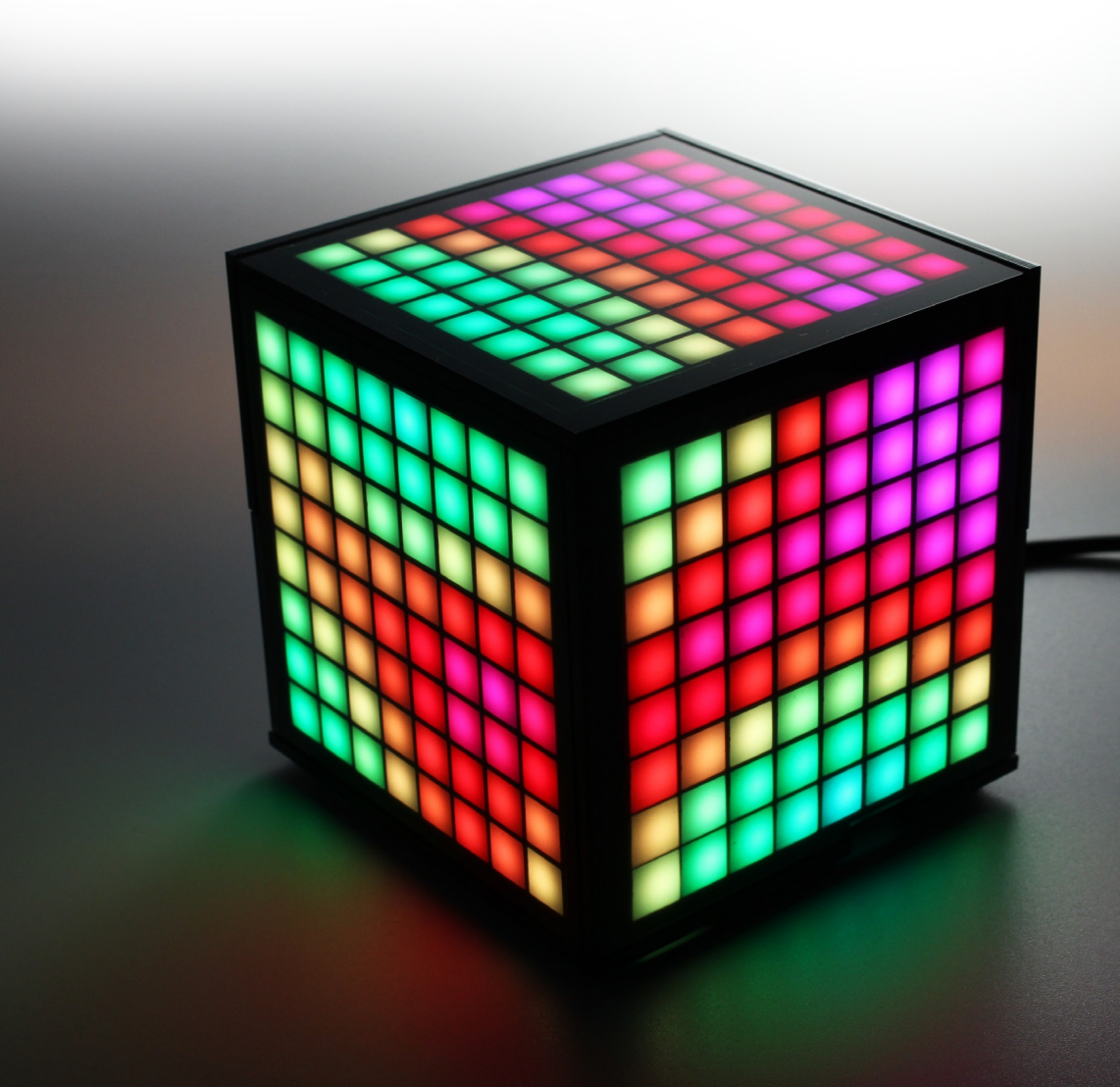
ABSTRACT FOUNDRY

LUMICUBE

Project Booklet



ATTENTION

• THIS PRODUCT IS NOT A TOY AND REQUIRES ADULT SUPERVISION

• FOR INDOOR USE ONLY – DO NOT GET WET

• DISPOSE OF ONLY AT AN ELECTRICAL AND ELECTRONIC EQUIPMENT RECYCLING FACILITY

An online version of this manual can be found at:
https://abstractfoundry.com/lumicube/manual.pdf

Image credits:
Equaliser - Chaiwut Sridara
Sun - Dimitris Vetsikas
Thunderstorm - Felix Mittermeier
Rollercoaster - Pixabay

Abstract Foundry Limited,
London, UK (11956163)
Made in China
5.0 V ⎓ 3.0 A

Contents

What's In The Box

Advanced kit only

Cable board

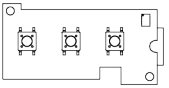
Environment sensor

IMU

LCD Screen

3 x button tops

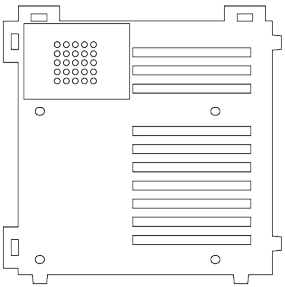
Button board



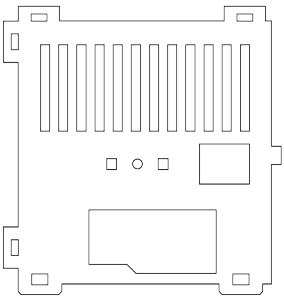
8 pin cable

2 x FFC cable

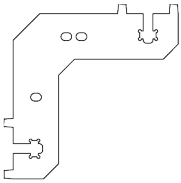
Bottom face



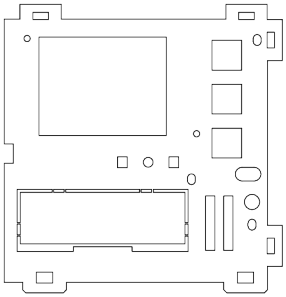
Back face



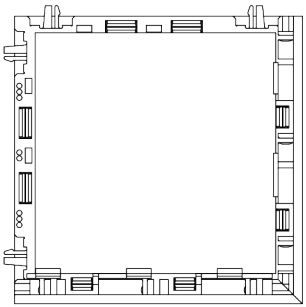
Bracket



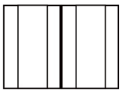
Screen face (adv. kit)
or Side face (starter kit)



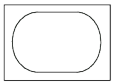
3 x LED panel



3 x Clip On Ferrite

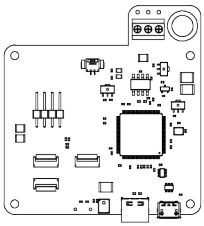


Speaker



4 pin cable

Base board



	M2	M2.5	M3
Nut	2 x	14 x	2 x
12mm Screw	2 x	7 x	2 x
6mm Screw		3 x	
6mm Standoff		7 x	
Washer		4 x	

Pre-Assembly

Setup your Raspberry Pi

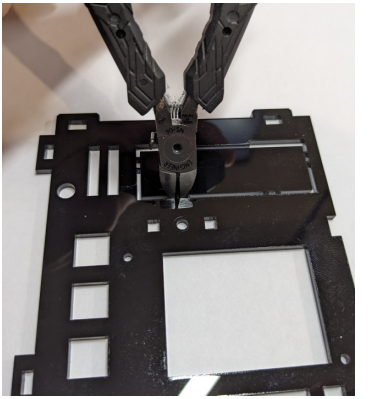
Follow the instructions on how to setup your Raspberry Pi here:
https://projects.raspberrypi.org/en/projects/raspberry-pi-setting-up/

Make sure the Raspberry Pi is setup and connected to the WiFi before assembly.

Install the LumiCube software

Follow the installation instructions on our website:
www.abstractfoundry.com/lumicube/resources

Peel the film off all acrylic pieces

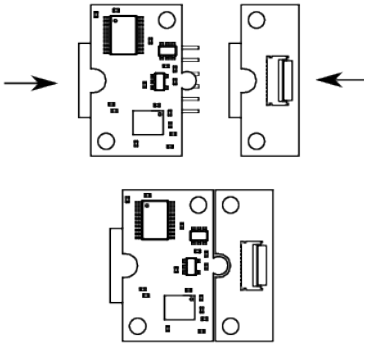


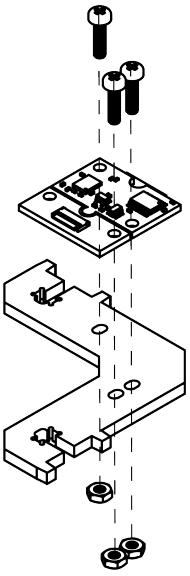
Cut away port panel (optional)

If you have the bigger Raspberry Pi 3B or 4B break off the port panel using either some snips or just a flat head screw driver. Just insert the screw driver into the gap and twist to break off the panel.

Advanced Kit Assembly

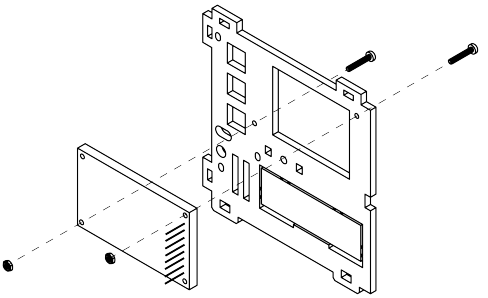
A1) Plug together the cable board and the IMU board.





