Raspberry Pi Projects

Flappy Astronaut

What you will make

In this activity you will create a Flappy Bird clone using your Raspberry Pi, a Sense HAT, and some Python code. In your Flappy Astronaut game the player uses the Sense HAT joystick to navigate an astronaut around pipes.

What you will learn

By making a Flappy Astronaut game with your Raspberry Pi and Sense HAT, you will learn how to:

- Light up Sense HAT LEDs
- Use the Sense HAT joystick
- Use 2D lists to represent a screen

Click on the arrow in the emulation below to play the game. Use your cursor keys to guide your astronaut around the pipes.

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

- Apply abstraction and decomposition to solve more complex problems (https://www.raspberrypi.org/curriculum/programming/developer)
- Use basic digital, analogue, and electromechanical components (https://www.raspberrypi.org/curriculum/physical-computing/creator)

What you will need

Hardware

If you are not using an emulator you will need the following hardware:

- Raspberry Pi
- Sense HAT

If you are using an emulator then you simply require a computer with access to https://trinket.io/.

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 a terminal window to install the Sense HAT package:

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

Flappy Astronaut

While the astronauts on the ISS are kept pretty busy, every now and then they still need to have a few minutes of relaxation. There are two Raspberry Pis with Sense HATs up there with them, so a little game of Flappy Astronaut would be the perfect way for an astronaut to unwind after a hard day's work in zero gravity.

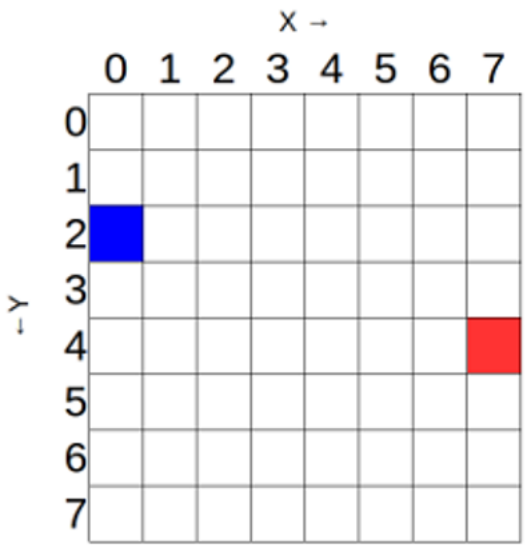
Representing pixels

The Sense HAT has an 8×8-pixel LED matrix, and each of the LED pixels can be illuminated in any colour. This is going to be the screen you will use to display your Flappy Astronaut game.

When programmers want to colour a specific pixel on a screen, they normally refer to its `x` and `y` coordinates. The same is true of the Sense HAT's LED matrix. If you want to see how `x` and `y` coordinates can be used to set a specific Sense HAT pixel, have a look at the section below.

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:

- Now try setting a single pixel on the Sense HAT. You can use the following code:

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

r = (255, 0, 0)
sense.set_pixel(0, 0, r)
```

- Have a go at changing the position of the pixel you are setting. Also try and change the colour.

If you want to set multiple pixels in this way, you're going to need a lot of lines of code. Luckily, the Sense HAT lets you set multiple pixels at a time by using a **list**. In the section below, you can read more about this, and see some example code.

Setting multiple pixels on the Sense HAT

You may be tempted to try to draw shapes on the Sense HAT's LED matrix by using the `set_pixel` command over and over. However, there is a `set_pixels` command with which you can change all 64 LEDs using one single line of code!
For example, you could draw a Minecraft creeper face on the LED Matrix:

You can even use more than two colours, like in this example of Steve from Minecraft:

- Have a go at creating an image using a list. You can use this code to begin:

