Sense HAT Pong

Introduction

In this activity you will create a Pong game using your Raspberry Pi, a Sense HAT, and some Python code. Pong is one of the oldest graphical games ever created, and was originally played on an oscilloscope!

What you will make

Open the trinket below to try out the game. Use your keyboard's arrow keys to control the paddle. Don't let the ball touch the edge or you will lose the game!

What you will learn

By coding a Pong game with your Raspberry Pi and Sense HAT, you will learn how to:

- Illuminate Sense HAT pixels
- Detect movements of the Sense HAT joystick

This resource covers elements from the following strands of the Raspberry Pi Digital Making Curriculum (https://www.raspberrypi.org/curriculum/):

- Apply programming constructs to solve a problem (https://www.raspberrypi.org/curriculum/programming/builder)
- Combine inputs and/or outputs to create projects or solve a problem (https://www.raspberrypi.org/curriculum/physical-computing/builder)

What you will need

Hardware

- Raspberry Pi
- Sense HAT (If you do not have a Sense HAT, you could create the project in a web browser using the Sense HAT emulator (https://trinket.io/sense-hat), or use the emulator software on you Pi.)

Software

You will need the latest version of Raspbian (https://www.raspberrypi.org/downloads/), which already includes the following software packages:

- Python 3
- Sense HAT for Python 3

If for any reason you need to install a package manually, follow these instructions:

Install a software package on the Raspberry Pi

Your Raspberry Pi will need to be online to install packages. Before installing a package, update and upgrade Raspbian, your Raspberry Pi's operating system.

- Open a terminal window and enter the following commands to do this:


```
sudo apt-get update
sudo apt-get upgrade
```

- Now you can install the packages you'll need by typing `install` commands into the terminal window. For example, here's how to install the Sense HAT software:

```
sudo apt-get install sense-hat
```

Type this command into the terminal to install the Sense HAT package:

```
sudo apt-get install sense-hat
```

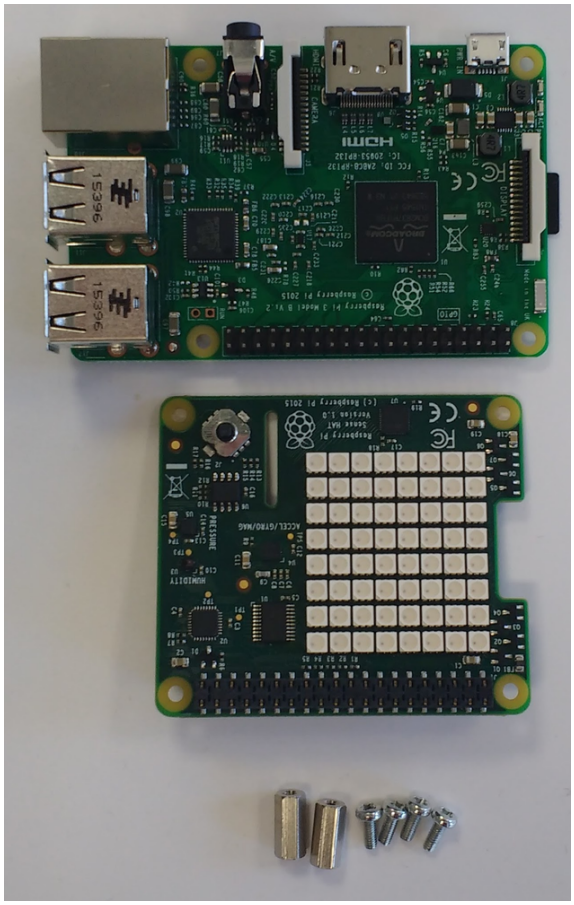
Connect to the Sense HAT

- If you have a Sense HAT, attach it to your Raspberry Pi.

Attaching a Sense HAT

Before attaching any HAT to your Raspberry Pi, ensure that the Pi is shut down.

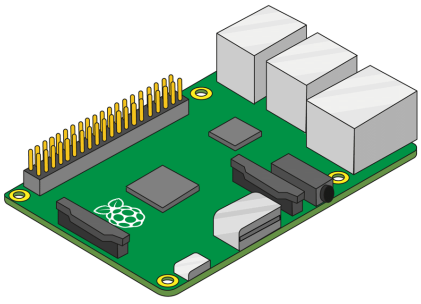
- Remove the Sense HAT and parts from their packaging.



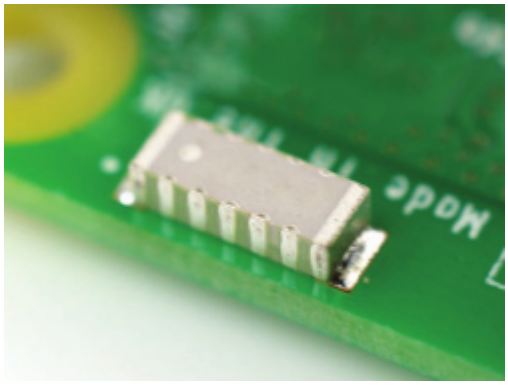
- Use two of the provided screws to attach the spacers to your Raspberry Pi, as shown below.

