect is a Sad Trombone (https://wompwompwomp.com/), which might be used when someone almost scores a goal but misses. Or perhaps it could be played when you forget to feed a virtual pet!

Write a program make your micro:bit play `music.WAWAWAWAA` (also known as "Sad Trombone") when Button A is pressed.

Your program should also display the sad face (`Image.SAD`) while the sound is playing, then clear the display afterwards.

Here's an example interaction:



You'll need

program.py

```
from microbit import *
import music
```

Testing

- [ ] Checking that your code contains an infinite loop.
- [ ] Testing that the micro:bit is initially not playing any sound.
- [ ] Testing that pressing button A starts playing music.
- [ ] Testing that pressing button A starts playing `music.WAWAWAWAA`.
- [ ] Testing that it shows the sad face.
- [ ] Testing that the display clears after the sound finishes.
- [ ] Testing that it works repeatedly.

3.5. Random

3.5.1. Importing `random`

Playing Rock–paper–scissors (https://en.wikipedia.org/wiki/Rock%E2%80%93paper%E2%80%93scissors) against a program that always does the same thing would get boring very quickly.

Often when we create games, adding randomness makes them a lot more fun!

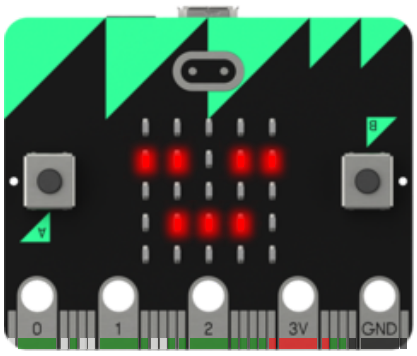
We can do that with the Python **`random`** module. Because it's a separate module (not just for the micro:bit), we need to import it separately:

```
import random
```

3.5.2. Choosing at random

The **`random`** module provides us with the function **`choice`**, which randomly chooses an element from a given **sequence**:

```
from microbit import *
import random

FACES = [Image.HAPPY, Image.ASLEEP, Image.SAD]

while True:
  display.show(random.choice(FACES))
  sleep(1000)
```



Run this program. Every second, **`random.choice(FACES)`** chooses and image from the list **`FACES`** to show.

Because it chooses randomly, the sequence is hard to guess. Sometimes a face repeats, but overall each face is **equally likely** to be shown.

3.5.3. Problem: Battle pets

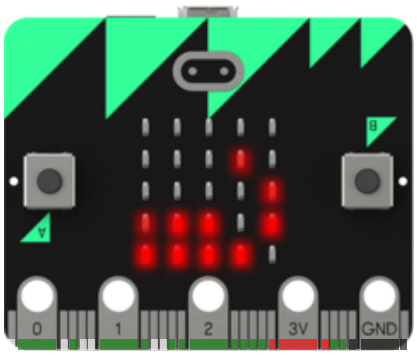
Now that we can add randomness, let's play Rock–paper–scissors (https://en.wikipedia.org/wiki/Rock%E2%80%93paper%E2%80%93scissors) but with virtual pets!.

We're going to play with a mouse, a cat and an elephant. The cat eats the mouse, the elephant stomps the cat and the mouse scares the elephant!

If Button A has been pressed, scroll `321`, then use `random.choice` to show one of the given pets (mouse, cat or elephant) at random.

The marker will expect you to use `random.choice`. Remember that you need to pass a list to `random.choice`.

Here's an example interaction (which you can play against by using a micro:bit programmed with the sample solution from the **I choose you!** problem):



You'll need

program.py

```
from microbit import *
import random

MOUSE = Image('00000:00060:00007:69909:99990')
CAT = Image('09009:99009:09909:09909:09990')
ELEPHANT = Image('09900:93999:89999:70909:60909')
```

Testing

- [ ] Checking that your code contains an infinite loop.
- [ ] Testing that the display starts blank.
- [ ] Testing that something happens when you press A.
- [ ] Testing that 3 2 1 scrolls past after pressing A.
- [ ] Testing that one of the mouse, cat or elephant is shown after 3 2 1.
- [ ] Testing that the mouse, cat or elephant image stays on the screen.
- [ ] Testing that you passed three items to random.choice().
- [ ] Testing that it shows the mouse correctly.
- [ ] Testing that it shows the cat correctly.
- [ ] Testing that it shows the elephant correctly.

3.6. Temperature

3.6.1. Numbers versus strings

When you try to `scroll` a number on the display, you get an error:

```
from microbit import *

count = 7
display.scroll(count)
```

```
Traceback (most recent call last):
  File "__main__", line 4, in &lt;module&gt;
TypeError: can't convert 'int' object to str implicitly
```

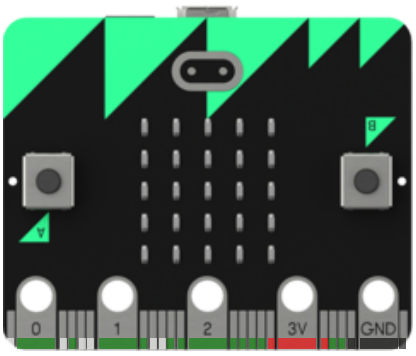
`display.scroll` complains! It can only scroll *strings* (messages), and doesn't know what to do with *integers* (whole numbers).