```
from sense_hat import SenseHat

sense = SenseHat()
```

```
g = (0, 255, 0)
b = (0, 0, 0)

creeper_pixels = [
    g, g, g, g, g, g, g, g,
    g, g, g, g, g, g, g, g,
    g, b, b, g, g, b, b, g,
    g, b, b, g, g, b, b, g,
    g, g, g, b, b, g, g, g,
    g, g, b, b, b, b, g, g,
    g, g, b, b, b, b, g, g,
    g, g, b, g, g, b, g, g
]

sense.set_pixels(creeper_pixels)
```

- Try creating another image using the list. How about creating a smiley face?

The only problem with using a single list like this is that it can be tricky to figure out which item in the list corresponds to which pixel on the screen. For instance, what is the list index of the pixel at `x = 5` and `y = 5`? It can be calculated, but it is a little tricky.

To get around this problem, programmers often use two-dimensional lists, which are also known as lists of lists, to represent the arrangement of pixels on a screen.

Here is a simple list of lists to describe a noughts and crosses (tic-tac-toe) board.

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

This is a great way to represent the board, because you can easily use `x` and `y` coordinates to find out what is in each of the squares. For instance, if you want to find out which character is in the bottom left-hand corner, you already know that it has an `x` position of `0` and a `y` position of `2` (don't forget that in Python we start counting items in a list from `0`).

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]`.

To find out the character in that position, you can just write`board[y][x]`. So in this example that would be `board[2][0]`.

Using 2D lists with the Sense HAT

Now you know that the best way to represent the pixels on the LED matrix is using a 2D list, let's see how this can be done with the Sense HAT.

- Open a new Python file, or use the Sense HAT emulator at trinket.io (https://trinket.io/).
- With the first two lines of code, import the Sense HAT modules and create a `SenseHAT` object that can be used to control the LED matrix:

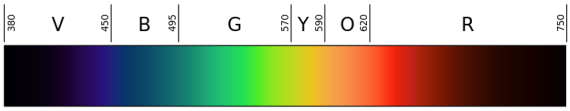
```
from sense_hat import SenseHat

sense = SenseHat()
```

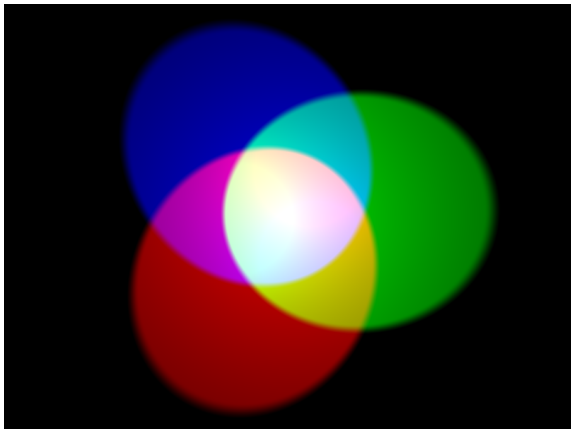
Then you need to create two variables that represent the pixel colours. To make this simple, you can use red and blue. If you want to learn a little more about how computers represent colours, have a look at the section below.

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).

- To store the colour information, you can use a pair of tuples, one for red and one for blue.

```
RED = (255, 0, 0)
BLUE = (0, 0, 255)
```

- Now you are going to create a list of lists filled with the variable `BLUE`. Manually creating it would mean a lot of typing, but instead you can use a list comprehension to complete the task in a single line.

```
matrix = [[BLUE for column in range(8)] for row in range(8)]
```

What does this code do? The section `[BLUE for column in range(8)]` creates one list with eight values of `(0, 0, 255)` inside it. Then the `for row in range(8)` part makes eight copies of that list inside another list. After running the code, you can switch over to the interpreter and type `matrix` if you want to see the result for yourself.

If you want to learn more about list comprehensions, take a look at the section below.

Simple Python list comprehensions

- If you want to generate a list using Python, it is quite easy to do so using a **for loop**.

```
new_list = []
for i in range(10):
    new_list.append(i)


>>> new_list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- The same list can be created in a single line using a construct that exists in many programming languages: a **list comprehension**.

```
new_list = [i for i in range(10)]


>>> new_list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- You can use any **iterable** in a list comprehension, so making a list from another list is easy.

```
numbers = [1, 2, 3, 4, 5]
numbers_copy = [number for number in numbers]


>>> numbers_copy
[1, 2, 3, 4, 5]
```

- You can also do calculations within a list comprehension.

```
numbers = [1, 2, 3, 4, 5]
double = [number * 2 for number in numbers]


>>> double
[2, 4, 6, 8, 10]
```

- String operations can be done as well.

