SLUG!

Introduction

Create your own fun and frantic version of the classic game Snake on a Sense HAT: guide the slug around the screen to let her eat vegetables, watch her grow, and increase your score. Don't let her bite into herself though, or it's game over!

What you will make

Use the arrow keys on your keyboard to direct the slug. Click **Run** to start the game.

What you will learn

This project covers elements from the following strands of the Raspberry Pi Digital Making Curriculum (http://rpf.io/curriculum):

- Apply abstraction and decomposition to solve more complex problems (https://curriculum.raspberrypi.org/programming/developer/)

What you will need

Hardware

- A Raspberry Pi
- A 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).)

Software

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

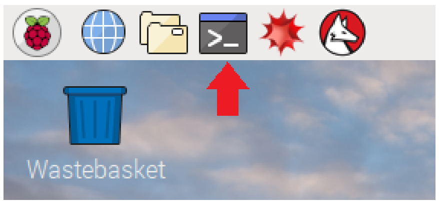
- Python 3
- Sense HAT for Python 3

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
```

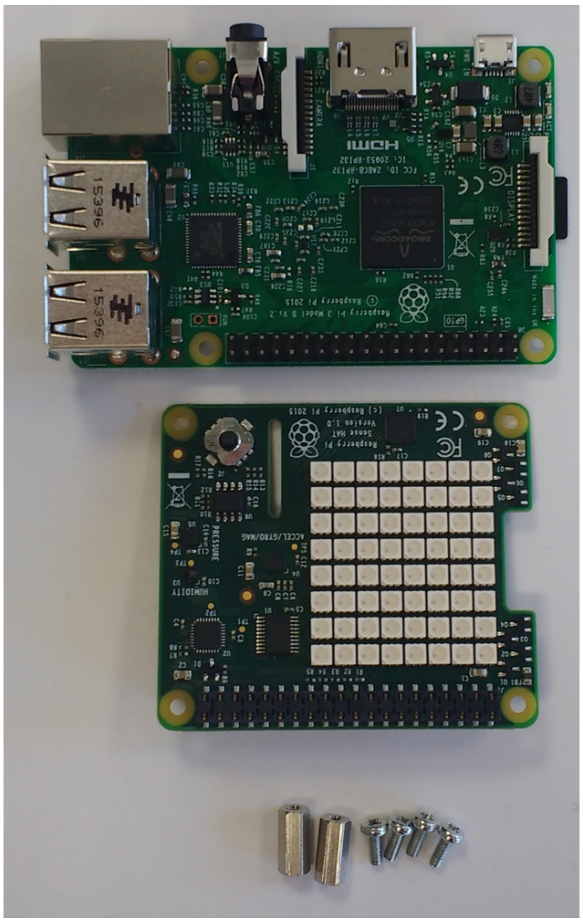
Connect to the Sense HAT

- 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.

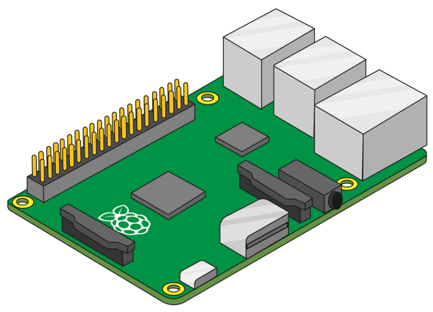
- Remove the Sense HAT and parts from their packaging.



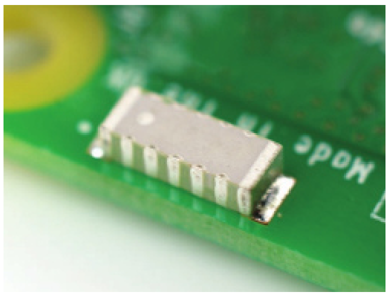
- Use two of the provided screws to attach the spacers to your Raspberry Pi, as shown below.

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.

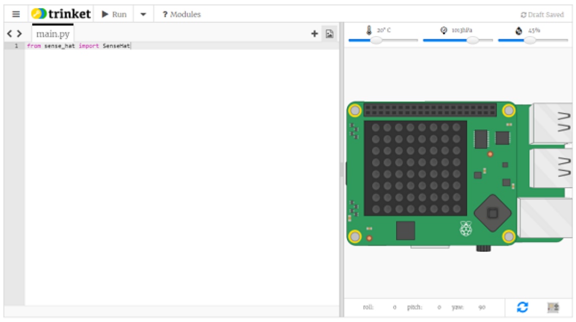
- If you do not have a Sense HAT, you can make this game using the Sense HAT emulator.

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.

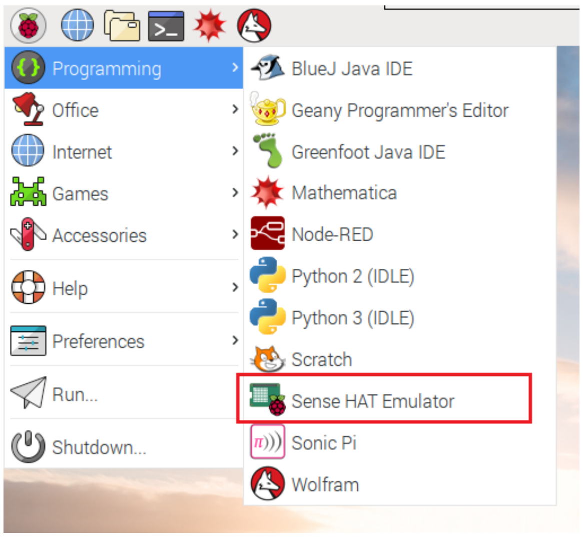


