

Contenido

app.py.....	3
Alarm.py	8
Clients.py.....	9
ConvBelt.py.....	14
Gates.py	16
Lights.py	19
Screen.py.....	22
Supervisor.py.....	23
Parte del front	31
Views.py	31
Urls.py	36

app.py

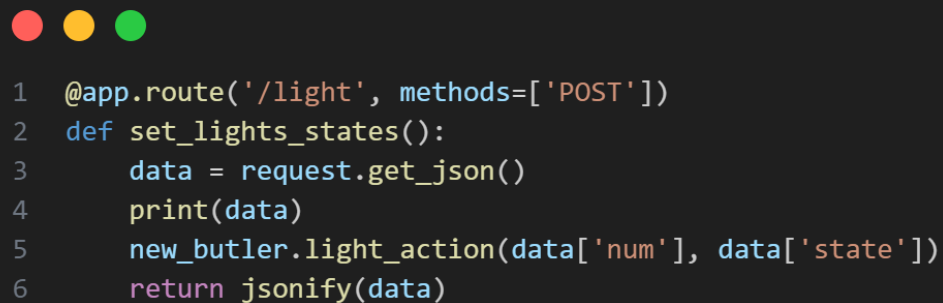
Ruta raíz (/)

Devuelve un simple mensaje "Hello, World!" cuando se accede a la raíz del servidor.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It contains three lines of Python code:

```
1 @app.route('/')
2 def index():
3     return "Hello, World!"
```

Control de Luces Individuales (/light)


A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It contains six lines of Python code:

```
1 @app.route('/light', methods=['POST'])
2 def set_lights_states():
3     data = request.get_json()
4     print(data)
5     new_butler.light_action(data['num'], data['state'])
6     return jsonify(data)
```

Recibe una solicitud POST con datos JSON para controlar el estado de una luz específica.

Llama a `light_action` del butler con el número de luz y su estado (encendido/apagado).


Control de Todas las Luces (/lights)



```
1 @app.route('/lights', methods=['POST'])
2 def set_all_lights_states():
3     data = request.get_json()
4     print(data)
5     new_butler.light_action(-1, data['state'])
6     return jsonify(data)
```

Similar al anterior, pero controla todas las luces a la vez pasando -1 como número de luz.


Activación de la Banda Transportadora (/transport_band)



```
1 @app.route('/transport_band', methods=['POST'])
2 def set_transport_band():
3     data = request.get_json()
4     print(data)
5     new_butler.belt_activate()
6     return jsonify(data)
```

Recibe una solicitud POST y activa la banda transportadora llamando a `belt_activate` del butler.


Estado de la Banda Transportadora (/transport_band_status)



```
1 @app.route('/transport_band_status', methods=['GET'])
2 def get_transport_band_status():
3     return jsonify({'state': 0})
```

Devuelve el estado actual de la banda transportadora. Aquí está hardcodeado a 0.

Control del Garaje (/Garage)



```
1 @app.route('/Garage', methods=['POST'])
2 def set_garage():
3     data = request.get_json()
4     new_butler.gate_action(data['state'])
5     return jsonify(data)
```

Recibe una solicitud POST para controlar el estado del garaje llamando a gate_action del butler.

Estado del Garaje (/garage_status)



```
1 @app.route('/garage_status', methods=['GET'])
2 def get_garage_status():
3     return jsonify({'state': new_butler.gate_state})
```

Devuelve el estado actual del garaje obteniéndolo del butler.

Conteo Actual de Personas (/actual_people)



```
1 @app.route('/actual_people', methods=['GET'])
2 def get_actual_people():
3     return jsonify({'actual_people': new_butler.clients})
```

Devuelve el número actual de personas supervisadas por el butler.


Conteo de Personas Total (/counter_people)



```
1 @app.route('/counter_people', methods=['GET'])
2 def get_counter_people():
3     return jsonify({'counter_people': new_butler.clients})
```

Devuelve el conteo total de personas (parece idéntico a get_actual_people en este ejemplo).