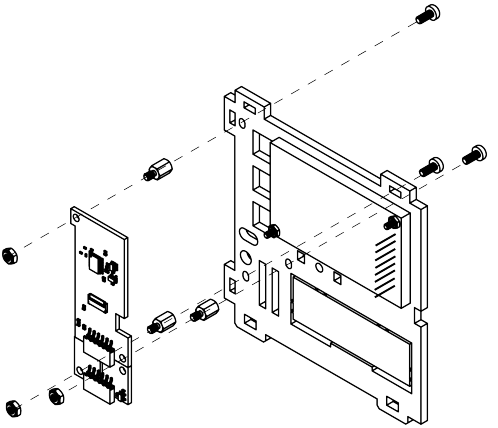
A2) Using three M2.5 screws and nuts bolt the IMU and cable board to the L-shaped bracket piece.

A3) Peel the protective film off the screen and using two M2 screws and nuts, bolt the screen to the screen face.

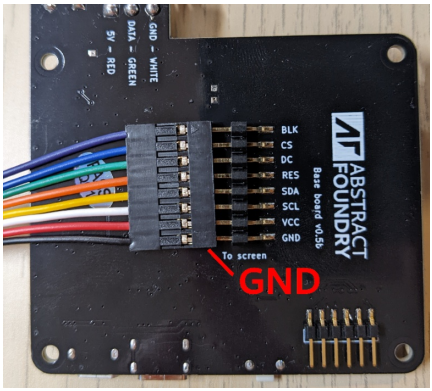


A4) Push the 3 button tops on to the button board

A5) Plug together the environment sensor and the button board (the same way as used for the IMU and cable board). Using three M2.5 standoffs, 6mm screws and nuts bolt them to the screen face.

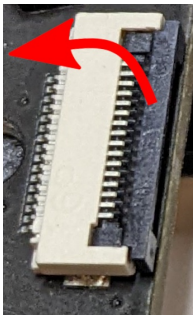


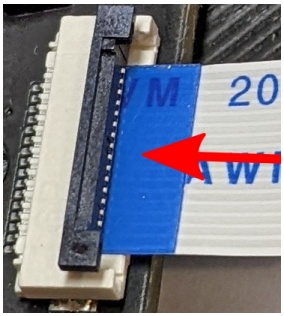
A6) Plug the 8 pin cable into the back of the base board and into the screen:

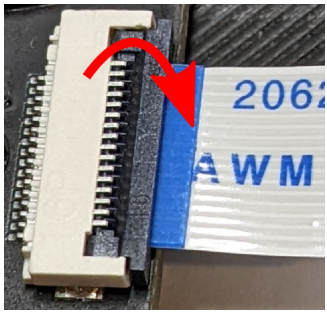


Make sure the pins on the board connect to the correct pins on the screen.
GND -> GND

A7) Connect two FFC cables to any 2 of the 3 base board connectors.



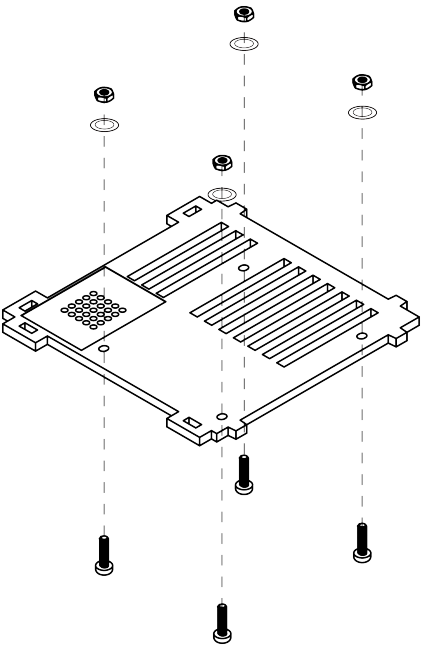




Make sure the blue side of the FFC cables are facing up and they are inserted straight.

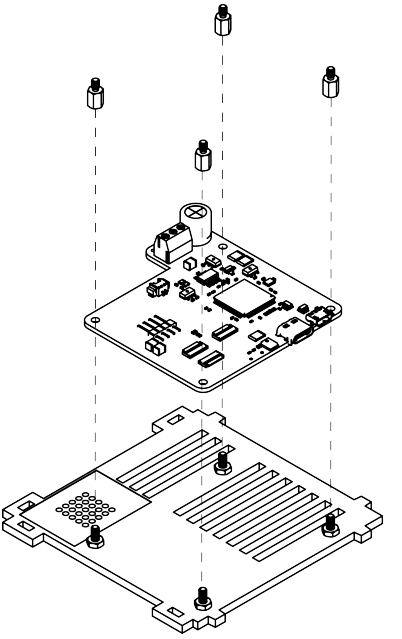
A8) Connect the other end of the FFC cables to the cable board and button board.

Main Assembly



1) Place 4 M2.5 screws up through the bottom face and screw on a washer and nut to hold them in place.

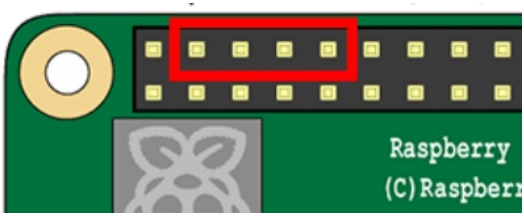
Make sure the bottom face is the right way up.

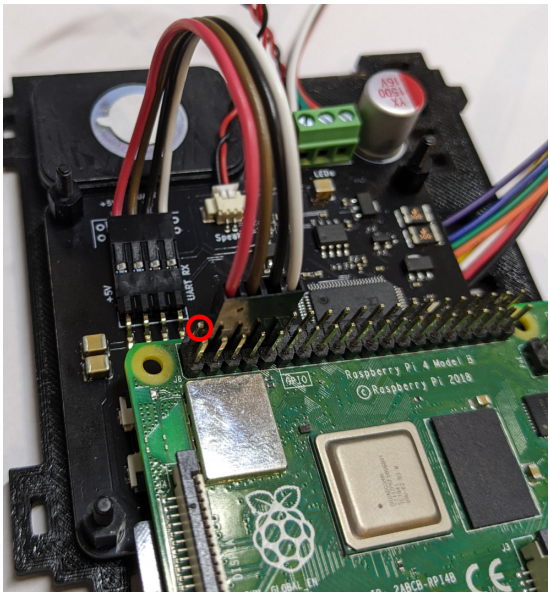


2) Place the base board on top and screw in four M2.5 standoffs.

3) Plug in the speaker. Peel the film off the front and stick it in the square marked on the bottom face with the cable pointing away from the edge.

4) Plug one end of the 4 pin cable into the header on the top left of the base board. Plug the other end into the Raspberry Pi header pins marked in the diagram.



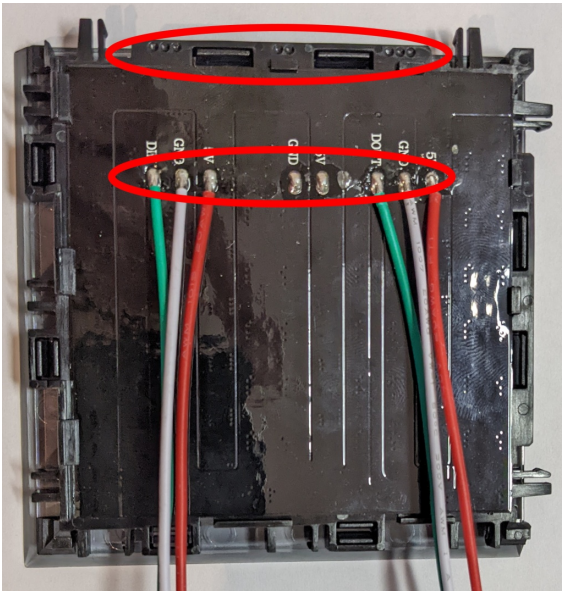


Make sure the cable is not twisted!

5) Place the Raspberry Pi on top of the base board (on the standoffs) and screw on four M2.5 nuts to keep it in place.

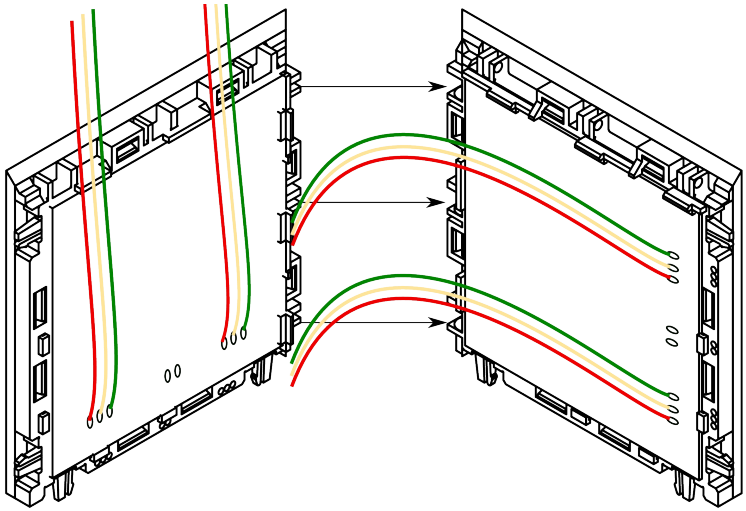
LED Panels

If at any point the LEDs come off the panel follow these instructions to put them back in. First make sure the solder joints of the LED panel are on the same side as the dots on plastic panel (marked in red). Bend the LED panel slightly and slide it under the catches on the left side. Then slide it under the catches on the right side. Then push it up into the catches at the top. Finally press down to make sure the LED panel is flat.