```
from sense_hat import SenseHat
```

- 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.

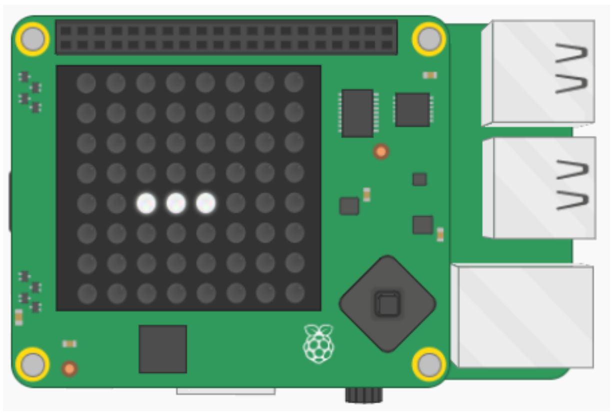
```
from sense_hat import SenseHat
```

- Open up the starter file (https://projects-static.raspberrypi.org/projects/slug/b16828603b87da007b0c8904643384d2101b40a4/en/resources/starter_code.py) if you are using a hardware Sense HAT, or the starter Trinket (https://trinket.io/embed/python/3bfbb0d42b) if you are using the emulator.

This file contains the code to connect to the Sense HAT, imports the modules you will need, and provides a structure for your code.

Draw the slug

Your first job is to draw the slug on the Sense HAT's LED display. It is important to keep track of which pixels the slug is inhabiting so that you can move her around the screen. You will use a 2D list to store the coordinates of the pixels the slug is currently inhabiting.



Two-dimensional lists in Python

A Python list can hold any number of items:

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

A 2D list is a list that holds a number of lists:

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

The lists inside the 2D list can be accessed using an index:

```
>>> board[0]
['X', 'O', 'X']
```

You can also look at a specific item in one of the lists: use the index of the list, and then the index of the item in that list.

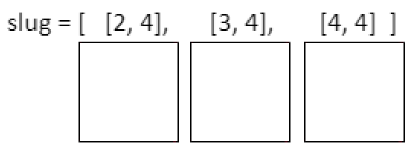
```
>>> board[1][1]
'X'
```

If you want to loop over a 2D list, you can use a **nested** for loop.