Control de la Alarma (/alarm)



```
1 @app.route('/alarm', methods=['POST'])
2 def set_alarm():
3     data = request.get_json()
4     print(data)
5     return jsonify(data)
```


Recibe una solicitud POST para configurar la alarma, pero no realiza ninguna acción específica en el butler.

Estado de la Alarma (/alarm_status)



```
1 @app.route('/alarm_status', methods=['GET'])
2 def get_alarm_status():
3     return jsonify({'state': 0})
```

Ejecución de la Aplicación



```
1  if __name__ == '__main__':  
2      app.run(debug=True)  
3
```

Ejecuta la aplicación Flask en modo de depuración cuando se ejecuta directamente el script.

Alarm.py

Primero, importa la función `sleep` del módulo `time` y `GPIO` del módulo `RPi.GPIO`. Esta función se utiliza para hacer una pausa en la ejecución del programa durante un número especificado de segundos.

Luego, se configura el modo de los pines `GPIO` a `GPIO.BOARD`, lo que significa que los números de los pines se refieren a los números de los pines del conector físico de la Raspberry Pi.

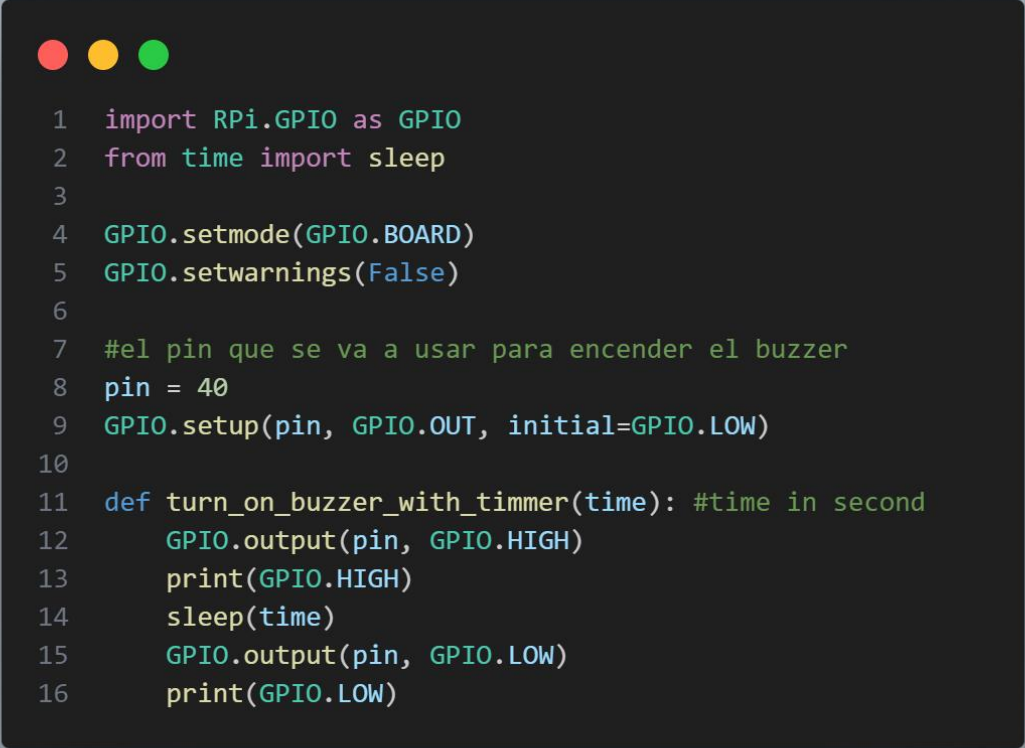
Después, se desactivan las advertencias con `GPIO.setwarnings(False)`. Esto es útil para evitar mensajes de advertencia si se ha configurado un pin de una manera particular en otro lugar de nuestro código.

Se define una variable `pin` con el valor 40, que representa el pin que se va a usar para controlar el buzzer.

La función `GPIO.setup` se utiliza para configurar el modo del pin como salida (`GPIO.OUT`). Además, se establece un estado inicial para el pin, que en este caso es `GPIO.LOW` (el buzzer está apagado).