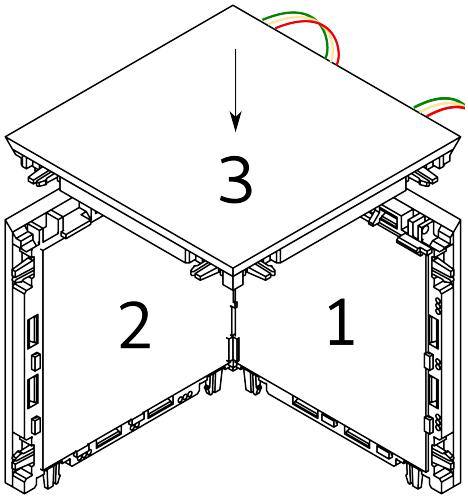


6) On each of the 3 panels snap a clip on ferrite around the 3 wires of the male connector (the smaller one).

7) Push any 2 LED panels as shown until they snap together, folding the cables out of the way.



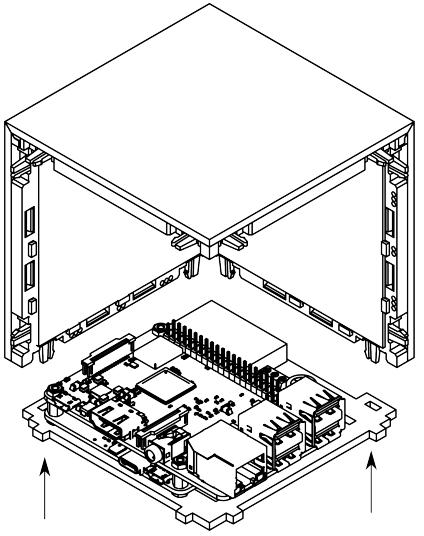
8) Push the third panel on to the top. Folding the cables out of the way.



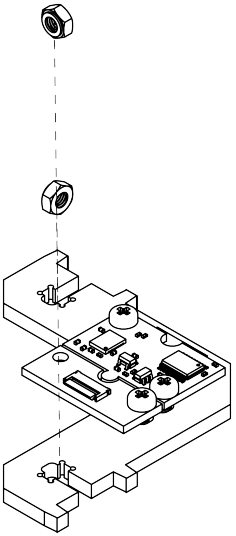
9) Connect up the wires of the LED panels. Start by inserting the male connector (the smaller one) of panel 3 into panel 2's female connector. Then connect panel 2's male to panel 1's female. Finally plug panel 1's male connector into the cable coming off the base board.

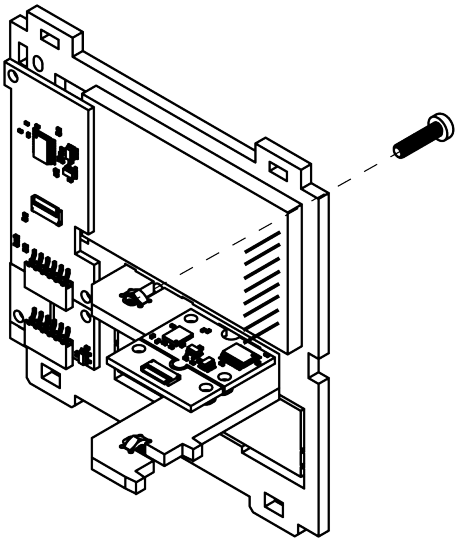
10) Push the bottom face onto the 3 panels, making sure none of the wires are caught.

Note: for clarity we haven't shown the wires, but it will be quite packed in the cube.

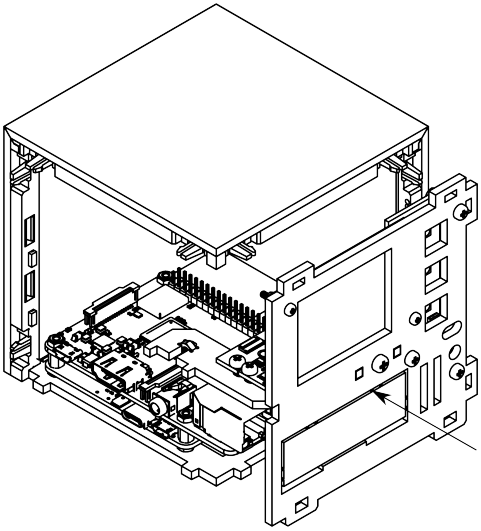


11) Place two M3 nuts in the cutouts on the L shaped bracket. Try different angles if the nut does not go in at first. The nut should be kept in place with friction. If it's too loose try a different angle or try sticking it there temporarily with some tape. Then place the bracket in to the slot on the screen/side face and screw in an M3 screw to bolt it in place.

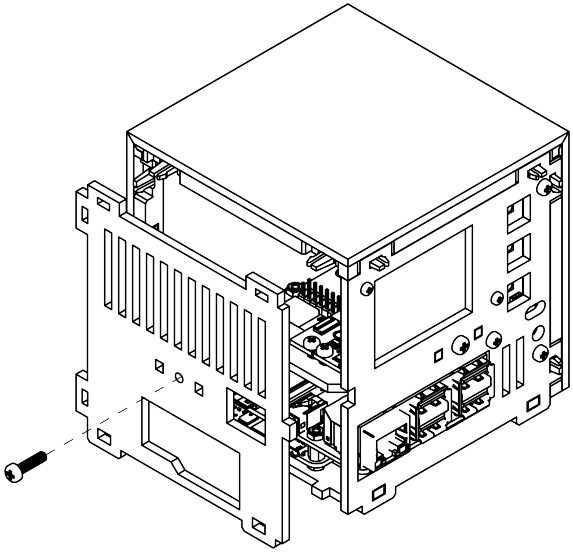




12) Push the screen/side face on to the cube.



13) Now push the back face onto the cube. You may have to wriggle the bracket around to get it into the right place. Once it is on screw in the M3 screw.



Congratulations! You have built your LumiCube!

Using the LumiCube

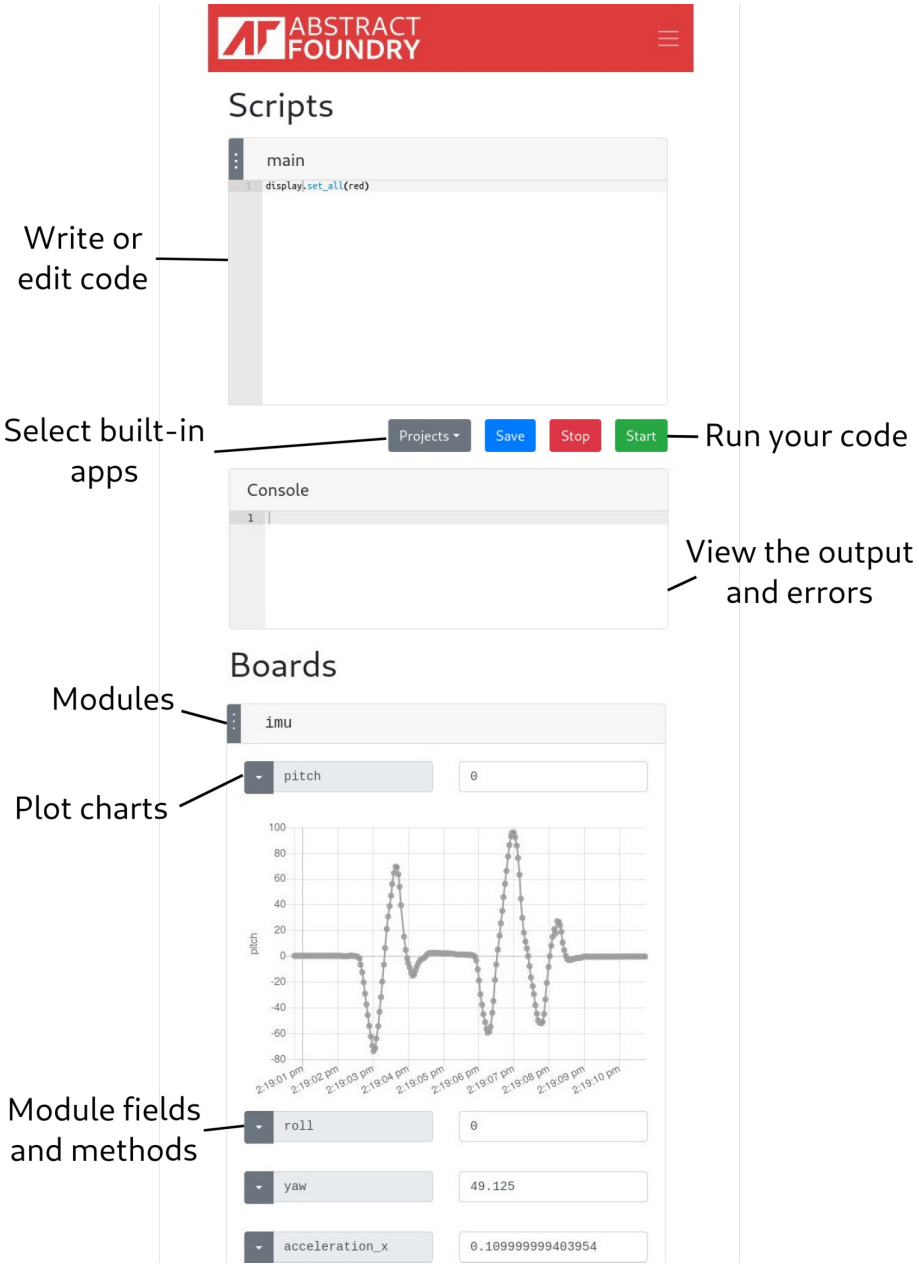
Plug in the power supply to the base board (not the Raspberry Pi)

All the LEDs should light up cyan. After 30 seconds the Raspberry Pi should have booted and started running the LumiCube software. At this point the LEDs will change to magenta to show the LumiCube is up and running.