```
for row in board:
    for item in row:
            print(item)
```

How will it work?

Your slug will begin by inhabiting three pixels on the LED display. Each pixel has a `x, y` coordinate which can be stored as a list, e.g. `[2, 4]`. The slug will inhabits three pixels, so it needs three coordinate lists. We will store the coordinate lists in another list, making a list of lists or a **2D list**.

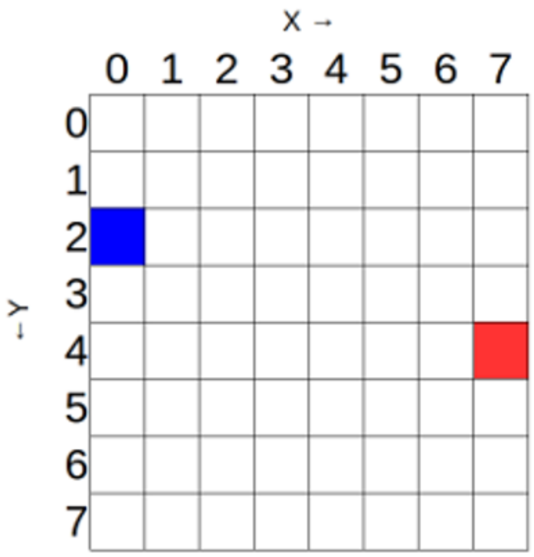


Write the code

To be able to light up a pixel on the LED display, you need to specify three things: the x and y coordinates of the pixel, and the colour you would like the LED to be.

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:

- In the variables section of your file, create an empty list called `slug`.

Create 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 empty list:

```
an_empty_list = []
```

You can create a list by giving it a name and adding the items of data inside square brackets:

```
compass = ["north", "south", "east", "west"]
```

Once a list has been created, you can add more data to the list using `append`:

```
numbers = [5, 10, 15, 20]
numbers.append(25)
print(numbers)

[5, 10, 15, 20, 25]
```

You can get rid of a piece of data from the list using `remove`:

```
numbers.remove(5)
print(numbers)
```

```
[10, 15, 20, 25]
```

You will light up three pixels in a horizontal row to make up the slug. Each pixel's position will be represented as a list containing an x and a y coordinate.

- Add the coordinate lists `[2, 4]`, `[3, 4]`, and `[4, 4]` (in that order) to your `slug` list to define the coordinates where the slug will start out. You have now created a 2D list, or a list of lists!

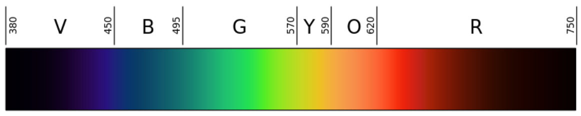
You also need to specify a colour for the slug.

- In the variables section, create a variable to store the RGB colour of your slug. We chose white, but you can choose any colour you like.

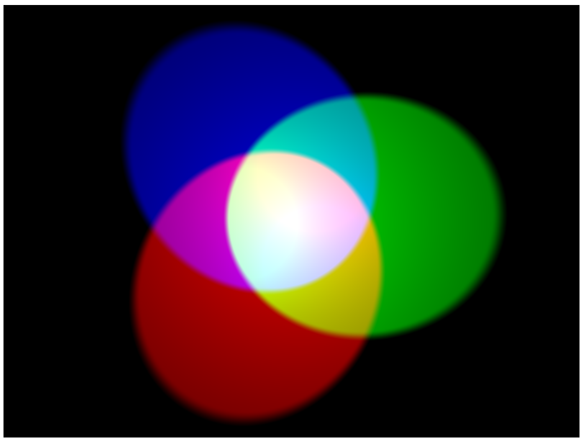
```
white = (255, 255, 255)
```

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).

- In the functions section, create a function called `draw_slug()`. You will put the code to draw the slug into it.

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()
```

- Inside your `draw_slug()` function, add a `for` loop to loop through each element in the `slug` list.

Each element in the list represents the `x, y` coordinates of one segment of the slug.

- Inside the loop, use the `set_pixel` method to light up each pixel you specified in the `slug` list, thus drawing all segments of the slug.

I need a hint

Here is how your code might look:

```
def draw_slug():
  for segment in slug:
      sense.set_pixel(segment[0], segment[1], white)
```

If you run your program at this point, nothing will happen. This is because you haven't called the function, and therefore the code will not execute.

- In the **main program** section, clear the LED screen and then call the function by adding the following code:

```
sense.clear()
draw_slug()
```

- Save and run your program, and check that you see a row of three pixels light up to form your slug.

Move the slug

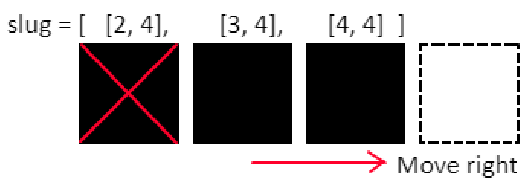
Next, let's make the slug move. The slug should always be moving, but it will only change direction when the player specifies it. Therefore, you need to store the direction in which the slug is moving.