Finalmente, se define una función `turn_on_buzzer_with_timmer` que toma un argumento `time` (el tiempo en segundos que el buzzer debe estar encendido). Dentro

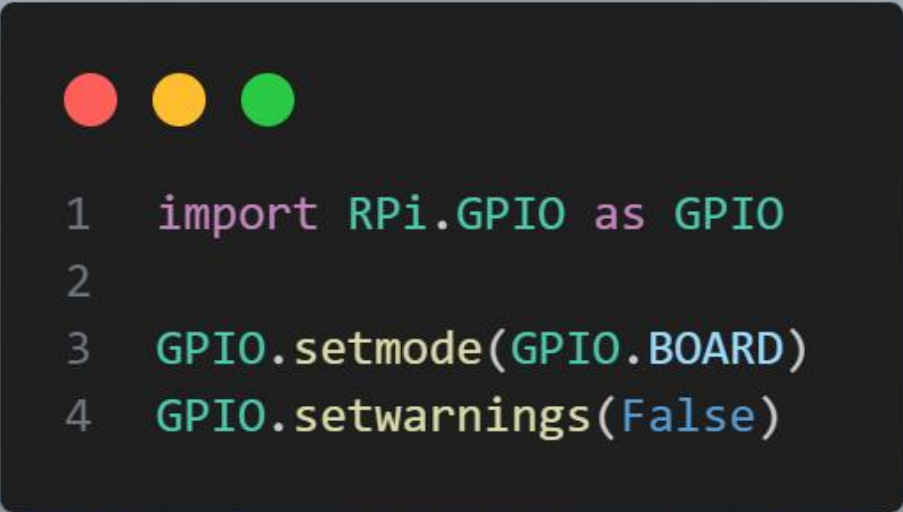
de esta función, se utiliza `GPIO.output` para cambiar el estado del pin a `GPIO.HIGH` (enciende el buzzer), se imprime el estado del pin, se hace una pausa por el tiempo especificado, luego se cambia el estado del pin a `GPIO.LOW` (apaga el buzzer) y se imprime el estado del pin.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The terminal displays Python code for controlling a buzzer on a Raspberry Pi. The code includes imports for RPi.GPIO and time, sets the GPIO mode to BOARD and disables warnings, defines a pin number (40), and sets up the pin as an output. A function 'turn_on_buzzer_with_timmer' is defined, which turns the buzzer on for a specified time, prints the state, and then turns it off and prints the state again.

```
1  import RPi.GPIO as GPIO
2  from time import sleep
3
4  GPIO.setmode(GPIO.BOARD)
5  GPIO.setwarnings(False)
6
7  #el pin que se va a usar para encender el buzzer
8  pin = 40
9  GPIO.setup(pin, GPIO.OUT, initial=GPIO.LOW)
10
11 def turn_on_buzzer_with_timmer(time): #time in second
12     GPIO.output(pin, GPIO.HIGH)
13     print(GPIO.HIGH)
14     sleep(time)
15     GPIO.output(pin, GPIO.LOW)
16     print(GPIO.LOW)
```

Clients.py


Primero, se establece el modo de numeración de los pines a `GPIO.BOARD` para usar la numeración física de la placa. Luego, se desactivan las advertencias con `GPIO.setwarnings(False)`.



```
1 import RPi.GPIO as GPIO
2
3 GPIO.setmode(GPIO.BOARD)
4 GPIO.setwarnings(False)
```

Se define una lista `pin_arr` que contiene los números de los pines que se van a utilizar. Para cada pin en `pin_arr`, se configura como salida (`GPIO.OUT`) y se inicializa en estado bajo (`GPIO.LOW`).