Note: the above step is optional — you do not have to attach the standoffs to the Sense HAT for it to work.

- Then push the Sense HAT carefully onto the pins of your Raspberry Pi, and secure it with the remaining screws.



Note: using a metal stand-off next to the Raspberry Pi 3's wireless antenna will degrade its performance and range. Either leave out this stand-off, or use nylon stand-offs and nylon screws instead.



Pro tip: be careful when taking off the Sense HAT, as the 40-pin black header tends to get stuck.

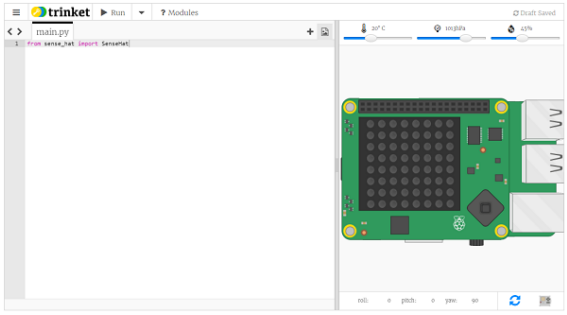
- If you do not have a Sense HAT, you can create the project using the Sense HAT emulator.

Using the Sense HAT emulator

If you don't have access to a Sense HAT, you can use the emulator.

Online Sense HAT emulator

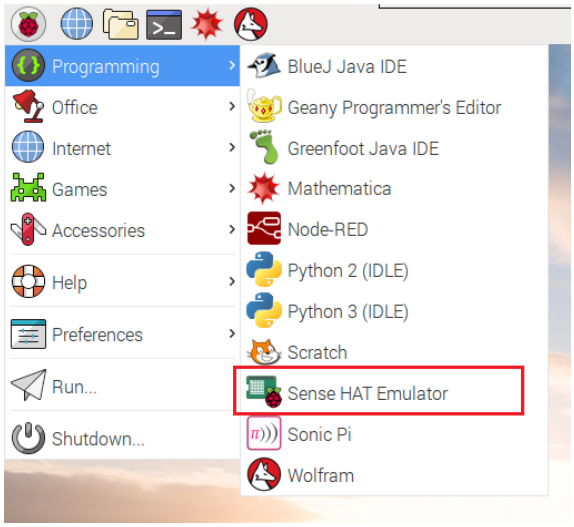
There is an online emulator you can use in your browser to write and test code for the Sense HAT.



- Open an internet browser and go to https://trinket.io/sense-hat (https://trinket.io/sense-hat).
- If you would like to save your work, you will need to create a free account (https://trinket.io/signup) on the Trinket website.

Sense HAT emulator on the Raspberry Pi

If you are using a Raspberry Pi, there is a Sense HAT emulator included in the Raspbian operating system.



- From the main menu, select **Programming** > **Sense HAT emulator** to open a window containing the emulator.
- If you are using this version of the emulator, your program must import from `sense_emu` instead of `sense_hat`:

```
from sense_emu import SenseHat
```

If you later want to run your code on a real Sense HAT, just change the import line as shown below. All other code can remain exactly the same.

```
from sense_hat import SenseHat
```

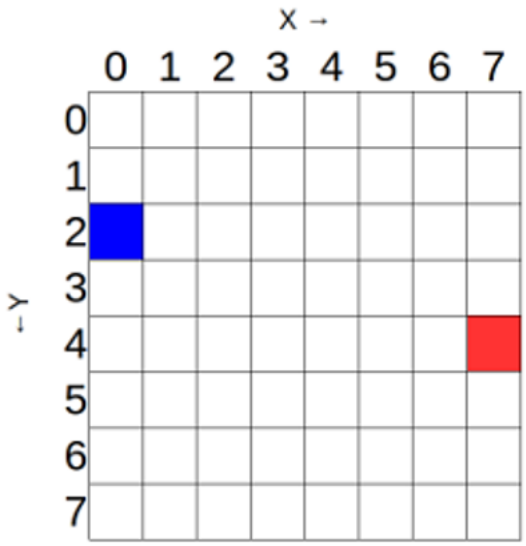
Light up an LED

Games often use the coordinates `x` and `y` to determine where an object is on the display. `x` is used to set the horizontal position of an object, and `y` is used to set the vertical position of an object.

We can do the same with the LEDs on the Sense HAT.

Sense HAT LED matrix coordinates

The Sense HAT's LED matrix uses a coordinate system with an x- and a y-axis. The numbering of both axes begins at `0` (not 1) in the top left-hand corner. Each LED can be used as one pixel of an image, and it can be addressed using an `x, y` notation.



The blue pixel is at coordinates `0, 2`.
The red pixel is at coordinates `7, 4`.

You can set pixels (LEDs) individually using the `set_pixel()` method.