- In the variables section, create a variable called `direction`. The slug will begin the game moving right, so initialise this variable to the string `"right"`.

You also need a way to 'erase' pixels so you can turn off an LED once the slug has moved on.

- Create a variable called `blank`, and set it to the RGB colour `(0, 0, 0)`.

Since you stored the pixel coordinates of the slug's current position in a list, you can now follow this process to move the slug:



- Find the last item in the `slug` list (`[4, 4]`)
- Find the next pixel in the `direction` in the slug is currently moving (`[5, 4]`)
- Add this pixel at the end of the `slug` list
- Set this pixel to the slug's colour
- Set the first pixel in the `slug` list (`[2, 4]`) to `blank`
- Remove this pixel from the list

This algorithm works even when the player changes the direction of the slug. When that happens, the slug's body will simply bend to point in the new direction.

The slug is actually a **queue** data structure.

What is a queue?

A queue is a data structure where the first piece of data added is the first piece of data to come out. It is also called a FIFO or 'first in, first out' data structure. This is like waiting in a supermarket to pay for your shopping: you join the queue at the back, and the person at the front gets to pay for their items first and then leaves the queue.

Imagine the pixels of the slug are bits of food queuing up to be pooped out of the slug. The first item in the list is at the front of the queue, which is the back of the slug: this item will exit the slug and be deleted. New pixels join the slug queue at the end, which is where the mouth of the slug is. They gradually work their way towards the front of the queue as the slug moves.

- In the functions section, create a function called `move()`.

- In the main program section, create an infinite loop which calls this function followed by a `sleep(0.5)`. Once you've written the code for the function, this loop will make the slug continually move around the screen.

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.

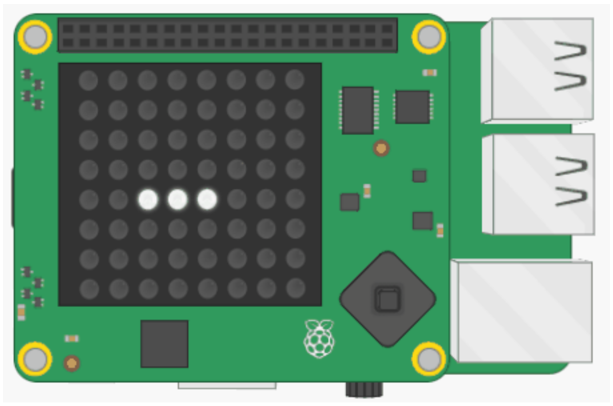
Here is some code to start off the `move()` function. It **does not** work properly yet.

- Copy this code into your function and run the program. We used the colour variable `white` for the slug, so if you chose a different variable name, you will need make sure you're using the right name in the function.

```
def move():
  # Find the last and first items in the slug list
  last = slug[-1]
  first = slug[0]
  next = list(last)     # Create a copy of the last item

  # Find the next pixel in the direction the slug is currently moving
  if direction == "right":

    # Move along the column
    next[0] = last[0] + 1

  # Add this pixel at the end of the slug list
  slug.append(next)

  # Set the new pixel to the slug's colour
  sense.set_pixel(next[0], next[1], white)

  # Set the first pixel in the slug list to blank
  sense.set_pixel(first[0], first[1], blank)

  # Remove the first pixel from the list
  slug.remove(first)
```

- Run the program and look at what happens to the slug. Can you explain why you're seeing what you're seeing?

- Fix the code so that, when the slug reaches the right-hand wall, she 'moves through' the wall and reappears at the same y coordinate but on the opposite side of the screen.



I need a hint

Here is how your code might look, but there are lots of different ways you could successfully write this section:

```
# Move along the column
if last[0] + 1 == 8:
 next[0] = 0
else:
 next[0] = last[0] + 1
```

- Add some more code to make the slug also able to move up, down, and left. This code will be very similar to the code for moving right, but you'll need to work out which coordinate to change and whether to make its value larger or smaller.