```
verbs = ['shout', 'walk', 'see']
present_participle = [word + 'ing' for word in verbs]


>>> present_participle
['shouting', 'walking', 'seeing']
```

- You can also extend lists quite easily.

```
animals = ['cat', 'dog', 'fish']
animals = animals + [animal.upper() for animal in animals]


>>> animals
['cat', 'dog', 'fish', 'CAT', 'DOG', 'FISH']
```

You can't use this list of lists with the Sense HAT, as its software only understands a **flat** one-dimensional list. To deal with this issue, you are going to create a function that turns 2D lists into 1D lists. You can then use this function every time the `matrix` needs to be displayed.

To flatten a 2D list into a 1D list, you can again use a list comprehension. Here's an example of how to flatten a list.

```
flattened = [pixel for row in matrix for pixel in row]
```

What does this do? The `for row in matrix` part looks at each of the lists in the matrix, and the `for pixel in row` section looks at the individual pixels in each row of that list. These pixels are then all placed into a single list.

- You can turn this into a function to avoid having to write it out all the time. Add this to your file:

```
def flatten(matrix):
    flattened = [pixel for row in matrix for pixel in row]
    return flattened
```

- To flatten your matrix and then display it on the Sense HAT, you can now simply add these lines of code to the bottom of your file.

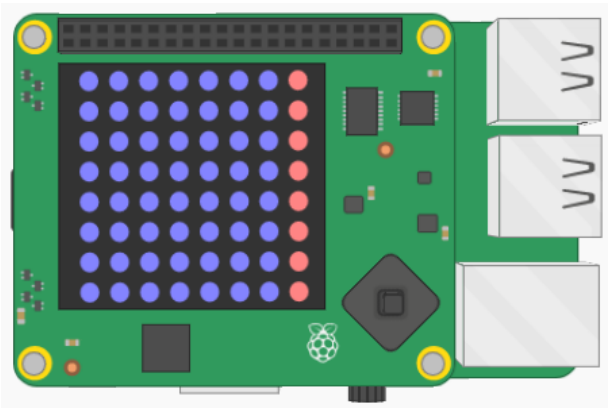
```
matrix = flatten(matrix)
sense.set_pixels(matrix)
```

- Save and run your code. You can see an example of the code and its output in the embedded Trinket below.

Generating pipes

In Flappy Astronaut, the astronaut will have to avoid 'pipes' that sprout from the top and bottom of the matrix. The colour of the pipes is going to be red.

To begin, you can create a single column of red pixels on the right-hand side of the matrix.



All you need to do is to set the last item in each of the lists within the matrix to be `RED` instead of `BLUE`. Below is a refresher of how to access items in a list.

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]`.

- You can use a for loop so that, for each list in the matrix, the last item is set to `RED`. Position this for loop so that it runs before you flatten and display the matrix. You can use the hints below to help you out if you need them.

I need a hint

- Then you set the last item in each list to `RED`.

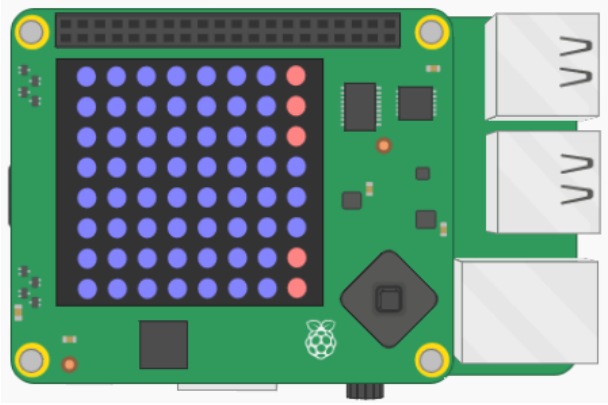
```
for row in matrix:
    row[-1] = RED
```

- Here's what it should do:

Mind the gaps

Now that you have a column of pixels representing a pipe on the right-hand side of the matrix, you need to insert a gap into it through which the astronaut can fly.

The gap needs to be three pixels high, and should be placed randomly in the column of red pixels.



You'll want the three-pixel-high gap to be centred around one of the rows between `1` and `6` (inclusive). You can use the `random` module to achieve this:

Randomness in Python

One of the standard modules in Python is the `random` module. You can use it to create pseudo-random numbers in your code.

randint

You can generate random integers between two values using the `randint` function. For example, the following line of code will produce a random integer between 0 and 10 (inclusive).