To replicate the diagram above, you would enter a program like this:

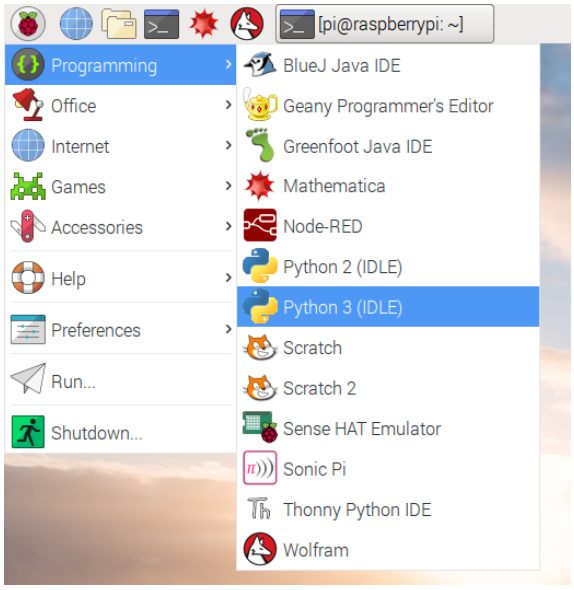
Let's start our Pong game by lighting up a single LED to create a ball, and then adding a few more to create a bat.

- Open IDLE if you are using a physical Sense HAT, or open a new trinket if you are using the emulator (http://trinket.io/sense-hat).

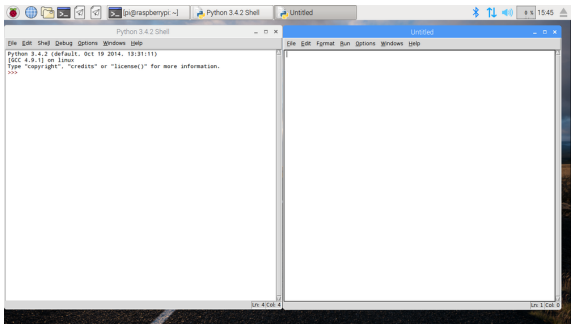
Opening IDLE3

IDLE is Python's **I**ntegrated **D**eve**l**opment **E**nvironment, which you can use to write and run code.

To open IDLE, go to the menu and choose `Programming`.
You should see two versions of IDLE - make sure you click on the one that says `Python 3 (IDLE)`.



To create a new file in IDLE, you can click on `File` and then `New File` in IDLE's menu bar.
This will open a second window in which you can write your code.



- Add this code at the start of your file to import the `sense_hat` module and connect to the Sense HAT.

```
from sense_hat import SenseHat
sense = SenseHat()
```

- The bat will be white. Define a variable called `white`, and set its value to `(255, 255, 255)`, which is the RGB colour representation of white.

Displaying a colour on the Sense HAT

- In a Python file, type in the following code:

```
from sense_hat import SenseHat

sense = SenseHat()

r = 255
g = 255
b = 255

sense.clear((r, g, b))
```

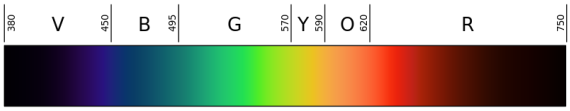
- Save and run your code. The LED matrix will then go bright white.
- The variables `r`, `g`, and `b` represent the colours red, green, and blue. Their values specify how bright each colour should be; each value can be between `0` and `255`. In the above code, the maximum value for each colour has been used, so the result is white.
- You can also define all three RGB values of a colour using a single line of code:

```
red = (255,0,0)
```

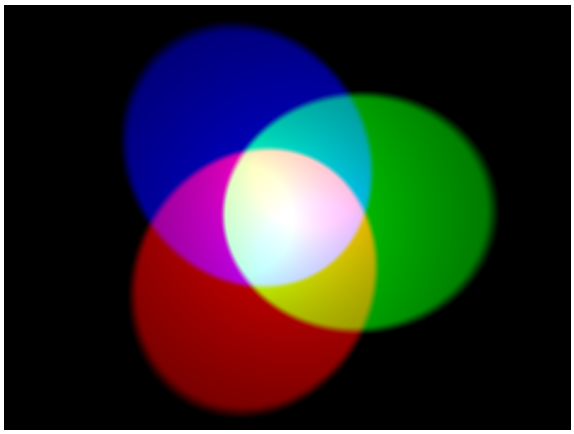
- Change the value of one of the colours, then run the code again. What do you see?
- Which other colours can you make?

Representing colours with numbers

The colour of an object depends on the colour of the light that it reflects or emits. Light can have different wavelengths, and the colour of light depends on the wavelength it has. The colour of light according to its wavelength can be seen in the diagram below. You might recognise this as the colours of the rainbow.



Humans see colour because of special cells in our eyes. These cells are called *cones*. We have three types of cone cells, and each type detects either red, blue, or green light. Therefore all the colours that we see are just mixtures of the colours red, blue, and green.