Se define una lista `binary_nums` que contiene representaciones binarias de los números del 0 al 9.



```
1 pin_arr = [35, 36, 37, 38] # 1, 2, 4, 8
2
3 for pin in pin_arr:
4     GPIO.setup(pin, GPIO.OUT, initial=GPIO.LOW)
5
6 binary_nums = [
7     [0, 0, 0, 0],
8     [0, 0, 0, 1],
9     [0, 0, 1, 0],
10    [0, 0, 1, 1],
11    [0, 1, 0, 0],
12    [0, 1, 0, 1],
13    [0, 1, 1, 0],
14    [0, 1, 1, 1],
15    [1, 0, 0, 0],
16    [1, 0, 0, 1]
17 ]
18
```

La función `convert_bin(num)` toma un número entero como entrada y devuelve su representación binaria como una lista de 1s y 0s. Si el número es mayor o igual a 10, devuelve la representación binaria del número 9.



```
1  # Parameter num is a integer
2  def convert_bin(num):
3      binary = []
4      if num < 10:
5          binary = binary_nums[num]
6      else:
7          binary = binary_nums[9]
8      return binary
```

La función `convert_num(binary)` toma una lista de 1s y 0s como entrada y devuelve el número entero correspondiente. Si no se encuentra una coincidencia en `binary_nums`, devuelve -1.



```
1  # Parameter binary is an array of 1s and 0s
2  def convert_num(binary):
3      num = -1
4      i = 0
5      for bin in binary_nums:
6          if bin == binary:
7              num = i
8              i = i + 1
9      return num
```

La función `show_binary(num)` toma un número entero como entrada, limpia el display con `clean_display()`, convierte el número a binario con `convert_bin(num)`, y luego enciende los pines correspondientes en la placa Raspberry Pi.



```
1  def show_binary(num):
2      clean_display()
3      binary = convert_bin(num)
4      i = 0
5      for j in range(0, 4):
6          if binary[j] == 1:
7              GPIO.output(pin_arr[j], GPIO.HIGH)
```

Finalmente, la función `enter_exit_client(entrada)` implementa una lógica para determinar si un cliente entra o sale. Utiliza dos sensores, A y B. Si el sensor B se


activa primero, se asume que un cliente ha entrado. Si el sensor A se activa primero, se asume que un cliente ha salido. En ambos casos, los sensores se resetean a False después de detectar una entrada o salida. Si no se detecta ninguna actividad, la función devuelve 0.

```
1  #esto es para la logica de ver si entra o sale un cliente
2  A = False # primer sensor, este debe de ir en la entrada
3  B = False # segundo sensor, este debe de ir adentro
4
5  def enter_exit_client(entrada):
6      global A, B
7      if entrada == "B":
8          B = not B
9          if A:
10             A = False
11             B = False
12             return 1 #esto indica que entro un cliente
13     elif entrada == "A":
14         A = not A
15         if B:
16             A = False
17             B = False
18             return -1 #esto indica que salio un cliente
19     return 0 #esto indica que no paso nada
```

ConvBelt.py

Primero, se configura el modo de numeración de los pines a GPIO.BOARD con GPIO.setmode(GPIO.BOARD). Esto significa que los números de los pines se corresponden con los números de los pines en el conector de la Raspberry Pi.


Luego, se desactivan las advertencias con GPIO.setwarnings(False). Esto es útil para evitar mensajes de advertencia si se ha configurado un pin en un script anterior.



```
1 import RPi.GPIO as GPIO
2
3 GPIO.setmode(GPIO.BOARD)
4 GPIO.setwarnings(False)
```

Después, se definen tres pines: pinLV (luz verde), pinLR (luz roja) y pinM (motor), que corresponden a los pines 8, 10 y 12 respectivamente.

A continuación, se configuran estos pines como salidas con GPIO.setup(). El parámetro initial define el estado inicial del pin. En este caso, el pin de la luz verde y el motor se inician en estado bajo (GPIO.LOW), lo que significa que están apagados, y el pin de la luz roja se inicializa en estado alto (GPIO.HIGH), lo que significa que está encendido.