```
from random import randint
num = randint(0,10)
```

uniform

If you want a random floating-point number (also called float), you can use the `uniform` function. For example, the following line of code will produce a random float that's equal to or greater than 0, but less than 10.

```
from random import uniform
num = uniform(0,10)
```

choice

If you want to choose a random item from a list, you can use the `choice` function.

```
from random import choice
deck = ['Ace', 'King', 'Queen', 'Jack']
card = choice(deck)
```

- Here's what you need to do:
  - Import the `randint` method at the top of your code
  - After the for loop has ended, create a variable called `gap`, and assign a random number between `1` and `6` as its value
  - Change the last pixel in that row of the matrix to `BLUE`

- Change the last pixel in row `gap - 1` to `BLUE`
- Change the last pixel in row `gap + 1` to `BLUE`

I need a hint

- Now set the pixels in the last column of each of the rows numbered `gap`, `gap + 1`, and `gap -1`.

```
for row in matrix:
  row[-1] = RED
gap = randint(1, 6)
matrix[gap][-1] = BLUE
matrix[gap - 1][-1] = BLUE
matrix[gap + 1][+1] = BLUE
```

- Here's what it should look like:

Create a function to make more pipes

The game would be a little easy if only one set of pipes were created. You can generate as many pipes as you like by using a function.

- Below your `flatten()` function, create a new function called `gen_pipes`:

```
def gen_pipes(matrix):
```

- Put the code you wrote to generate a set of pipes into this function. You can just add some indentation to do this. At the end of the function, you should `return` the altered `matrix`.

```
def gen_pipes(matrix):
    for row in matrix:
      row[-1] = RED
    gap = randint(1, 6)
    matrix[gap][-1] = BLUE
    matrix[gap - 1][-1] = BLUE
    matrix[gap + 1][-1] = BLUE
    return matrix
```

- Then call the function before you flatten and display the `matrix`.

```
matrix = gen_pipes(matrix)
matrix = flatten(matrix)
sense.set_pixels(matrix)
```

- Here's what the code should look like now:

Moving pipes algorithm

Now that you can generate as many pipes as you want, you need to move them across the matrix so that they proceed towards the left of the screen.

It might be easier to picture this on a smaller scale. For instance, here's a 5×5 matrix:

```
0 1 2 3 4
0 b b b b r
1 b b b b r
2 b b b b b
3 b b b b r
4 b b b b r
```

To move the red pixels (`r`) to the left, you can follow a simple algorithm:

1. Move all the items at index `1` in each of the rows to index `0`
2. Move all the items at index `2` in each of the tows to index `1`
3. Move all the items at index `3` in each of the tows to index `2`
4. Move all the items at index `4` in each of the tows to index `3`
5. Fill all the items at index `5` in each row with a `b`

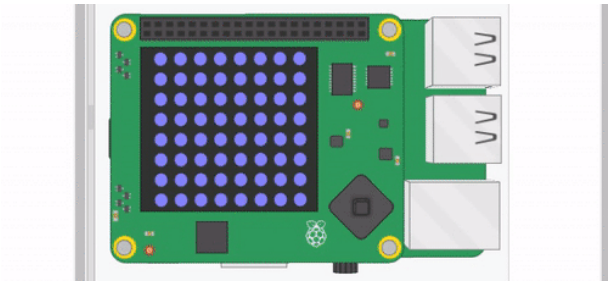
This would then give you a matrix that looks like this:

```
0 1 2 3 4
0 b b b r b
1 b b b r b
2 b b b b b
3 b b b r b
4 b b b r b
```

You could run the algorithm again to repeat the movement, which would give you this:

```
0 1 2 3 4
0 b b r b b
1 b b r b b
2 b b b b b
3 b b r b b
4 b b r b b
```

If you do this with your matrix, then the following will happen:



Moving the pipes

As mentioned in the section before, the algorithm to move the pipes will need to be repeated each time you want to shift the pixels left by `1`. Any code that needs to be repeated can be placed inside a function. Create this function below your `gen_pipes(matrix)` function:

```
def move_pipes(matrix):
```

The algorithm for this function can be broken down like this:

1. For each row in the matrix
    1. For each item in the row from `0` to `7`
    2. Set the item to be the same as the next item in the row
2. Set the last item in the row to be `BLUE`.

- Try and complete this by yourself, and use the hints below if you need some help.

I need a hint

- To finish, set the last item in each row to be `BLUE`, and then return the altered matrix.