In additive colour mixing, three colours (red, green, and blue) are used to make other colours. In the image above, there are three spotlights of equal brightness, one for each colour. In the absence of any colour the result is black. If all three colours are mixed, the result is white. When red and green combine, the result is yellow. When red and blue combine, the result is magenta. When blue and green combine, the result is cyan. It's possible to make even more colours than this by varying the brightness of the three original colours used.

Computers store everything as 1s and 0s. These 1s and 0s are often organised into sets of 8, called **bytes**.

A single byte (A set of 8 bits, for example 10011001) can represent any number from 0 up to 255.

When we want to represent a colour in a computer program, we can do this by defining the amounts of red, blue, and green that make up that colour. These amounts are usually stored as a single byte (A set of 8 bits, for example 10011001) and therefore as a number between 0 and 255.

Here's a table showing some colour values:

Red	Green	Blue	Colour
255	0	0	Red
0	255	0	Green
0	0	255	Blue
255	255	0	Yellow
255	0	255	Magenta
0	255	255	Cyan

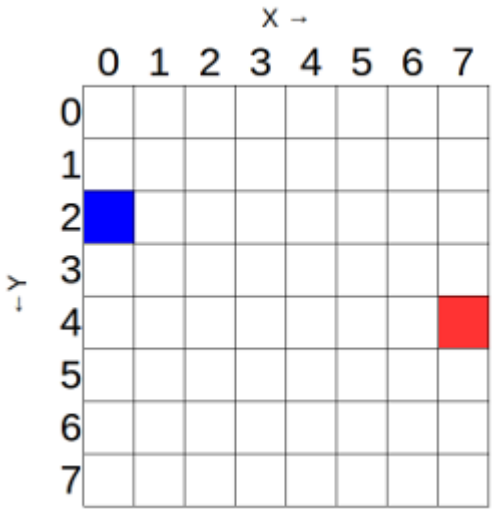
You can find a nice colour picker to play with at w3schools (https://www.w3schools.com/colors/colors_rgb.asp).

The bat will always be on the far left-hand column of pixels, so its `x` value will always be `0`, but its `y` value will change as you move the bat up and down.

- Create another variable `bat_y` and set its value to `4`.
- Set the LED at the position `(0, bat_y)` to `white` using the `set_pixel` method.

Setting a single pixel on the Sense HAT

You can use the `set_pixel` command to control individual LEDs on the Sense HAT. To do this, you set the `x` and `y` variables the `set_pixel` command takes. `x` indicates the HAT's horizontal axis, and can have a value between `0` (on the left) and `7` (on the right). `y` indicates the HAT's vertical axis, and can have a value between `0` (at the ltop) and `7` (at the bottom). Therefore, the `x, y` coordinates `0, 0` address the top left-hand LED, and the `x, y` coordinates `7, 7` address the bottom right-hand LED.



The grid above corresponds with the Raspberry Pi when it is this way around:

Let try out this example for setting a different colour in each corner of the Sense HAT's LED matrix. You will need to use the `set_pixel` command multiple times in your code, like this:

```
from sense_hat import SenseHat

sense = SenseHat() # This clears any pixels left on the Sense HAT.
You may not need this step and may want to choose when to add it in.

sense.clear()
sense.set_pixel(0, 0, 255, 0, 0)
sense.set_pixel(0, 7, 0, 255, 0)
sense.set_pixel(7, 0, 0, 0, 255)
sense.set_pixel(7, 7, 255, 0, 255)
```

Test setting the colour of different pixels using the Sense HAT Emulator:

I need a hint

Your finished code should look like this:

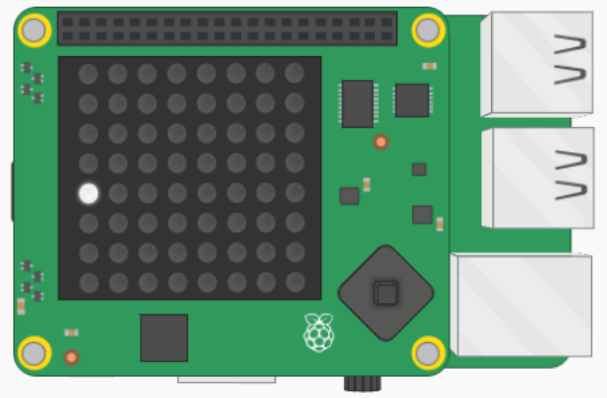
```
from sense_hat import SenseHat
sense = SenseHat()

white = (255, 255, 255)

bat_y = 4

sense.set_pixel(0, bat_y, white)
```

- Save and run your code. A single LED should now be illuminated in white on the left side of the LED matrix.



Make a bat

Let's draw the rest of the bat by illuminating the LEDs immediately above and below the one that's currently illuminated. To do this, we will make a **function**.

Creating and calling functions in Python

When coding, sometimes you may want to use the same few lines of code multiple times within your script. Alternatively, you may want to have the same few lines of code run every time a certain event occurs, e.g. when a specific key is pressed, or a particular phrase is typed. For tasks like this, you might want to consider using a **function**.