To connect to it you need the Raspberry Pi's IP address. To find out the IP address you can hold down the small button near the power port at the back and it will display on the LEDs. Or if you have a screen it will be displayed there.

To program the LumiCube just open a browser on your computer or phone and in the address bar type the IP address of the cube. For example:

The browser should show the dashboard. Try selecting one of the built in projects and running it!


ABSTRACT FOUNDRY
Scripts
main
display.set_all(red)
Write or edit code
Select built-in apps
Projects
Save
Stop
Start
Run your code
Console
View the output and errors
Boards
Modules
imu
Plot charts
pitch
0
100
80
60
40
20
0
-20
-40
-60
-80
2:19:01 pm
2:19:02 pm
2:19:03 pm
2:19:04 pm
2:19:05 pm
2:19:06 pm
2:19:07 pm
2:19:08 pm
2:19:09 pm
2:19:10 pm
Module fields and methods
roll
0
yaw
49.125
acceleration_x
0.109999999403954

Python Cheat Sheet

Check out our website for links to Python tutorials and other resources:
www.abstractfoundry.com/lumicube/resources

Operators

```
Comments: #
Logic operators: and or not > < <= >= == !=
Maths operators: + - * /  % // **
```

Lists

```
items = [red, orange, green]  # Create a list
items.append(blue)       # Add to the end
items.insert(2, yellow)  # Add yellow after orange
(position 2)
items.remove(green)      # Remove element
colour = items.pop(1)    # Remove orange (position 1)
sub_list = items[0:2]    # Take the elements 0 to 1
length = len(items)
```

Dictionaries

```
data = {"name" : "Billy", "age" : 24} # Create a
dictionary
data["surname"] = "Bob"             # Add to dictionary
age = data["age"]                 # Get value
```

```
keys = data.keys()                  # Get keys list
entries = data.items()              # Get key value pairs
list
```

Loops

```
while a:
    # loop until a equals False

for i in range(2, 10):
    # loop with i set to 2, 3, 4 ... 9

for element in list:
    # loop over each entry in the list

for key, value in dictionary.items():
    # loop over each key and value in the dictionary
```

If

```
if a == b:
    # if a equals b
elif a > b:
    # else if a is greater than b
else:
    # else
```

Functions

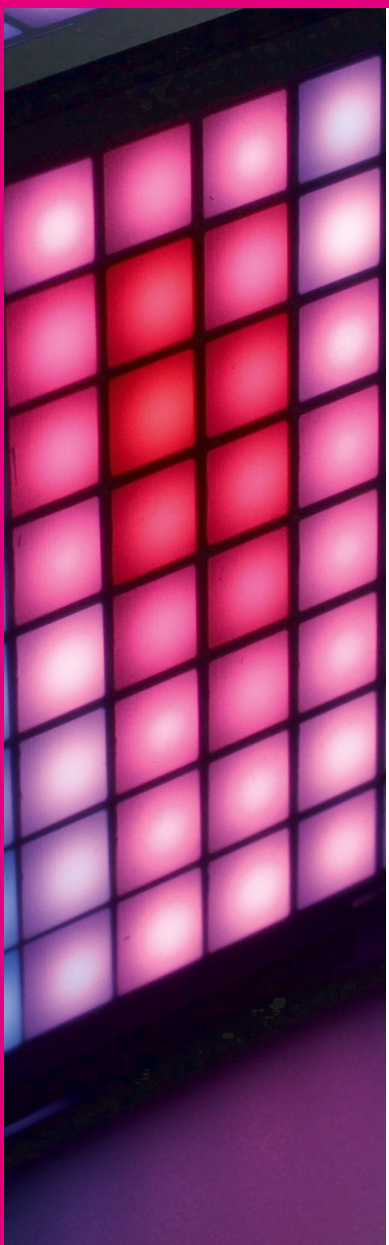
```
def my_function(argument1, argument2):
    # Do something
    return True
```

Common gotcha's

1) If checking a and b are equal make sure you use a == b not a =b.

2) Python is very picky about white space. Make sure you use 4 spaces instead of tabs.

LEDs



API

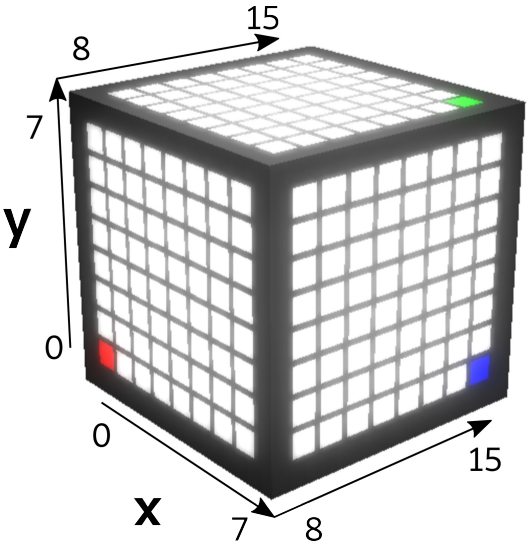
```
display.brightness = value

display.set_all(colour)

display.set_led(x, y, colour)

display.set_leds(x_y_to_colour_dict)

display.set_3d(x_y_z_to_colour_dict)

display.set_panel(panel,
2d_colour_list)

display.scroll_text(
       text,
       colour = white,
       background_colour = black,
       speed = 1
)
```

Set all the LEDs!

```
display.set_all(white)
```

Set a few LEDs

```
display.set_led(0, 0, red)
display.set_led(15, 0, blue)
display.set_led(7, 15, green)
```



Did it do what you expected? We use 2D coordinates where the x coordinate goes from left to right and y goes from bottom to top.

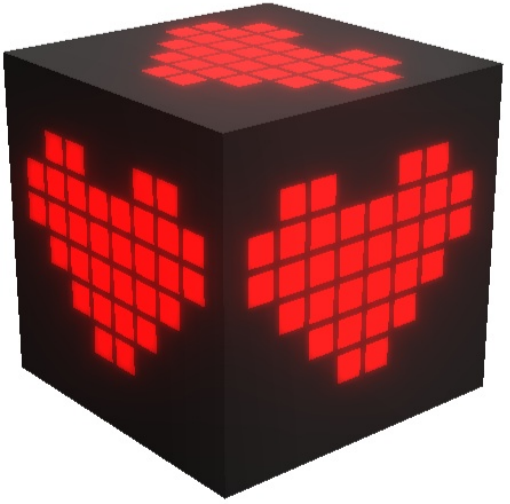
This forms a sort of L shape, but folded on to a cube.

Note: the cube can't handle all the LEDs at maximum brightness so by default the brightness is 50. If you increase the brightness too much it will "trip" and set the brightness back down to 5. You can see the current used by the display (estimated_current) and you can set the trip current (max_current).

Draw pixel art

The set_panel method allows you to easily draw pixel art on one of the 3 LED panels. You pass it a list of lists of colours as shown below, along with the name of the panel e.g. "left", "right", or "top".

```
r = red
heart = [
    [0,0,0,0,0,0,0,0],
    [0,r,r,0,0,r,r,0],
    [r,r,r,r,r,r,r,r],
    [r,r,r,r,r,r,r,r],
    [0,r,r,r,r,r,r,0],
    [0,0,r,r,r,r,0,0],
    [0,0,0,r,r,0,0,0],
    [0,0,0,0,0,0,0,0],
]
display.set_panel("left", heart)
display.set_panel("right", heart)
display.set_panel("top", heart)
```



Colours

There are 3 ways to specify an LED colour in our system:

1) Hex colours - e.g. 0xFF8C00 (orange)
A 24 bit value, made of 8 bits of red, green and blue. You can search online for the hex code for a colour.

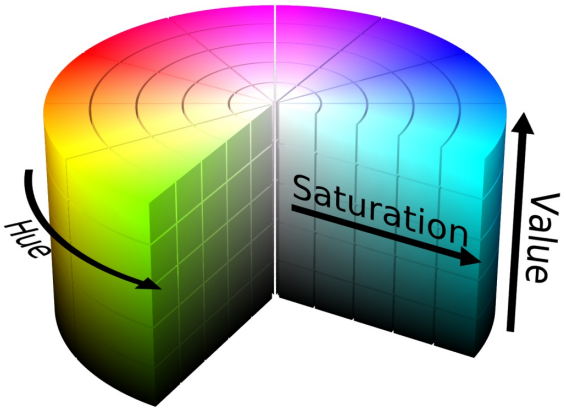
```
display.set_all(0xFF8C00)
```

2) Predefined colours - e.g. red, orange, yellow
These are some variables we've added to make life easier.

```
display.set_all(red)
```

3) HSV colours - hsv_colour(hue, saturation, value)
HSV is a neat standard for defining colours, see the diagram below. Hue determines where along a rainbow the colour is. 0.0 is red, 0.15 is yellow, 0.3 is green, 0.6 is blue, etc. Saturation is how intense the colour should be, where 1.0 means the full colour and 0.0 means full white. Value is the brightness, where 0.0 is black and 1.0 full brightness. Try it out:

```
display.set_all(hsv_colour(0.3, 1, 1))
```



Setting multiple LEDs at once

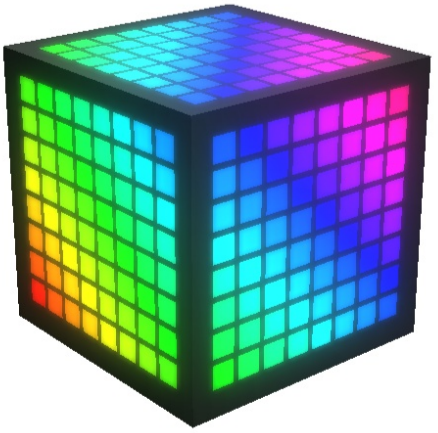
set_leds is similar to set_led but instead takes a dictionary with the (x, y) coordinate as a key and the colour as a value.

```
leds = {}
leds[2, 2] = yellow
leds[3, 3] = green
display.set_leds(leds)
```

Using set_leds() is much faster than calling set_led() multiple times. It also means the leds update all at the same time, leading to nicer animations.