I need a hint

Here is how your code might look. Again, there are lots of potential solutions, so your code might look different and work correctly anyway.

```
# Find the next pixel in the direction the slug is currently moving
 if direction == "right":
   if last[0] + 1 == 8:
     next[0] = 0
   else:
     next[0] = last[0] + 1

 elif direction == "left":
   if last[0] - 1 == -1:
     next[0] = 7
   else:
     next[0] = last[0] - 1

 elif direction == "down":
   if last[1] + 1 == 8:
     next[1] = 0
   else:
     next[1] = last[1] + 1

 elif direction == "up":
   if last[1] - 1 == -1:
     next[1] = 7
   else:
     next[1] = last[1] - 1
```

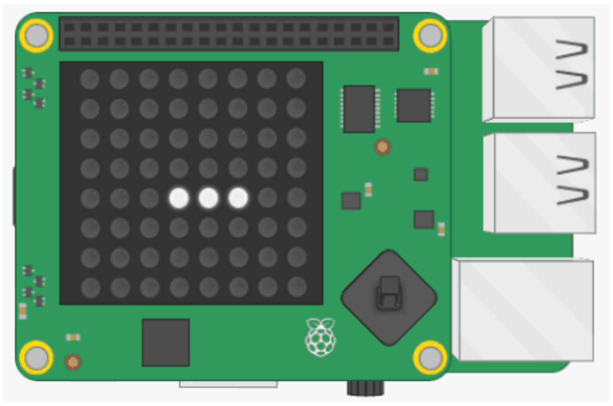
A more efficient way

The code suggested in the previous hint is quite inefficient: there is a lot of repetition. One possible different way of solving this problem would be to first add or subtract from the coordinate value regardless of whether doing so creates a coordinate lying outside the edge of the LED matrix. Then, before performing any actions with the new coordinate, run it through a `wrap()` function to check if it is off the edge and if so, reposition it. Your function might look something like this:

```
def wrap(pix):
    # Wrap x coordinate
    if pix[0] > 7:
        pix[0] = 0
    if pix[0] < 0:
        pix[0] = 7
    # Wrap y coordinate
    if pix[1] < 0:
        pix[1] = 7
    if pix[1] > 7:
        pix[1] = 0

    return pix
```

Use the joystick

Next, let's link up the Sense HAT's joystick so that the player can use it to control the movement of the slug.



- In the functions section, create a new function:

```
def joystick_moved(event):
```

You will call this function whenever the joystick is moved. It will automatically receive a parameter called `event`, which will let you find out in which direction the joystick was moved.

You'll want to set the `direction` variable to the direction in which the joystick was pushed. So that you are allowed to change the value of the variable from within this function, you need to specify `global` for the variable. To find out why, read about scope in Python functions.

Scope in Python functions

Any variable created inside a function is **local** to that function. That is, any code outside of the function won't have access to that variable. Here is a simple function:

```
def say_hello():
    word = "Hello"
    print(word)
```

The variable `word` is local to this function. We also say that the variable is within the function's **scope**. That means you can't use that variable outside of the function. For instance, this code causes an error:

```
def say_hello():
    word = "Hello"
    print(word)

say_hello()
print(word)
```

This is because the variable `word` is outside of the scope of the main program — it only exists within the function.

You can only change the value of a global variable (A **global** variable is a variable declared outside of a function, i.e. within the main program.) within a function if you tell the function that the variable is global. In the example below, changing the value of the variable `word` inside the function has no effect on the variable `word` in the main program. A completely separate copy of the variable has been created and altered within the function:

```
word = "Goodbye"

def say_hello():
    word = "Hello"
    print("The function thinks that the variable word is:" + word)

say_hello()
print("The main program thinks the variable word is:" + word)
```

However, if you declare the variable `word` to be global within the function, the function will edit the copy of the variable in the main program:

```
word = "Goodbye"

def say_hello():
    global word
    word = "Hello"
    print("The function thinks that the variable word is:" + word)

say_hello()
print("The main program thinks the variable word is:" + word)
```

- Add `global direction` to your function:

```
def joystick_moved(event):
    global direction
```

You can access the direction the joystick was moved in with the help of the `event` parameter: use the command `event.direction`.

- Inside your function, set the `direction` variable to be equal to `event.direction`.

- Finally, in the main part of your program, write a line of code to say that, when the Sense HAT joystick is pressed in any direction, the `joystick_moved` function will be called.