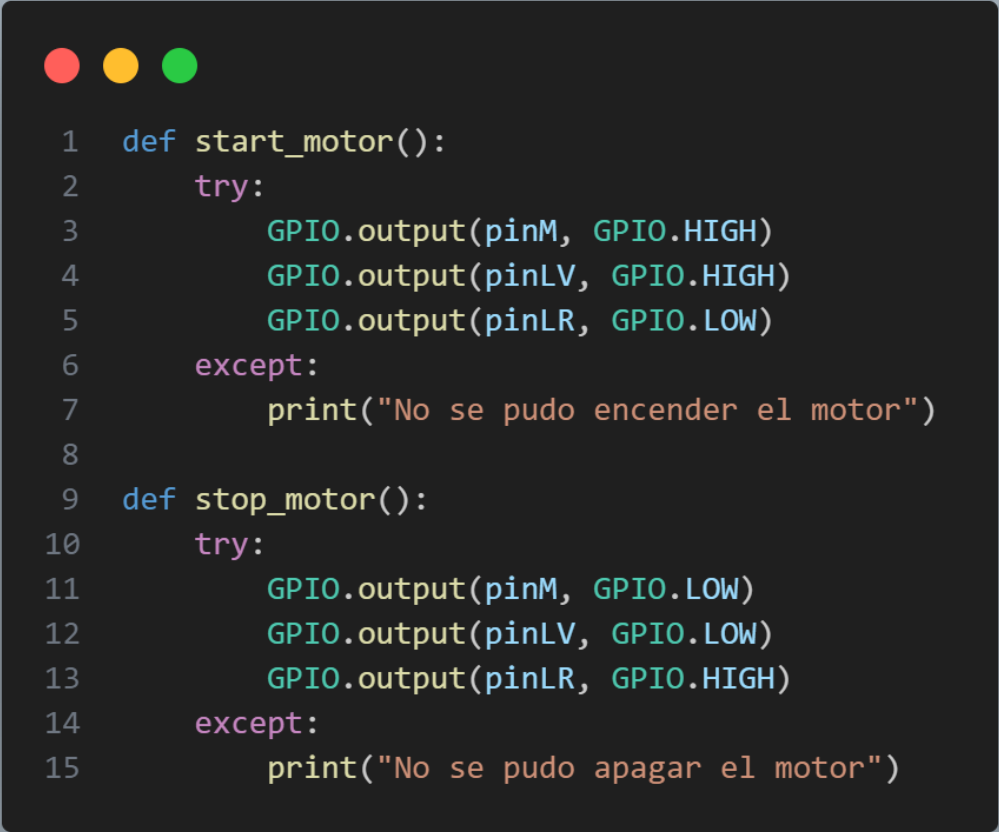


```
1 pinLV = 8 #luz verde
2 pinLR = 10 #luz roja
3 pinM = 12 #motor
4 GPIO.setup(pinLV, GPIO.OUT, initial=GPIO.LOW)
5 GPIO.setup(pinLR, GPIO.OUT, initial=GPIO.HIGH)
6 GPIO.setup(pinM, GPIO.OUT, initial=GPIO.LOW)
```

El código también define dos funciones: `start_motor()` y `stop_motor()`. Estas funciones intentan encender y apagar el motor respectivamente, cambiando el estado de los pines correspondientes. Si ocurre algún error al intentar cambiar el estado de los pines, se imprime un mensaje de error.

En `start_motor()`, se intenta poner el pin del motor y la luz verde en estado alto, y la luz roja en estado bajo. Esto enciende el motor y la luz verde, y apaga la luz roja.

En `stop_motor()`, se intenta poner el pin del motor y la luz verde en estado bajo, y la luz roja en estado alto. Esto apaga el motor y la luz verde, y enciende la luz roja.

A screenshot of a code editor with a dark background and syntax highlighting. The code defines two functions: `start_motor()` and `stop_motor()`. `start_motor()` sets `pinM` and `pinLV` to `GPIO.HIGH` and `pinLR` to `GPIO.LOW`. `stop_motor()` sets `pinM` and `pinLV` to `GPIO.LOW` and `pinLR` to `GPIO.HIGH`. Both functions have an `except` block that prints an error message. The code is numbered from 1 to 15.