Functions are **named** blocks of code that perform a defined task. Just about the simplest function you can create in Python looks like this:

```
def hello():
    print('Hello World!')
```

You tell Python that you're creating a new function by using the `def` keyword, followed by the name of the function. In this case it is called `hello`. The parentheses after the function name are important.

The colon at the end of the line indicates that the code inside the function will be indented on the next line, just like in a `for` or `while` loop or an `if/elif/else` conditional.

You can **call (run the lines of code within the function)** a function by typing its name with the parentheses included. So to run the example function, you would type `hello()`. Here's the complete program:

```
def hello():
    print('Hello World!')

hello()
```

- **Indent** the line `sense.set_pixel(0, bat_y, white)` by putting your cursor at the start of the line and pressing the **tab** key.
- On the line immediately above this line, start a function called `draw_bat`:

main.py

```
1  from sense_hat import SenseHat
2  sense = SenseHat()
3
4  white = (255, 255, 255)
5
6  bat_y = 4
7
8  def draw_bat():
9    sense.set_pixel(0, bat_y, white)
```

The lines following the start of a function are indented to show that they are **inside** the function.

You could add a **comment** just above the start of your function to show that this section will contain your functions — we will be writing some more later on.

```
# Functions ----------------
```

- Add two more lines of code inside the function to illuminate the LEDs at positions `bat_y + 1`, and `bat_y - 1` as well.

I need a hint

Here is how your function should look:

```
# Functions ----------------
def draw_bat():
        sense.set_pixel(0, bat_y, white)
        sense.set_pixel(0, bat_y + 1, white)
        sense.set_pixel(0, bat_y - 1, white)
```

If you run your code now, nothing will happen. The code you just wrote inside the function will not do anything at all until the function is **called**.

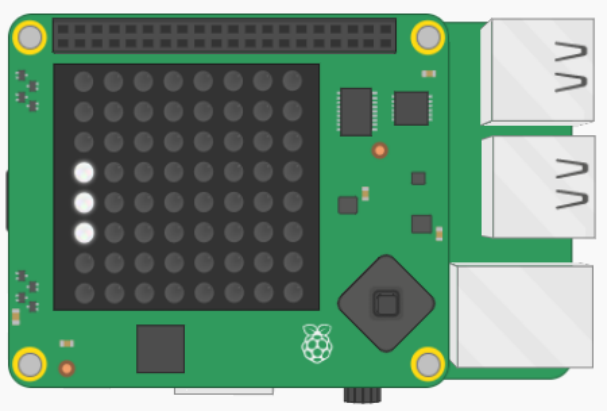
- Add a new comment underneath the function to show that this section is where the main program starts. Make sure this comment is **not indented**.

```
# Main program -------------
```

- Add this line of code in the main program section to call the function:

```
draw_bat()
```

- Run the code and check that three LEDs are now illuminated.



Move the bat

Let's make the bat move up and down when the Sense HAT's joystick is moved.

- In your functions section, define a new function called `move_up(event)`.

Some data called `event` will be passed to this function. The event data the function will receive is information about has happened to the Sense HAT joystick. This will include the time that the stick was used, the direction it was pushed in, and whether it was pressed, released, or held.

- Inside the `move_up` function, add an if statement to test if the `event.action` was `'pressed'` (in other words, whether the joystick was moved).

```
if event.action == 'pressed':
```

If the condition is met, we want the bat to move upwards. Upwards in the coordinate system on our LED screen means making the y coordinate smaller - remember that the top pixel's y coordinate is `0`.

- If the `event.action` was `'pressed'`, take away `1` from the `bat_y` coordinate. This will allow us to redraw the bat at a different position. **Note:** because the `bat_y` variable is defined outside of this function, we also have to tell Python to use the **global** version of this variable so that we are allowed to change it from inside the function.

main.py

```
8  # Functions ----------------
9  def draw_bat():
10   sense.set_pixel(0, bat_y, white)
11   sense.set_pixel(0, bat_y + 1, white)
12   sense.set_pixel(0, bat_y - 1, white)
13
14 def move_up(event):
15   global bat_y
16   if event.action == 'pressed':
17     bat_y -= 1
```

Remember that just like our `draw_bat` function, this function will do nothing until it is **called**.

- In main program section, add this line of code above the `draw_bat` function call. This line says "When the Sense HAT stick is pushed up, call the function `move_up`."

```
sense.stick.direction_up = move_up
```

If you run your code at this point, nothing will happen. This is because, at the moment, we are only checking for joystick movement once when the function is run. To make this function useful for our game, we need to continually check whether the joystick was moved.

- In your main program section, put the function call to `draw_bat` inside an infinite loop.

While True loop in Python

The purpose of a **while** loop is to repeat code over and over while a condition is `True`. This is why while loops are sometimes referred to as **condition-controlled** loops.