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'pressed')
```

- 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')
```

I need a hint

Here is how the code in the **main program** should look:

```
sense.stick.direction_any = joystick_moved
```

Here is the full **function** code:

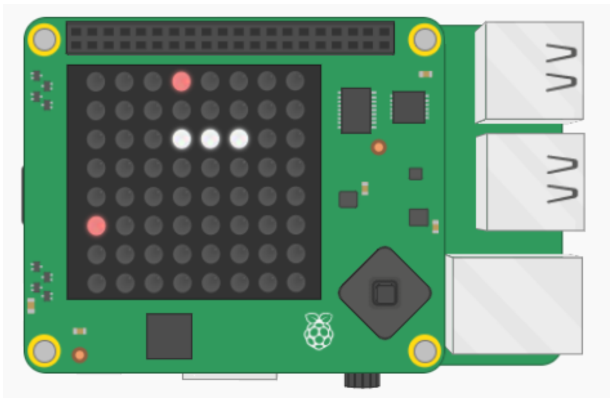
```
def joystick_moved(event):
    global direction
    direction = event.direction
```

- Run your program and test that it works. If you are using the emulator, you can simulate moving the joystick by pressing the arrow keys on your keyboard.

At this point it is possible to move the slug back “through” herself, which looks rather odd. Later on we will add some code which causes the game to end if the slug bites herself, so there is no need to worry about resolving this glitch.

Create vegetables

Our slug is hungry, so she needs something to eat! Let’s generate some vegetables for her at random locations on the LED matrix.



Creating the vegetables is fairly straightforward:

1. Pick a `x, y` random coordinate on the LED matrix
2. Check if this coordinate is currently inhabited by the slug
3. If it is, repeat steps 1 and 2 until you pick a location that is outside the slug
4. Draw the vegetable on the LED matrix

The code you need is very similar code you’ve written earlier for the slug, so try to do this bit by yourself. If you get stuck, use the hints.

- Create a new variable to define the colour of the vegetables you’re going to make. You can do this in the same way you defined the colour of your slug.

Create the function

- Define a new function called `make_veg()` in your functions section. The code to put inside the function is explained in the following steps.

- Inside the function, write some code to pick a random coordinate on the LED matrix.

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)
```

I need a hint

Here is how your code should look:

```
x = randint(0, 7)
y = randint(0, 7)
new = [x, y]
```

- Check if this `x, y` coordinate is in the `slug` list. If it is, pick a new coordinate and check it against the list. Repeat this until the coordinate you've picked isn't in the slug list.

Checking whether an item is in a list using Python

Checking whether an item is in a list is easy in Python: you can use the `in` keyword like in the example here. If the item is in the list, the message "Found it!" will be printed.

```
search_item = "paper"
items = ["rock", "paper", "scissors"]

if search_item in items:
    print("Found it!")
```

You can also use a loop to continue executing a bit of code until an item is in a given list. This is useful for validating input. Below we are using the opposite of the `in` keyword: `not in`.

```
direction = ""
while direction not in ["N", "S", "E", "W"]:
    direction = input("Please enter a direction (N, S, E or W): ")
print(direction)
```

This loop will keep asking for a direction until the user enters one that is in the list. Once they have entered one of the directions given as options, that direction will be printed.

I need a hint

Here is how your code might look:

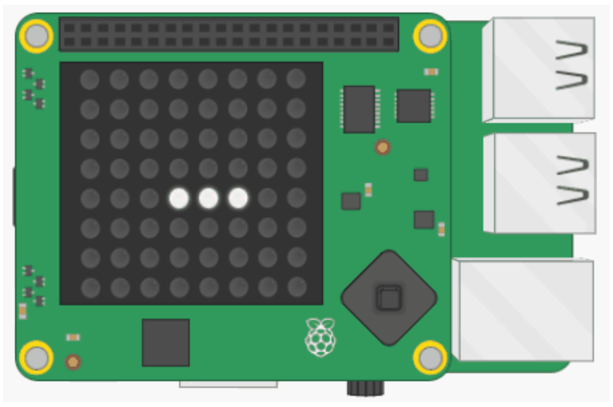
```
new = slug[0]
while new in slug:
    x = randint(0, 7)
    y = randint(0, 7)
    new = [x, y]
```

- Once you have found an `x, y` coordinate which isn't inside the slug, draw the vegetable on the screen using your new colour variable.

Call the function

- In your main program, call the `make_veg` function and check that vegetables randomly appear on the LED matrix.

You will probably notice that rather a lot of vegetables appear, so your slug is quickly overrun!



You need a way to track how many vegetables there are, so that you can prevent this dangerous spreading of veggies!

Keep track of the vegetables

- Create a new empty list called **vegetables** in your variables section.
- Write a line of code at the end of your **make_veg** function to add the coordinates of the new vegetable to your **vegetables** list.

Adding data to a list in Python

Python lists are mutable. This means data can be added or removed from them. Let's try this out!

- Start by creating an empty list.