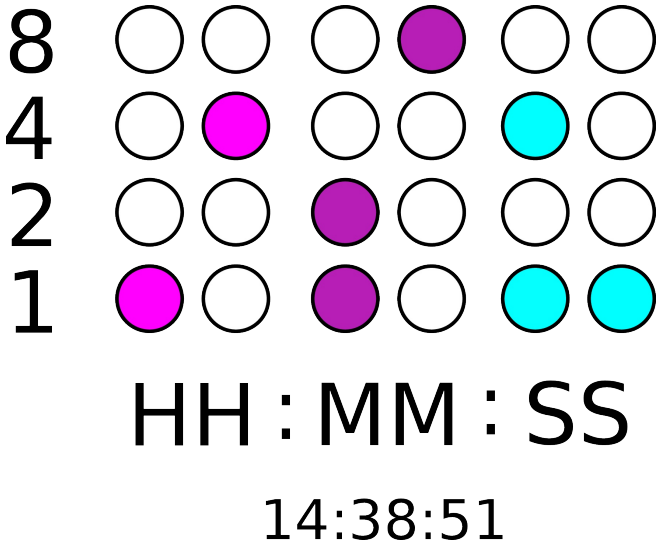
See if you can understand the following code. Looping over x and y creates a square of coordinates. The if statement then ensures we don't set any in the top right quadrant.

```
leds = {}
for y in range(16):
    for x in range(16):
        if x < 8 or y < 8:
            leds[x, y] = hsv_colour((x+y)/24, 1, 1)
display.set_leds(leds)
```



Binary Clock

Everything in a computer is made of ones and zeros so what better way to combine the digital world with the physical one than with a binary clock. A binary clock takes a 24 hour clock and converts each decimal digit (0-9) to a 4-bit binary value, with the most significant bit at the top.



The time shown is 14:38:51. For more details see:
https://www.wikihow.com/Read-a-Binary-Clock

Converting to binary

Python has a useful function called format which allows you to convert between different number representations. To convert decimal to a 4 bit binary string use the following:

```
format(decimal_digit, '04b')
```

The first zero says to include the leading zeros, e.g. 2 -> 0010 instead of 2 -> 10. The 4 means four digits and the b means binary.

Drawing a column

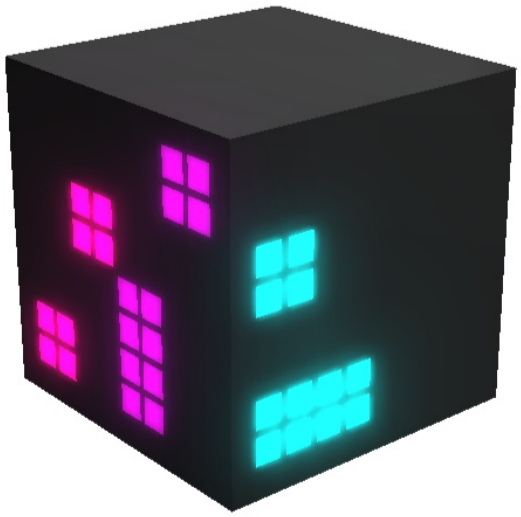
So first let's create a function that draws a binary value as a column. Instead of using 1 pixel per value we'll use 2x2 to make it look clearer.

```
def draw_column(decimal_digit, x, colour):
    # Convert digit to four digit binary
    binary = list(format(decimal_digit, '04b'))
    # Start at the bottom with the least significant
digit
    binary.reverse()
    leds = {}
    for i, value in enumerate(binary):
        # Set all the leds in a 2x2 square
        pixel = colour if value == '1' else black
        y = i * 2
        leds[x,   y  ] = pixel
        leds[x,   y+1] = pixel
        leds[x+1, y  ] = pixel
        leds[x+1, y+1] = pixel
    display.set_leds(leds)
```

Drawing the clock

To create the full clock we need to call it for each decimal digit, so first we get the current time with the datetime library, then we use format again to convert the value into two decimal digits in a string.

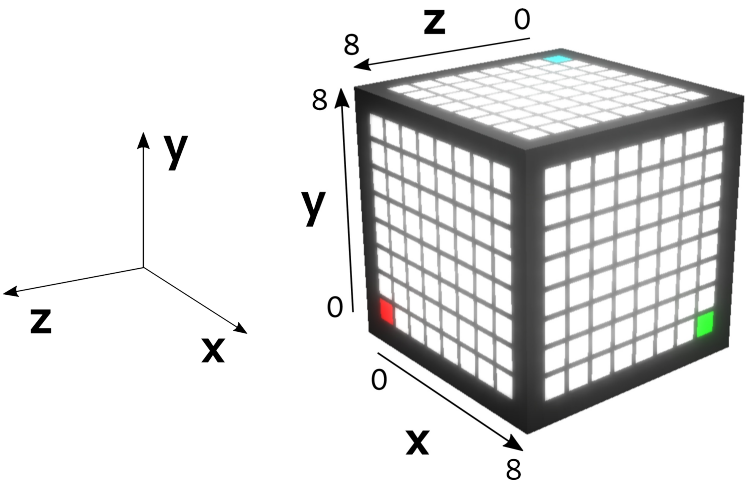
```
import datetime
display.set_all(black)
while True:
    time_now = datetime.datetime.now()
    seconds = format(time_now.second, '02')
    minutes = format(time_now.minute, '02')
    hours   = format(time_now.hour,   '02')
    draw_column(int(hours[0]),    0, pink)
    draw_column(int(hours[1]),    2, pink)
    draw_column(int(minutes[0]),  4, purple)
    draw_column(int(minutes[1]),  6, purple)
    draw_column(int(seconds[0]),  8, cyan)
    draw_column(int(seconds[1]), 10, cyan)
    ti
```



Setting LEDs with 3D coordinates

This method is useful when trying to make something which interacts with the 3D world like the water level project on page 52. It takes a dictionary like set_leds but with 3 coordinates. The origin is at the bottom left pixel. Try:

```
leds = {}
leds[0, 0, 8] = red
leds[8, 0, 0] = green
leds[0, 8, 0] = cyan
display.set_3d(leds)
```

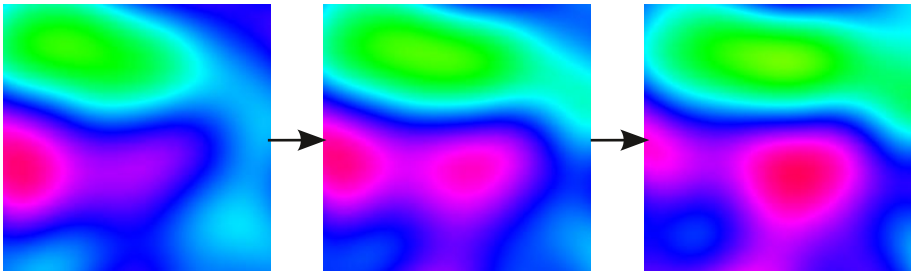


The left panel is set with z equal to 8, and x and y going from 0 to 7.
The right panel is set with x equal to 8, and y and z going from 0 to 7.
The top panel is set with y equal to 8, and x and z going from 0 to 7.

This might seem a little strange at first but the reason the faces are at 8, not 7 is because there is a gap of almost exactly one LED between the LED panels. So if you want something to 'flow' across the panels it looks more natural with this coordinate scheme.

Lava Lamp

In this project we use a kind of pattern called “Simplex noise”. It is great for creating random patches that grow and shrink and move around. It's a very powerful tool used in all sorts of applications, such as generating random landscapes, fire effects or clouds. In our case this looks a lot like lava flowing over time.



While the simplest kind of Simplex noise is 2D noise, we'll use 4D noise - where the fourth dimension is time. We've created a function to generate this noise for you, just call:

noise_4d(x, y, z, time)

It returns a value between -1 and 1.

Calculate one LED

First create a function which returns the colour of each LED, based on the time and its position:

```
def lava_colour(x, y, z, t):
    scale = 0.10
    speed = 0.05
    hue = noise_4d(scale * x, scale * y, scale * z,
speed * t)
    return hsv_colour(hue, 1, 1)
```

The noise value is simply used to control the hue of the colour. The variables scale and speed determine the size of the blobs and how fast they change.