```
def move_pipes(matrix):
  for row in matrix:
      for i in range(7):
          row[i] = row[i + 1]
      row[-1] = BLUE
  return matrix
```

Watching the pipes move

At the moment, running your code won't do much. You need to call your functions in a loop to see it working.

Right now you should have these three lines at the bottom of your code:

```
matrix = gen_pipes(matrix)
matrix = flatten(matrix)
sense.set_pixels(matrix)
```

- Instead of using two lines of code to flatten the matrix and then display it, you can use a single line to do this. This will avoid flattening the actual matrix each time, and instead just use a flattened version of the matrix for the display. Replace the last three lines with this:

```
matrix = gen_pipes(matrix)
sense.set_pixels(flatten(matrix))
```

- Now you can add in your `move_pipes(matrix)` function call to move the pipes:

```
matrix = gen_pipes(matrix)
sense.set_pixels(flatten(matrix))
matrix = move_pipes(matrix)
```

- Although this will move the pipes, they won't be displayed, as there is no second `set_pixels` call. To solve this, you can just add in a loop so that moving and displaying always follow each other.

```
matrix = gen_pipes(matrix)
for i in range(9):
    sense.set_pixels(flatten(matrix))
    matrix = move_pipes(matrix)
```

Try and run this code, and see what happens. Was it a little fast?

- You can solve this by adding a `sleep()` command. At the top of your code, import the `sleep` method from the `time` module:

```
from time import sleep
```

- Then add a `sleep` command into your loop:

```
matrix = gen_pipes(matrix)
for i in range(9):
    sense.set_pixels(flatten(matrix))
    matrix = move_pipes(matrix)
    sleep(1)
```

- If you run the code now, you should see something a little more like this:

- One small alteration will give you a continuous stream of pipes. Simply change the for loop to repeat `3` or `4` times, and then enclose the entire last section of code in an infinite `while True` loop:

```
while True:
  matrix = gen_pipes(matrix)
  for i in range(3):
      sense.set_pixels(flatten(matrix))
      matrix = move_pipes(matrix)
      sleep(1)
```

- This should give you something that looks like this:

Adding your astronaut

Your astronaut will be represented by a single coloured pixel. You can choose whatever colour you want for the astronaut, but the example will use yellow.

- Where you have set your other colour variables, now create a new tuple for your chosen astronaut colour.

```
RED = (255, 0, 0)
BLUE = (0, 0, 255)
YELLOW = (255, 255, 0)
```

- As the astronaut is going to be a single pixel, she'll need an `x` and a `y` coordinate, so that the pixel at those coordinates can be illuminated. Near where you have set your colours, set an `x` and `y` position for the astronaut.

```
x = 0
y = 0
```

The player is going to control the astronaut with the Sense HAT's joystick. The joystick can be set up so that whenever it is used, it sends the **event** to a function you have created. Events can be things such as 'pressed up' or 'released right'. To learn how to use the `sensehat` module's joystick events, have a look at the section below.

Triggering function calls with the Sense HAT joystick

The Sense HAT joystick can be used to trigger function calls in response to being moved.

- For instance, you can tell your program to continually 'listen' for a specific event, such as the joystick being pushed up (`direction_up`), and to then trigger a function (called `pushed_up` in this example) in response.

```
sense.stick.direction_up = pushed_up
```

- The function triggered by the event can either have no parameters, or it can take the event as a parameter. In the example below, the event is simply printed out.

```
def pushed_up(event):
    print(event)
```

- This function would print a timestamp of the event, the direction in which the joystick was moved, and the specific action. The output would look similar to this:

```
InputEvent(timestamp=1503565327.399252, direction=u'up', action=u'pre
```

- Another useful example is the `direction_any` method:

```
sense.stick.direction_any = do_thing
```

- If you use this method as seen in the example, the `do_thing` function will be triggered in response to any joystick event. For instance, you could define the `do_thing` function so that it reports the exact event in plain English.