```
my_list = []
```

- By using the keyword **append**, you can add any data you want to the list.

```
my_list.append('A string')
```

- This will produce a list that looks like this:

```
['A string']
```

- Lists can hold data of any type:

```
my_list.append(1)
my_list.append(['another', 'list'])
my_list.append(('a', 'tuple'))
```

- The final result of all these operations would be:

```
['A string', 1, ['another', 'list'], ('a', 'tuple')]
```

- Change the way you call the **make_veg** function in the main program so that it will only create a new vegetable if there are fewer than three items in the **vegetables** list.

I need a hint

Here is how your code should look:

```
if len(vegetables) < 3:
    make_veg()
```

Challenge

Can you change your code so that, if there are fewer than 3 vegetables in the list, there is only a 20% chance of creating a new vegetable each time the function runs? This will make it less predictable when vegetables might appear. To create the 20% chance, randomly pick a number between 1 and 5, and only create a vegetable for one specific number in this range.

Challenge solution

```
# Let there be a 20% chance of making a veggie if there aren't many about
if len(vegetables) < 3 and randint(1, 5) > 4:
    make_veg()
```

Eating vegetables

Your slug already appears to eat the vegetables — great! However, you have probably found that, once she has eaten the first three vegetables, no more vegetables ever appear regardless of how long you wait.

Can you work out why this is?

Answer

Since the `make_veg` function only generates vegetables if the `vegetables` list contains fewer than three items, no new vegetables will appear once the list contains three vegetables.

At the moment, your first three vegetables are generated and added to the `vegetables` list. When the slug moves to a pixel containing a vegetable, the vegetable disappears because the `move` function draws the slug's pixels over the top of it. This is why the slug appears to eat it. However, the vegetable is never removed from the `vegetables` list, so the `make_veg` function isn't allowed to make more.

- Add some code at the end of the `move()` function so that, whenever the slug moves to a new pixel, the function checks whether that pixel is in the `vegetables` list. If the pixel is in the `vegetables` list, remove it from the list.

Let's also add a score to keep track of how many vegetables the slug has eaten, so that the player knows how well they have done when their round ends.

- In the variables section, create a `score` variable which starts as `0`.

- Whenever the slug eats a vegetable, add 1 to the score. Don't forget that, because the `score` variable was created outside of the function, you need to specify that you want to use `global score` at the start of the `move()` function so that the function is allowed to change it.

I need a hint

Here is the code you should add to the end of the `move()` function:

```
if next in vegetables:
  vegetables.remove(next)
  score += 1
```

Don't forget to also add `global score` on the first line of the `move()` function, and to initialise the `score` variable to `0` in the variables section.

Growing and speeding up

To keep the game interesting for the player, as the slug eats vegetables she should grow in length and move faster. This makes her harder to control — and worse, she might accidentally bite into her own body!

The code you've written so far makes it so that, when the slug advances a pixel, her end segment is deleted. To enable her to grow, you can add a new segment to the `slug` list but **not** delete the last one. So each time the slug moves, you need to make a decision as to whether to remove a segment or not.

- At the start of the `move()` function, create a Boolean variable called `remove` and set it to `True`, as most of the time we do want to remove the end segment when the slug moves.

- Add a conditional statement to your function so that the following two lines of code are only executed `if` the `remove` variable is `True`.

```
sense.set_pixel(first[0], first[1], blank)
slug.remove(first)
```

- Decide how often you want your slug to grow. In our example, we chose for our slug to grow one segment for every 5 vegetables she eats.

To help your program decide when to trigger the slug to grow, you can use the **mod** operator `%`. By using it, you can work out whether the current score is a multiple of 5.

Using the mod operator in Python

The modulo or mod operator divides one number by another and returns the remainder. The symbol of this operator is `%`.