Loop over all LEDs

Now create a function which updates the colour of each LED. The “if” condition skips coordinates which are not on the cube’s surface:

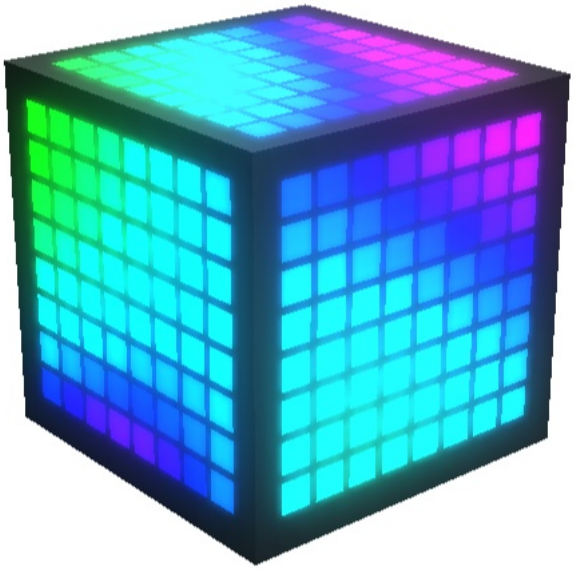
```
def paint_cube(t):
    colours = {}
    for x in range(9):
        for y in range(9):
            for z in range(9):
                if x == 8 or y == 8 or z == 8:
                    colour = lava_colour(x, y, z, t)
                    colours[x,y,z] = colour
    display.set_3d(colours)
```

Repeat over time

Finally create a loop which paints the entire cube 30 times per second:

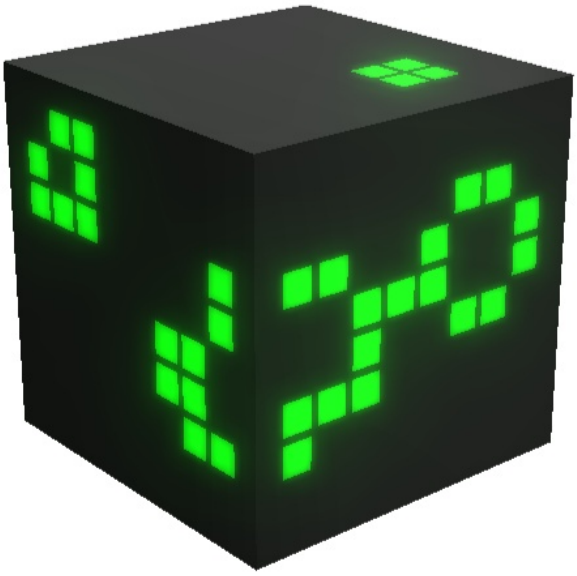
```
t = 0
while True:
    paint_cube(t)
    time.sleep(1/30)
    t += 1
```

Try changing the scale and speed to give different effects.



Conway's Game Of Life

Conway's Game Of Life is one of the first and most famous cellular automatons. It consists of a grid of 'cells' that can either become alive or die each turn, based on the cells around them. It creates all sorts of interesting patterns as the game evolves.



Rules

The game has a few very simple rules:

- Any alive cell with fewer than two neighbours dies (underpopulation).
- Any alive cell with more than three neighbours dies (overpopulation).
- Any alive cell with two or three neighbours lives on to the next turn.
- Any dead cell with exactly three neighbours becomes alive, as if by reproduction.

We'll play out this game on the grid of LEDs. If an LED is off the cell is dead and if it's on the cell is alive.

Setup the game

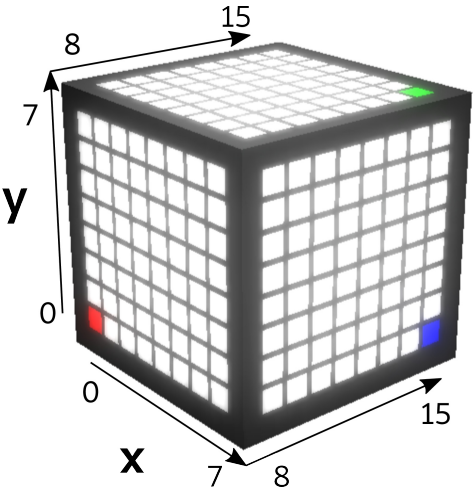
Let's first create a function to setup a new game. We'll keep track of all the alive cells with a global list of x and y coordinates. We need to populate the game with some living cells, not too many or too few though otherwise they'll all quickly die. Plus we will keep track of the number of turns of the game and choose a random colour.

```
import random
max_turns = 50
starting_live_ratio = 0.4
colour = random_colour()
num_turns = 0
alive_cells = []
for x in range(0,16):
    for y in range(0,16):
        if x < 8 or y < 8:
            if random.random() < starting_live_ratio:
                alive_cells.append((x,y))
```

Work out the neighbours

Then we need a function to work out how many alive neighbours a cell has, so we loop through all cells left and right, above and below. We also need to handle any cells that go over the edge because this is on a cube and not a flat grid.

```
def num_alive_neighbours(x, y):
    num_neighbours = 0
    for x2 in [x-1, x, x+1]:
        for y2 in [y-1, y, y+1]:
            if (x2, y2) != (x, y):
                neighbour = (x2, y2)
                # Account for 3D nature of panels
                if x2 == 8 and y2 >= 8:
                    neighbour = (y2, 7)
                elif y2 == 8 and x2 >= 8:
                    neighbour = (7, x2)
                if neighbour in alive_cells:
                    num_neighbours += 1
    return num_neighbours
```



Run the game

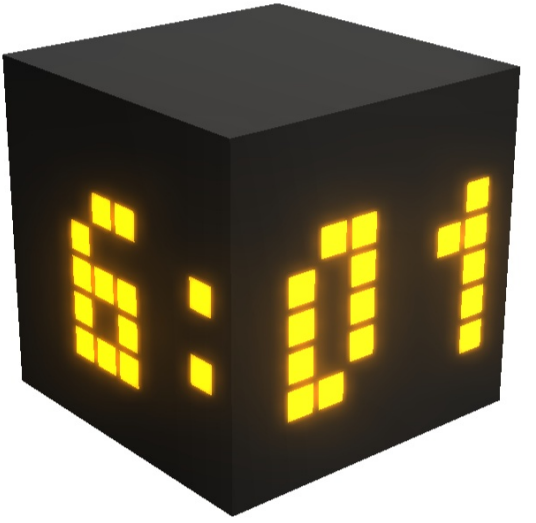
Now run the actual game. Each turn check each cell's neighbours and create a new list of the next alive cells.

```
while (num_turns < max_turns):
    next_cells = []
    leds = {}
    for x in range(0,16):
        for y in range(0,16):
            if x < 8 or y < 8:
                alive = (x, y) in alive_cells
                neighbours = num_alive_neighbours(x, y)
                # Remains alive
                if alive and (neighbours == 2
                        or neighbours == 3):
                    next_cells.append((x, y))
                    leds[x, y] = colour
                # Becomes alive
                elif not alive and neighbours == 3:
                    next_cells.append((x, y))
                    leds[x, y] = colour
                # Else dead
                else:
                    leds[x, y] = black
    alive_cells = next_cells
    display.set_leds(leds)
    time.sleep(0.3)
    num_turns += 1
```

Scrolling Clock

We've also added a method that allows you to write text that scrolls across the LEDs. So creating a scrolling clock is as simple as:

```
import datetime
while True:
    time_text = datetime.datetime.now().strftime("%H:
%M")
    display.scroll_text(time_text, orange)
    time.sleep(20)
```



Speaker

API

```
speaker.volume = value

speaker.say(text)

speaker.play(file)

speaker.tone(
  frequency=261.626,
  duration=0.5,
  volume=0.25,
  function=sine_wave
)
```

Text to speech

We've integrated text to speech software with our system, so to read out some text it's this simple.

```
  speaker.say("Hello there")
```

Play a tone

To make your own sounds you can call the tone method. It will play a fixed frequency for the number of seconds you specify:

```
  speaker.tone(800, 2)
```

By default it plays a sine wave but you can also tell it to use a square wave or white noise function to give you different effects.

Play an audio file

To play an audio file in Python you first have to upload an audio file to the Raspberry Pi. You can do this using the upload button on the dashboard. It will place it on the Raspberry Pi's SD card in "/home/pi/Desktop." You can then just call play:

```
  speaker.play("rain.mp3")
```

If you place a file elsewhere make sure you pass the full path:

```
  speaker.play("/home/pi/Audio/rain.mp3")
```

We use ffmpeg under the covers to convert the file, so most audio formats are supported.

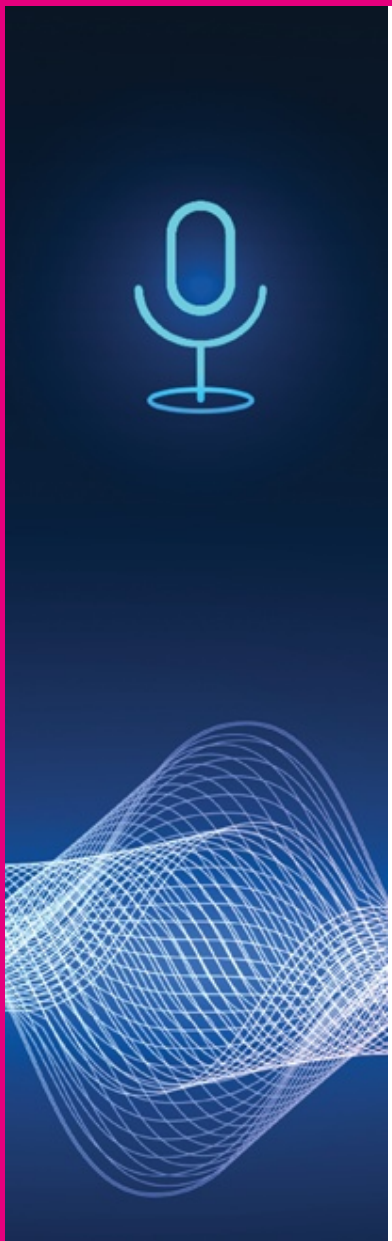
Chiptunes

Here we create a program that generates it's own tune. Notice how white noise can give you a bit of a drum beat sound.

```
while True:
    # Play a rising piece
    for frequency in range(500, 2000, 100):
        speaker.tone(frequency, 0.01)
    # Play a beat and then another beat
    time.sleep(0.02)
    speaker.tone(500, 0.1, 0.1, function=white_noise)
    time.sleep(0.05)
    speaker.tone(500, 0.1, 0.1, function=white_noise)
    # Play 3 different tones
    speaker.tone(500 + 500 * random.random(), 0.1)
    speaker.tone(500 + 500 * random.random(), 0.1)
    speaker.tone(500 + 500 * random.random(), 0.1)
    # Play another rising piece
    for frequency in range(200, 1000, 10):
        speaker.tone(frequency, 0.003)
```

It's not exactly Super Mario but hopefully you can see how you can create interesting sound effects with just a few lines of code.