```
def do_thing(event):
    if event.action == 'pressed':
        print('You pressed me')
        if event.direction == 'up':
            print('Up')
        elif event.direction == 'down':
            print('Down')
    elif event.action == 'released':
        print('You released me')
```

- Just above your `while True` loop, add in code for the joystick to use a function (one which you have not yet created):

```
sense.stick.direction_any = draw_astronaut
```

- Now you need to create this `draw_astronaut` function. It will have a single parameter, which is the event. Create the function below one of your other functions.

```
def draw_astronaut(event):
```

- This function is going to need to alter the `x` and `y` variables you set earlier for the astronaut's position. In Python, a function is not normally allowed to alter the value of variables that have been declared outside of the function. To allow your `draw_astronaut` function to do set the `x` and `y` variables, you need to state that `x` and `y` are **global** variables:

```
def draw_astronaut(event):
    global x
    global y
```

- Now you can illuminate the pixel at the `x` and `y` coordinates:

```
def draw_astronaut(event):
    global x
    global y
    sense.set_pixel(x, y, YELLOW)
```

- The pixel will be illuminated as soon as you move the Sense HAT's joystick. In the emulator below, you can try it out using your keyboard's cursor keys.

Moving the astronaut

- You can now program the pixel representing the astronaut to move around the screen in response to the joystick's movements by using **conditional selection**. The basic algorithm inside your `draw_astronaut` function should do the following:
  - When a joystick event is detected:
    - change the colour to `BLUE` to 'hide' the astronaut
    - if the direction is up
      - decrease `y` by `1`
    - if the direction is down
      - increase `y` by `1`
    - if the direction is right
      - increase `x` by `1`
    - if the direction is left
      - decrease `x` by `1`
    - change the colour to `YELLOW` to show the astronaut
- Add code to your `draw_astronaut` function so that the pixel will move around the LED matrix when the joystick is pressed.

I need a hint

- Here's the complete function:

```
def draw_astronaut(event):
    global y
    global x
```

```
        sense.set_pixel(x, y, BLUE)
        if event.action == "pressed":
            if event.direction == "up":
                y -= 1
            elif event.direction == "down":
                y += 1
            elif event.direction == "right":
                x += 1
            elif event.direction == "left":
                x -= 1
        sense.set_pixel(x, y, YELLOW)
```

- You can see it in action here - just use the cursor keys to control the astronaut. You'll notice that you can only see the astronaut when the keys are being pressed.

- To finish off this section, you'll need to display the astronaut within your main game loop.

```
while True:
  matrix = gen_pipes(matrix)
  for i in range(3):
      sense.set_pixels(flatten(matrix))
      matrix = move_pipes(matrix)
      sense.set_pixel(x, y, YELLOW) ##THIS IS THE NEW CODE
      sleep(1)
```

Edge detection

You may notice that if your astronaut drifts off the edge of the screen, your program crashes. Try it out if this hasn't happened to you yet.

This happens because the `sense_hat` module throws an error whenever the `x` or `y` variables go above `7` or below `0`, since there are no LEDs at these coordinates.

You can use a logical operator to help prevent this. For instance, you would only move the astronaut pixel up if the joystick event was `up` **and** the `y` coordinate is greater than `0`.

Have a look at the section below to see how to use Boolean logical operators within your conditional selection.

Boolean operators in Python conditionals

- A standard `if` statement in Python checks for a single condition. For instance:

```
x = 5
if x > 0:
        print('x is greater than zero')
```

- But sometimes you might want to check for more than one condition. In such cases, you can use logical operators in your code.

- The `and` operator checks to see if two conditions have both been met. For instance:

```
x = 5
if x > 0 and x < 10:
```

```
    print('x is between 0 and 10')
```

- So long as `x` is any number within the group - `1,2,3,4,5,6,7,8,9`, then the condition will be true.
- You can use the `or` operator to check if either of the conditions is true.