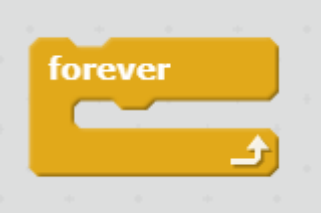
The example below is a while loop that will run forever - an infinite loop. The loop will run forever because the condition is always `True`.

```
while True:
    print("Hello world")
```

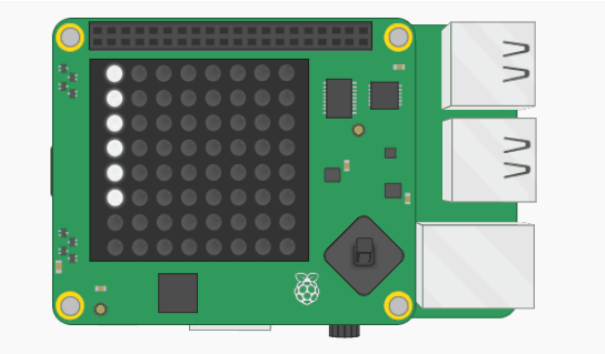
Note: The `while` line states the loop **condition**. The `print` line of code below it is slightly further to the right. This is called **indentation** - the line is indented to show that it is inside the loop. Any code inside the loop will be repeated.

An infinite loop is useful in situations where you want to perform the same actions over and over again, for example checking the value of a sensor. An infinite loop like this will **block** - this means that any lines of code written after the loop will never happen.

If you have used Scratch before, this should be familiar, as it is the same as using a forever loop.



- Save and run your code. Press the joystick on the Sense HAT up (or use the arrow keys on your keyboard if you are using the emulator).



Oh dear — the result looks a bit like you are smudging the bat upwards on the screen rather than moving it! We need to clear the screen and wait a while before each time we draw the bat in the infinite loop.

- Add this line to your infinite loop to clear the LED matrix each time before the bat is drawn.

```
sense.clear(0, 0, 0)
```

- To make the program wait a little while, add a line inside the loop after `draw_bat` to `sleep` for 0.25 seconds.

Using Python's sleep command

You can use the `sleep` function to temporarily pause your Python program.

- Add this line of code at the top of your program to import the `sleep` function.

```
from time import sleep
```

- Whenever you want a pause in your program, call the `sleep` function. The number in the brackets indicates how many seconds you would like the pause to be.

```
sleep(2)
```

You can pause for fractions of a second as well.

```
sleep(0.5)
```

- Save and run your code again. Try moving the bat and check whether it now moves up as expected.

If you move the bat too far upwards, your program tries to draw it outside the LED screen, and then the program crashes. You need to make sure that the value of the `bat_y` variable is never less than `1`, so that the bat remains on the grid at all times.

- Add code to your `move_up` function to make sure the `bat_y` variable's value can never become smaller than `1`.

main.py

```
15 def move_up(event):
16   global bat_y
17   if event.action == 'pressed' and bat_y > 1:
18     bat_y -= 1
```

- Now follow these steps again, making a few changes to allow you to to move your bat **downwards** on the LED matrix as well as upwards.

I need a hint

Here is how your code should look:

main.py

```
20 def move_down(event):
21   global bat_y
22   if event.action == 'pressed' and bat_y < 6:
23     bat_y += 1
24
25 # Main program -------------
26 sense.stick.direction_up = move_up
27 sense.stick.direction_down = move_down
```

Create a ball

The next step is to create the ball. But first, a little maths!

A ball moving in two dimensions has two essential properties you need to consider:

Position - like the bat, the ball has a vertical and horizontal coordinate on the grid.

Velocity - the speed of the ball in a straight line. This can also be described by two numbers: how fast it's moving in the `x` dimension, and how fast it's moving in the `y` dimension.

- Locate the `bat_y` variable in your program and, below it, add two lists to describe the ball's properties:

```
ball_position = [3, 3]
ball_velocity = [1, 1]
```

- Choose a colour for your ball and, below your `white` variable, define a variable with your chosen colour value. We have used `(0, 0, 255)`, which is blue.

- In your functions section, create a function called `draw_ball`:

```
def draw_ball():
```

- Add a line of code to the `draw_ball` function to illuminate an LED at the `ball_position`.

Index a list in Python

A Python list is a type of data structure. It can hold collections of any data type, and even a mixture of data types.

- Here is an example of a list of strings in Python:

```
band = ['paul', 'john', 'ringo', 'george']
```

- In Python, lists are indexed from `0`. That means you can talk about the zeroth item in a list. In our example, the zeroth item is `'paul'`.

- To find the value of an item in a list, you simply type the name of the list followed by the index.

```
>>> band[0]
'paul'
>>> band[2]
'ringo'
```

- To find the value of the last item in a Python list, you can also use the index `-1`.

```
>>> band[3]
'george'
>>> band[-1]
'george'
```

- Moreover, you can find out value of the penultimate item using the index `-2`, and so on.