Microphone



API

```
microphone.start_recording(file)

microphone.stop_recording()

microphone.start_voice_recognition()

microphone.wait_for_sentence(timeout)

microphone.stop_voice_recognition()
```

Voice Recognition

With a microphone and the power of the Raspberry Pi you can do full blown voice recognition.

As with all voice recognition programs you need to say the wake word to start it converting the speech to text. Amazon for example has "Alexa", we are using Mycroft's wake word detector so you need to say:

"Hey, Mycroft"

It will then reply with :

"Yes"

After that anything you say will be converted to text ready for your program to use.

```
microphone.start_voice_recognition()
while True:
    # Wait up to 1000 seconds for some speech
    sentence = microphone.wait_for_sentence(1000.0)
    # Convert text back to speech
    speaker.say('You said ' + sentence)
```

Buttons

API

```
buttons.get_next_action(timeout)

buttons.top_pressed

buttons.top_pressed_count

buttons.middle_pressed

buttons.middle_pressed_count

buttons.bottom_pressed

buttons.bottom_pressed_count
```

Dictaphone

buttons.get_next_action(timeout) waits until a button is pressed or until it times out. It returns a string with the name of the button that was pressed.

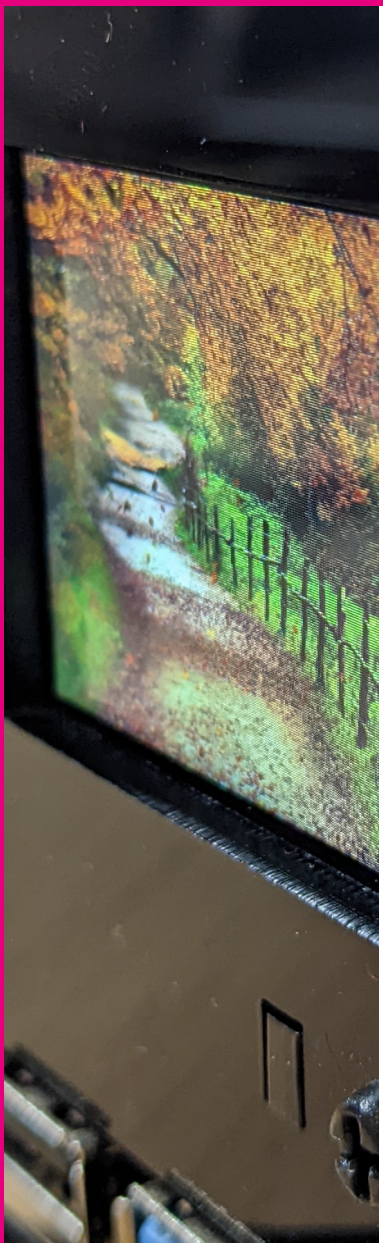
Let's use it to create a dictaphone for recording ideas or speeches!

```
file = "voicenote.mp3"
while True:
    # Wait up to 100 seconds for the next action
    action = buttons.get_next_action(100.0)
    if action == "top":
        microphone.start_recording(file)
    elif action == "middle":
        microphone.stop_recording()
    elif action == "bottom":
        speaker.play(file)
```

Here we continually loop waiting for the next button press. The top button starts the recording, the middle stops it and the bottom plays what we've recorded.

What if someone presses the top button twice though? Can you improve this code to make it more robust?

Screen



API

```
screen.resolution_scaling

screen.set_pixel(x, y, colour)

screen.set_pixels(x, y, width,
height,      pixels)

screen.draw_rectangle(x, y, width,
height,      colour)

screen.write_text(x, y, text, size,
colour, background_colour)

screen.draw_image(image, x=0, y=0,
width=320, height=240)
```

Draw rectangles and write text

Try playing around:

```
screen.draw_rectangle(20, 20, 280, 200, pink)
screen.write_text(50, 50, "LumiCube", 2, pink, white)
```

Setting pixels

Try setting a pixel:

```
screen.set_pixel(2, 2, green)
```

It's pretty small right! The screen resolution is 320 x 240 pixels which is an awful lot of pixels, so it can be helpful to reduce the resolution:.

```
screen.resolution_scaling = 10
screen.set_pixel(2, 2, green)
screen.resolution_scaling = 1
```

Now each pixel is effectively 10x10 pixels and the resolution is only 32 new pixels wide and 24 new pixels high.

Draw image

To draw an image you need to upload it using the upload button on the dashboard. It will place the file on the Desktop. There's already a file "autumn.jpg" uploaded for you to try:

```
screen.draw_image("autumn.jpg")
```

If you place it somewhere else you need to pass the full path.

Pi stats example

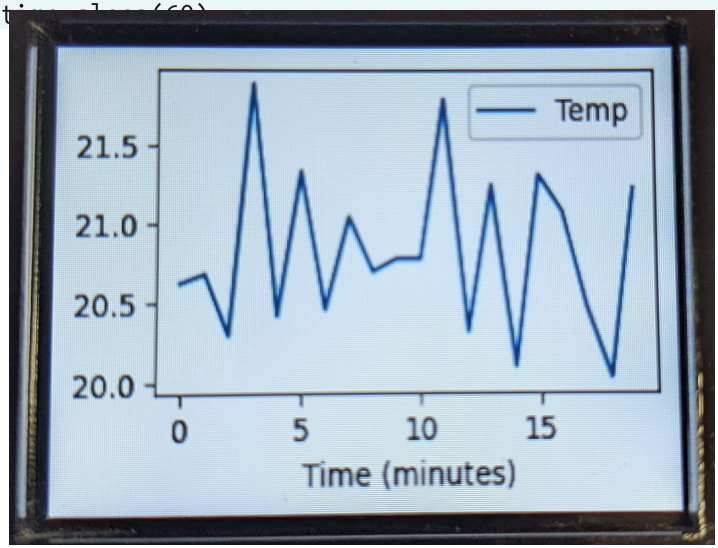
Display some Raspberry Pi stats on the screen. Note: you can use \n to denote a new line when calling write_text.

```
def to_text(value):
    return str("{:.1f}".format(value))

screen.draw_rectangle(0, 0, 320, 240, black)
height = 36
while True:
    text = ("IP Address: " + pi.ip_address() + "\n"
        + "CPU temp  : " + to_text(pi.cpu_temp()) + "\n"
        + "CPU usage : " + to_text(pi.cpu_percent()) +
"\n"
        + "RAM usage : " +
to_text(pi.ram_percent_used()) +"\n"
        + "Disk usage: " + to_text(pi.disk_percent()) +
"\n")
    screen.write_text(10, 18, text, 1, white, black)
    time.sleep(5)
```

Advanced example: Drawing charts

If you want to do some more powerful things you can import other libraries. For drawing charts try this matplotlib example:

```
import matplotlib.pyplot as plt
file_name = "Chart.jpg"
temperatures = [0 for i in range(0,20)]
while True:
    temperatures.pop(0)
    temperatures.append(env_sensor.temperature)
    fig = plt.figure(figsize=(3.20, 2.40*0.8))
    times = [i for i in range(0,20)] # plot 20 minutes
    plt.plot(times, temperatures, label = "Temp")
    plt.xlabel('Time (minutes)')
    plt.legend()
    fig.savefig(file_name, bbox_inches='tight')
    screen.draw_image(file_name)
```



Light sensor

API

```
light_sensor.get_next_gesture(timeout
)

light_sensor.ambient_light

light_sensor.red

light_sensor.green

light_sensor.blue

light_sensor.last_gesture

light_sensor.num_gestures

light_sensor.within_proximity
```

Light levels

Try plotting the ambient_light on the dashboard and see how it changes as you move your hand over the sensor. Now make the brightness of the cube react:

```
display.set_all(green)
while True:
    display.brightness = min(50,
light_sensor.ambient_light /              500)
    time.sleep(0.01)
```

Colour sensing

As well as an ambient light sensor there are also red, green and blue sensors. These sensors have a very similar response as the colour receptors in our eyes. Note, the result of these sensors is very different to RGB colour. A pure red light (0xFF0000) will also stimulate the green and blue sensors.

Gesture sensing

The sensor has an infra-red LED with four detectors. These are arranged so that it can tell which direction your hand is moving by which sensor triggers first. Try:

```
while True:
    gesture = light_sensor.get_next_gesture(100)
    if gesture == "up":
        display.set_all(blue)
    elif gesture == "down":
        display.set_all(red)
```

Environment Sensor

API

```
env_sensor.temperature

env_sensor.pressure

env_sensor.humidity
```

Barometer

There are plenty of interesting things you can measure with the environment sensor. For example you can measure someone breathing on the cube by an increase in the humidity (try the windmill app!). Or you can detect if a door is shut in a closed room with the pressure sensor.

Let's use the pressure sensor to create a barometer. Barometers have been used for hundreds of years to estimate the weather, where high pressure means sunny weather and low pressure means stormy weather.

```
while True:
    if env_sensor.pressure > 102000:
        display.scroll_text("Fair", yellow)
    elif env_sensor.pressure > 99000:
        display.scroll_text("Change", white)
    else:
        display.scroll_text("Rain", blue)
    time.sleep(20)
```

Instead of text you could draw images of the sun or clouds depending on the pressure. You could even go for the advanced option and create some animations like the rain animation app.