```
x = 5
if x > 0 or x < 10:
        print('x exists')
```

- In this case, the condition will be true as long as `x` is greater than `0`, or if it is less than `10`.

- Now add some edge detection to your `draw_astronaut` function, so that the pixel coordinate values can't be less than `0` or greater than `7`.

I need a hint

- Here's the whole function:

```
def draw_astronaut(event):
    global y
    global x
    sense.set_pixel(x, y, BLUE)
    if event.action == "pressed":
        if event.direction == "up" and y > 0:
            y -= 1
        elif event.direction == "down" and y < 7:
            y += 1
        elif event.direction == "right" and x < 7:
            x += 1
        elif event.direction == "left" and x > 0:
            x -= 1
    sense.set_pixel(x, y, YELLOW)
```

- Here's an example of the completed code:

Collisions

To finish off the game, you need to ensure that it ends whenever the astronaut collides with one of the pipes.

- Create a new function called `check_collision` that takes the `matrix` as a parameter.

```
def check_collision(matrix):
```

- Now all you need to do is check whether the astronaut's `x, y` position corresponds to a `RED` item in the matrix. If it does, you can return `True`, and if not, return `False`.

I need a hint

- Here's the complete function:

```
def check_collision(matrix):
    if matrix[y][x] == RED:
        return True
    else:
        return False
```

- Now that you are checking for a collision, you can use this conditional in you game loop. It will need to be in both the for loop and the while loop. If it returns `True`, then you can exit the loops.

```
while True:
    matrix = gen_pipes(matrix)
    if check_collision(matrix):
        break
    for i in range(3):
        matrix = move_pipes(matrix)
        sense.set_pixels(flatten(matrix))
        sense.set_pixel(x, y, YELLOW)
        if check_collision(matrix):
            break
        sleep(1)
```

Finishing your game

- There are just a couple of things to do before you're completely done. On the very last line, you should insert code to display a message to indicate that the game is over.

```
sense.show_message('You lose')
```

- Now, when your astronaut collides with the pipes, this message should scroll.
- You may notice a small bug in the program. Sometimes you can fly your astronaut straight through the pipes. This is because the joystick detection is operating outside of the main game loop. The joystick events are not synchronized with the moving columns.
- To fix this, create a new variable called `game_over`, and set it to `False` near where you have set your colour constants. This variable can be used to control when the game ends.

```
game_over = False
```

- Now you can change your `while True` loop so that it becomes a `while not game_over` loop:

```
while not game_over:
    matrix = gen_pipes(matrix)
```

```
    if check_collision(matrix):
        break
    for i in range(3):
        matrix = move_pipes(matrix)
        sense.set_pixels(flatten(matrix))
        sense.set_pixel(x, y, YELLOW)
        if check_collision(matrix):
            break
        sleep(1)
```

- Then you can get rid of those nasty `breaks` by setting `game_over` to `True`.

```
while not game_over:
    matrix = gen_pipes(matrix)
    if check_collision(matrix):
        game_over = True
    for i in range(3):
        matrix = move_pipes(matrix)
        sense.set_pixels(flatten(matrix))
        sense.set_pixel(x, y, YELLOW)
        if check_collision(matrix):
            game_over = True
        sleep(1)
```

- Lastly, add `global game_over` to the draw astronaut function so that, if there is a collision, `game_over` can become `True`and the main game loop will come to an end:

```
def draw_astronaut(event):
    global y
    global x
    global game_over
    sense.set_pixel(x, y, BLUE)
    if event.action == "pressed":
        if event.direction == "up" and y > 0:
            y -= 1
        elif event.direction == "down" and y < 7:
            y += 1
        elif event.direction == "right" and x < 7:
            x += 1
        elif event.direction == "left" and x > 0:
            x -= 1
    sense.set_pixel(x, y, YELLOW)
    if matrix[y][x] == RED:
        game_over = True
```

- Now you should have a finished program, and a game you can play! Have a look at the last step to get some ideas for how you can improve it.

Challenge: taking it further

Here are some ideas for how you could alter your game:

- The for loop alters the spacing between the pipes. Try changing it to see if that makes the game trickier.
- Alter the `sleep` function to make the pipes move faster.
- Add a score, so that for each for loop passed the score increases by 1. Then display the score when the game ends.
- Try making the game a little more like Flappy Bird. Can you make it so that the astronaut constantly falls, but flicking the joystick causes her to move upwards?