- Sometimes you might want to use a two-dimensional list, i.e. a list of lists. To find a specific item, you will have to provide two indices. Here is a list representing a noughts and crosses game:

```
board = [['X', 'O', 'X'],
         ['O', 'X', 'O'],
```

```
        ['O', 'O', 'X']]
```

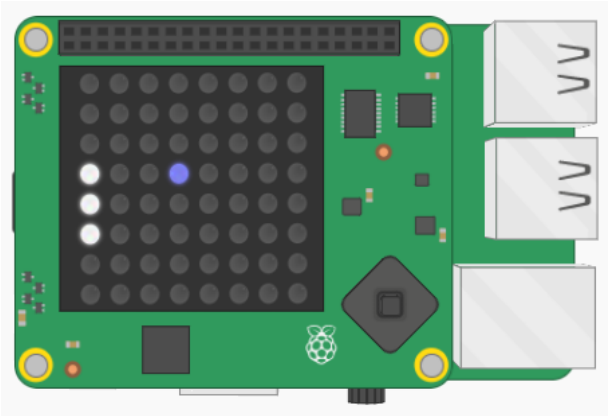
- To find the central character in this list of lists, you would use `board[1][1]`.

I need a hint

Here is how your code should look (assuming you also chose the colour blue for your ball):

```
def draw_ball():
    sense.set_pixel(ball_position[0], ball_position[1], blue)
```

- In your `while` loop, call the function `draw_ball`.
- Save and run your code, and check that the ball is displayed on the LED matrix.



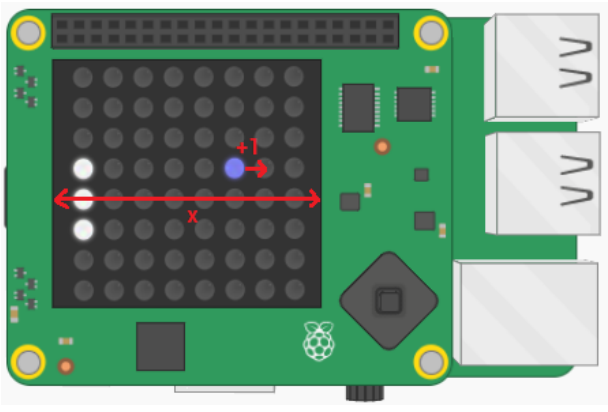
Move the ball

To get the ball moving, you need to change its **x** position by its **x** velocity, and its **y** position by its **y** velocity.

The first coordinate in each list you just created represents the ball's **x** property — so `ball_position[0]` is the current **x** coordinate, and `ball_velocity[0]` is how fast it should move in the **x** direction.

- Inside your `draw_ball` function, add this line of code to add the ball's velocity (currently `1`) to the ball's current position in the **x** direction.

```
ball_position[0] += ball_velocity[0]
```



- Save and run your code. The ball will move across the screen until it reaches the edge, and then the program will crash. Why do you think this happens?

Answer

You've probably seen the same error before when you moved the paddle. The ball moves across the LED matrix and then the program crashes with the error `ValueError: X position must be between 0 and 7`.

The ball moved to an `x` position larger than 7, which is outside of the LED matrix.

- Immediately after the line of code to move the ball, add a conditional stating that, if the `ball_position[0]` reaches `7`, its velocity gets reversed so it goes in the other direction:

```
if ball_position[0] == 7:
    ball_velocity[0] = -ball_velocity[0]
```

- Save and run your code again. The ball should bounce off the right edge of the matrix — but when it reaches the left edge, you'll get another error because it is still trying to move off the screen in that direction!

- Add to the conditional to say that the ball should reverse direction if its position is equal to `7` **or** is equal to `0`.

I need a hint

Here is how your code should look:

```
if ball_position[0] == 7 or ball_position[0] == 0:
    ball_velocity[0] = -ball_velocity[0]
```

Why does this work?

The ball's velocity starts off as `1`. If the ball's `x` position equals `7`, we change the `x` velocity to `-1` to make the ball reverse. Then the code will be adding `-1` to the ball's `x` position to move the ball to the left across the matrix.

But why does this work when the ball gets all the way to the left? Look at the code:

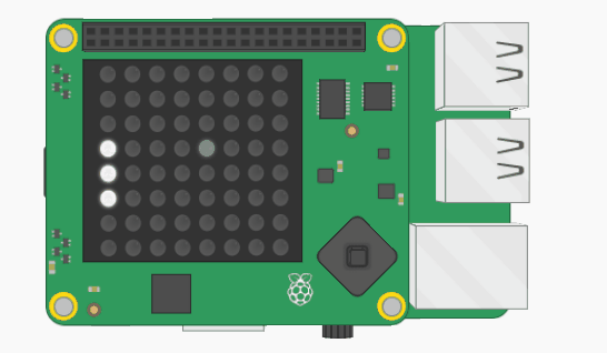
```
ball_velocity[0] = -ball_velocity[0]
```

When the ball is travelling leftward, its `x` velocity is `-1`. When we insert this value in the line of code, we get the following:

```
ball_velocity[0] = -(-1)
```

Minus (minus one) equals…plus one! So the velocity is now `1`, and the ball begins travelling back the other way.