Python gives a `TypeError` because **strings and integers are different *types* of information**.

3.6.2. Converting numbers to strings

To fix this, we use the `str` function to turn an integer into a string:

```
from microbit import *

count = 7
display.scroll(str(count))
```



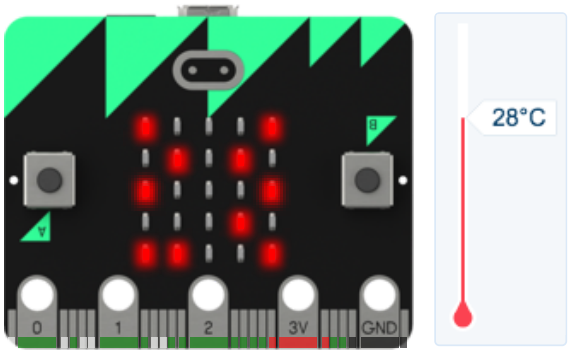
3.6.3. Reading the temperature

The micro:bit has a temperature sensor on the board. We can read it (in degrees Celsius (https://en.wikipedia.org/wiki/Celsius) or °C) by calling the `temperature` function.

This example scrolls the temperature (after converting to a string):

```
from microbit import *

while True:
  the_temp = temperature()
  display.scroll(str(the_temp))
```

Once the program is running, **drag the arrow on the thermometer to change the simulated temperature:**



Don't forget the brackets after `temperature`. We need them to call the function even though it doesn't have arguments, just like `was_pressed()`.

3.6.4. The `or` operator

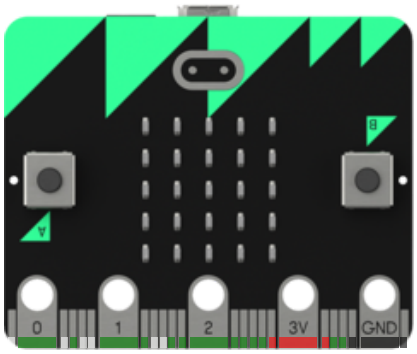
We've used the `and` operator to check whether two conditions are both `True` (e.g. whether both buttons are pressed at once).

The `or` operator lets us run code when **either condition is `True`**:

- if the first condition is `True`; *or*
- if the second condition is `True`; *or*
- if they're *both* `True`.

Try out this program pressing just Button A, just Button B, and both buttons. When does the face show?

```
from microbit import *

while True:
  if button_a.is_pressed() or button_b.is_pressed():
    display.show(Image.HAPPY)
  else:
    display.clear()
```



3.6.5. Problem: Hatching Chicks

Chicken eggs take about 21 days to hatch, but you have to keep them at the right temperature the whole time. No more than 38°C and no less than 37°C.

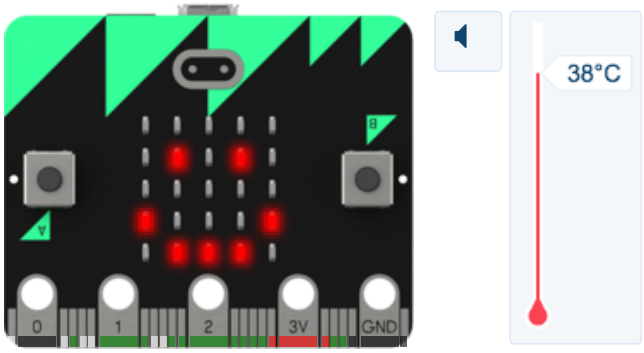
Many egg incubators (https://en.wikipedia.org/wiki/Incubator_(egg)) use embedded devices to check this.

Write a program that monitors the temperature. If it's less than 37°C or more than 38°C, play the note `C6` (4 beats long) on loop as an alarm, and scroll the temperature on the display.

Otherwise, if the temperature is safe, it should stop playing the alarm and display a happy face (`Image.HAPPY`).

Remember you can set the temperature in the simulator (by dragging the arrow) to test your program.

Here's an example interaction with the program:



You'll need

program.py

```
from microbit import *
import music
```

Testing

- [ ] Checking that your code contains an infinite loop.
- [ ] Testing that something plays when the temperature is too low (25°C).
- [ ] Testing that the alarm tone plays when the temperature is too low (25°C).
- [ ] Testing that the alarm tone loops when the temperature is too low (25°C).
- [ ] Testing that your program scrolls a low temperature (28°C).
- [ ] Testing that a temperature of 37°C stops the alarm.
- [ ] Testing that a temperature of 37°C displays the happy face.
- [ ] Testing that a temperature of 38°C also stops the alarm and shows the happy face.
- [ ] Testing that unsafe temperatures close to safe temperatures trigger the alarm.
- [ ] Testing that your alarm holds up to multiple cycles of safe/unsafe temperatures.