IMU

API

imu.pitch

imu.roll

imu.yaw

imu.acceleration_x

imu.acceleration_y

imu.acceleration_z

imu.angular_velocity_x

imu.angular_velocity_y

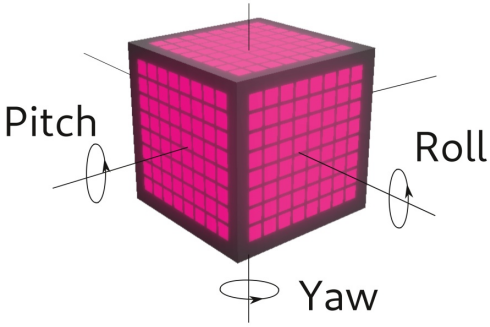
imu.angular_velocity_z

imu.gravity_x

imu.gravity_y

imu.gravity_z

Orientation



The pitch, roll and yaw values represent the orientation of the cube measured in degrees. Pitch goes from -180 to 180, roll from -90 to 90, and yaw from 0 to 360 degrees. To get a feel for it try using the yaw to set the brightness:

```
display.set_all(purple)
while True:
    display.brightness = int(imu.yaw * 50 / 360)
    time.sleep(0.01)
```

Now try rotating the cube on a flat surface. Try repeating this with the pitch and the roll.

Acceleration

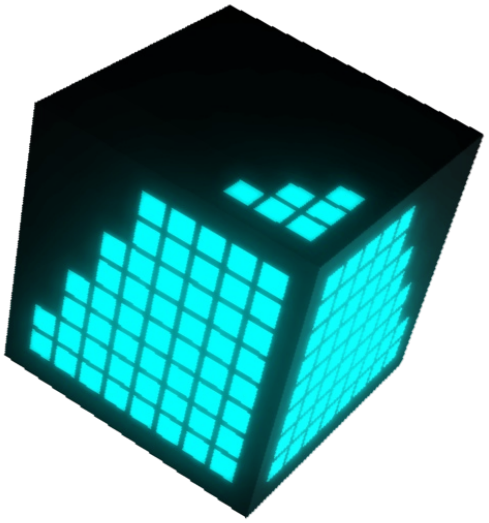
The IMU cleverly splits apart gravity and other acceleration so you can measure if the cube is pushed or knocked in any direction.

Angular velocity

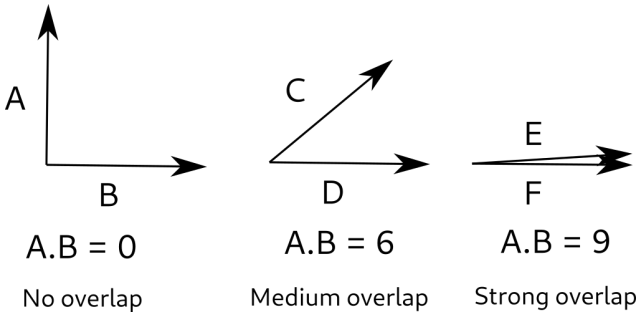
The angular velocity is how fast the LumiCube is spinning. We don't recommend you use this too much :)

Water Level

Make the cube look like it's filled with water. Water that moves around as you move the cube.

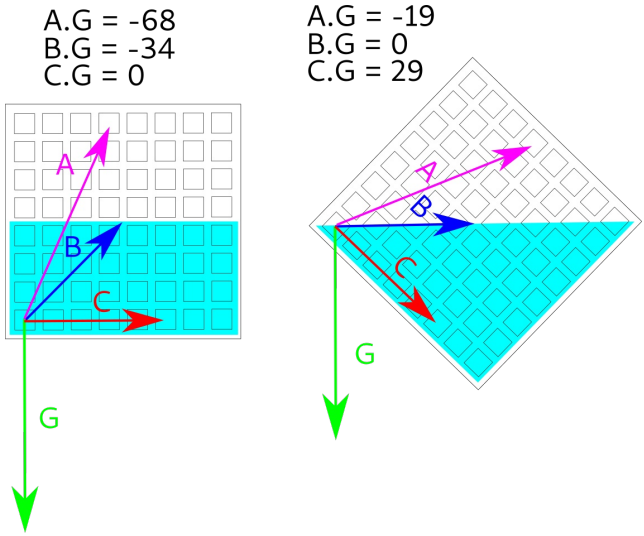


This project is going to use some advanced maths, namely vectors. A vector is a way of representing a direction. In our case a 3D direction, so it has a x, y, and z value. Two vectors can be "multiplied" using the dot product to give the amount that they overlap. If two vectors are in the same direction you get a strong overlap, if they are in opposite directions you get a strong negative overlap, and if they are in perpendicular directions then you get little or no overlap.



So how does this help us? Taking the vector from an origin to a LED on the cube and doing the dot product with gravity gives a value that represents how 'high' that LED is above the origin. Actually you need to take the negative of the dot product to represent height as gravity is pointing down.

The diagram below shows three vectors (A, B, C) pointing to different points on the cube, and gravity (G) pointing down. B is the vector to the midpoint. As you rotate the cube the dot product gives you a representation of the height of each point above the origin. You can then compare the height of the LED to the midpoint to see whether it would be under water.



It might seem complicated but the way you calculate the dot product is actually really easy:

```
def dot_product(vector1, vector2):
    (x1,y1,z1) = vector1
    (x2,y2,z2) = vector2
    return (x1 * x2) + (y1 * y2) + (z1 * z2)
```

With a sort of measure of how high an LED is we just need to compare the height of each LED with the height of the midpoint. If the LED is lower than the midpoint then it should be filled with water, so colour it blue.

```
def led_below_water_level(x, y, z, gravity):
    mid_point_height = dot_product((4.5, 4.5, 4.5),
gravity)
    led_bottom = (x, y, z)
    led_top = (x+1, y+1, z+1)
    max_height = max(dot_product(led_bottom, gravity),
        dot_product(led_top, gravity))
    return (max_height < mid_point_height)
```

Now we just need to run it per LED.

```
while True:
    gravity = (imu.gravity_x, imu.gravity_y,
imu.gravity_z)
    leds = {}
    for x in range(9):
        for y in range(9):
            for z in range(9):
                if x == 8 or y == 8 or z == 8:
                    if led_below_water_level(x,y,z,
gravity):
                        leds[x, y, z] = cyan
                    else:
```

Multiple Cubes

It's easy to make two cubes talk to one another. All the modules and methods we've covered so far all actually exist on a cube object as well as on their own. So display.set_leds() is the same as cube.display.set_leds().

To talk to another cube just create a new Cube object with that cube's IP address and use it as normal. For example:

```
cube2 = LumiCube("192.168.0.12")
cube2.display.set_all(pink)
```

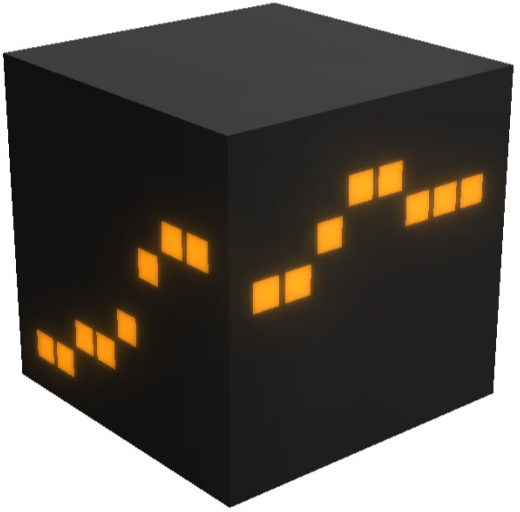
Try mixing and matching different calls to different cubes:

```
while True:
    colour1 = hsv_colour(cube2.imu.yaw / 360, 1, 1)
    colour2 = hsv_colour(cube1.imu.yaw / 360, 1, 1)
    cube1.display.set_all(colour1)
    cube2.display.set_all(colour2)
    time.sleep(0.1)
```

The cube has to be on the same network for this to work. Also transfers over the network can be quite slow.

Crypto Display

There are thousands of weird, wonderful and incredibly useful data sources available on the internet, all of which can be used with the LumiCube. First you need to find a website that provides the data you want and check out their API documentation. Note, most sources require authentication. For this example we're getting cryptocurrency prices from a free provider that doesn't require authentication: www.gemini.com



To get the price of Bitcoin in USD you need to send this HTTP GET request:
https://api.gemini.com/v1/pricefeed/btcusd

You will then get back a HTTP response of:
[{'pair': 'BTCUSD', 'price': '41759.89', 'percentChange24h': '0.0172'}]

So let's write a function to get the price. First we import the requests package to handle HTTP gets and responses. Then we do a get and convert the response to a JSON dictionary which we can index into to get our price.

```
import requests
def get_price(from_currency, to_currency):
    r = requests.get("https://api.gemini.com/v1/
pricefeed/"
        + from_currency + to_currency)
    price = float(r.json()[0]['price'])
    return price
```

Now, let's display the prices as a scrolling chart on the cube. Every 2 seconds we fetch the new price and redraw the chart.

```
initial = get_price('btc', 'usd')
prices = [initial for i in range(16)]
while True:
    # Add the new price and remove the oldest one
    prices.append(get_price('btc', 'usd'))
    prices = prices[1:]
    step = 3.0
    led_colours = {}
    for x in range(0, len(prices)):
        y_value = int((prices[x] - min(prices))/step)
        for y in range(8):
            colour = white if y == y_value else black
            led_colours[x, y] = colour
    display.set_leds(led_colours)
    time.sleep(2)
```


ABSTRACT
FOUNDRY