- Save and run your program to check that your ball bounces happily from the left edge to the right edge.



- Now make your ball move according to its **y** velocity as well as its **x** velocity by following these steps again with a few changes.

I need a hint

The highlighted code is the part you should add:

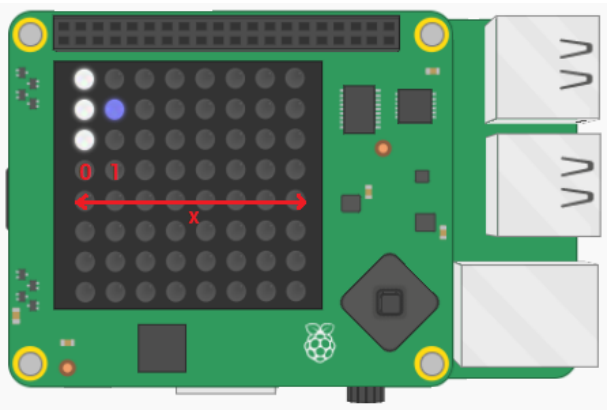
```
main.py
28 def draw_ball():
29     sense.set_pixel(ball_position[0], ball_position[1], blue)
30     ball_position[0] += ball_velocity[0]
31     if ball_position[0] == 7 or ball_position[0] == 0:
32         ball_velocity[0] = -ball_velocity[0]
33     ball_position[1] += ball_velocity[1]
34     if ball_position[1] == 7 or ball_position[1] == 0:
35         ball_velocity[1] = -ball_velocity[1]
```

Collision with the bat

Now that the ball bounces in both directions, let's make it bounce off the bat.

The bat is always situated in the far left column of the LED grid, so its **x** coordinate is always **0**.

The ball will bounce off the bat if it is in the row next to the bat — that is, if the ball's **x** position is equal to **1**.



- Add this code to the end of the `draw_ball` function:

```
if ball_position[0] == 1:
    ball_velocity[0] = -ball_velocity[0]
```

This code will cause the ball to reverse direction if it reaches an **x** coordinate of **1**. But now the ball reverses regardless of whether the bat is there or not!

- Add to the condition to require the ball's **y** position to also (**and**) be anywhere between the top and bottom of the bat.

Remember that the bat is made up of three pixels. So for the ball to 'bounce off' the bat, the y coordinate of the ball can be anywhere **between** the top of the bat (`bat_y - 1`) and the bottom of the bat (`bat_y + 1`).

I need a hint

Here is how your finished code should look. The bit you should add is highlighted in blue:

main.py

```
def draw_ball():
  sense.set_pixel(ball_position[0], ball_position[1], blue)
  ball_position[0] += ball_velocity[0]
  if ball_position[0] == 7 or ball_position[0] == 0:
      ball_velocity[0] = -ball_velocity[0]
  ball_position[1] += ball_velocity[1]
  if ball_position[1] == 7 or ball_position[1] == 0:
      ball_velocity[1] = -ball_velocity[1]
  if ball_position[0] == 1 and (bat_y - 1) <= ball_position[1] <= (bat_y +1):
      ball_velocity[0] = -ball_velocity[0]
```

- Save and run your code. Check that the ball bounces off the bat only when the bat is in the correct position!

You lose

If you miss the ball with the bat, at the moment it bounces off the far left wall. Let's change the code so that if the player misses the ball, they lose the game.

- Add another if statement at the end of your `draw_ball` function to check whether the ball's `x` position equals `0`, which would mean the ball has reached the far end of the screen.
- If this condition is true, display the message "You lose".

I need a hint

Here is how your code should look. The part to add is highlighted in blue:

main.py

```
def draw_ball():
  sense.set_pixel(ball_position[0], ball_position[1], blue)
  ball_position[0] += ball_velocity[0]
  if ball_position[0] == 7 or ball_position[0] == 0:
      ball_velocity[0] = -ball_velocity[0]
  ball_position[1] += ball_velocity[1]
  if ball_position[1] == 7 or ball_position[1] == 0:
      ball_velocity[1] = -ball_velocity[1]
  if ball_position[0] == 1 and (bat_y - 1) <= ball_position[1] <= (bat_y +1):
      ball_velocity[0] = -ball_velocity[0]
  if ball_position[0] == 0:
    sense.show_message("You lose")
```

- Save and run your code. Check that, if you miss the ball, the message "You lose" appears. The game will restart after the message has been displayed.

Challenge: add more features

- Add a score which increases each time the ball bounces off the bat. Report the score to the player when they lose.
- Make the game easier or harder by changing the length of the `sleep` period in the while loop.
- Give the player three lives: each time they miss the ball, they lose a life and the ball resets. Once all lives are lost, the game ends.

Hour of Code

I've finished my Hour of Code! (https://code.org/api/hour/finish)