```
remainder = 5 % 2
print(remainder)

>> 1
```

The calculation `5 / 2` is performed, and the remainder is returned. In this case, 2 goes into 5 twice, with 1 left over as the remainder. Hence, the result is 1.

You can use mod to check whether a number is a multiple of another number. If it is, the remainder will be 0, like in this example:

```
test_number = 50
if test_number % 5 == 0:
    print("This is a multiple of 5")
```

You'll want to ensure that the slug can only grow when it has eaten a vegetable. To do this, you should add your new code to potentially grow the slug inside this `if` statement:

```
if next in vegetables:
```

- Add code to check whether the current score is a multiple of 5. If it is, set `remove` to `False` so no segment gets removed.
- Save and run the code. You might be disappointed to see that, when you eat 5 vegetables, the slug doesn't grow! Why not?

Answer

The code to remove the segment currently runs before the code to check whether the new pixel was a vegetable. This means that the program doesn't care whether we've told it to remove or not remove a segment, because by the time this decision is taken, the segment has already been removed!

- Alter the order of the code so that the last thing in the `move()` function is the code to potentially remove a segment. Test again, and you should see your slug grow.

As well as setting `remove` to `False` to grow the slug, let's speed it up! Currently, the slug's speed is regulated by this line of code in the main program:

```
sleep(0.5)
```

This command tells your infinite loop to wait for half a second between each run, so that the slug moves at a speed of one pixel per half-second. To make it move faster, you'll want to gradually decrease this pause.

- In the variables section, create a variable called `pause` and set it to `0.5`.
- Replace the `0.5` in the brackets of the `sleep()` function with the variable `pause`. Now the speed of the slug will be determined by the value of `pause`.

You've already added some code to section which check whether the current score is a multiple of 5, so your slug can grow. That is also the point at which the value of `pause` should become smaller. Amend your `move` function like this:

```
global pause

if score % 5 == 0:
    remove = False
    pause = pause - 0.1
```

- Save and run your program. What happens? Why is this code not a very good idea, and how could you improve it?

Answer

If `pause` begins at `0.5`, and we subtract `0.1` for every five eaten vegetables, `pause` will eventually become `0`. That will make the game impossible!

Instead of subtracting a fixed amount, why not make `pause` proportionally smaller? For example:

```
pause = pause * 0.8
```

This will make `pause` 80% of its previous value. In this way, it will never become `0`.

Game over

The final thing you need to check when the slug moves to a new pixel is whether she will bite into her own body. If this happens, she will die and the player will lose the game. The way to do this is very similar to how you checked checked whether the slug is eating a vegetable.

- In the variables section, create a new Boolean variable called `dead` and initialise it to `False` (so the slug doesn't start off dead!).
- Inside the `move()` function, add some code to check whether the next pixel the slug is moving to is part of herself.
- If it is, set the `dead` variable to `True`. Do you remember what you need to do to allow a function to alter a global variable?

I need a hint

Here is how your code should look. Make sure you add it to the `move()` function **above** the code which adds the `next` pixel to the `slug` list, otherwise you will be permanently dead!

```
if next in slug:
    dead = True
```

Save and run your code. When you make the slug bite itself now, you'll see that the game doesn't end, even though the `dead` variable is changed to `True`!

- Change the game loop in the main program from an infinite loop to a loop which only runs while the `dead` variable is not `True`.

Once the game loop ends, the player will want to know how they did in their game.

- In the main program, add code to display how many vegetables the player guided the slug to by making use of the `score` variable.

Show a message on the Sense HAT

Make sure you have the following lines of code in your Python program to set up your connection with the Sense HAT. There is no need to add them more than once.

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

- Add this code to display a message on the Sense HAT's LED matrix.

```
sense.show_message("Hello world")
```

The message "Hello world" will now scroll across the LED screen.

- Change the words in the quotes (`""`) to see a different message.

You can try it out here:

Challenge: crazy veggies

There are lots of possible features you could add to your game to make it more difficult and entertaining for the player! Here are a few examples – can you come up with your own ideas?

- Growth vegetables: add a new type of vegetable which causes the slug to grow by more than one segment at a time. This makes the game harder because the slug will become big quickly, making it difficult to stop her from biting herself!
- Slug trails: leave a trail behind your slug in a different colour so you can see where you've been. Perhaps you could then make the game harder by not allowing vegetables to be generated in spaces where there is a slug trail?
- Digest the veg: add code so that, when the slug eats a vegetable, you can see it working its way through the slug's digestive system and getting pooped out of the end! Perhaps you could make the game also end if the slug moves to a space with a pooped-out vegetable? Yuck!