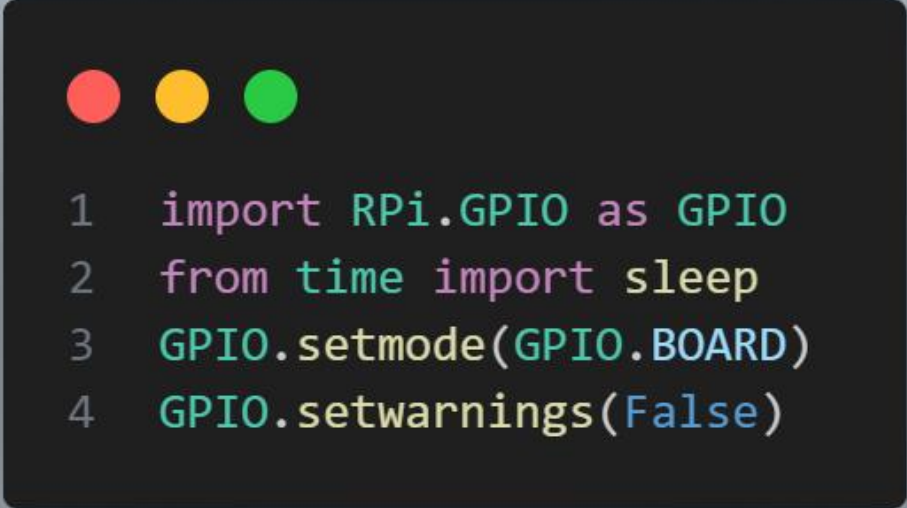
```
1 def start_motor():
2     try:
3         GPIO.output(pinM, GPIO.HIGH)
4         GPIO.output(pinLV, GPIO.HIGH)
5         GPIO.output(pinLR, GPIO.LOW)
6     except:
7         print("No se pudo encender el motor")
8
9 def stop_motor():
10     try:
11         GPIO.output(pinM, GPIO.LOW)
12         GPIO.output(pinLV, GPIO.LOW)
13         GPIO.output(pinLR, GPIO.HIGH)
14     except:
15         print("No se pudo apagar el motor")
```

Gates.py

Primero, se importa la función `sleep` del módulo `time`. Esta función se utiliza para hacer una pausa en la ejecución del programa durante un número especificado de segundos.

Luego, se configura el modo de los pines GPIO a GPIO.BOARD para referirse a los pines por el número del pin del conector físico.

La línea GPIO.setwarnings(False) se utiliza para desactivar las advertencias de GPIO.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains four lines of Python code for configuring the GPIO module.

```
1 import RPi.GPIO as GPIO
2 from time import sleep
3 GPIO.setmode(GPIO.BOARD)
4 GPIO.setwarnings(False)
```

La función put_angle toma un ángulo como entrada y calcula un valor de ciclo de trabajo para un servo. Si el ángulo está fuera del rango de 0 a 180 grados, la función devuelve False. De lo contrario, calcula un valor de ciclo de trabajo proporcional al ángulo y lo devuelve.




```
1 def put_angle(angle):  
2     if angle < 0 or angle > 180:  
3         return False  
4     starts = 4  
5     ends = 12.5  
6     ratio = (ends - starts)/100  
7     percentage = angle * ratio  
8     return starts + percentage
```

Las funciones `open_gates` y `close_gates` se utilizan para abrir y cerrar las compuertas, respectivamente. Ambas funciones configuran el pin 32 como salida, inician un objeto PWM en el pin 32 con una frecuencia de 50Hz, y luego cambian el ciclo de trabajo del PWM.

En `open_gates`, el ciclo de trabajo se establece en 3, lo que puede mover el servo a una posición que abre las compuertas. En `close_gates`, el ciclo de trabajo se establece en 12, lo que puede mover el servo a una posición que cierra las compuertas.

Después de cambiar el ciclo de trabajo, el programa se pausa durante 0.25 segundos para dar tiempo al servo para moverse, y luego se detiene el PWM.

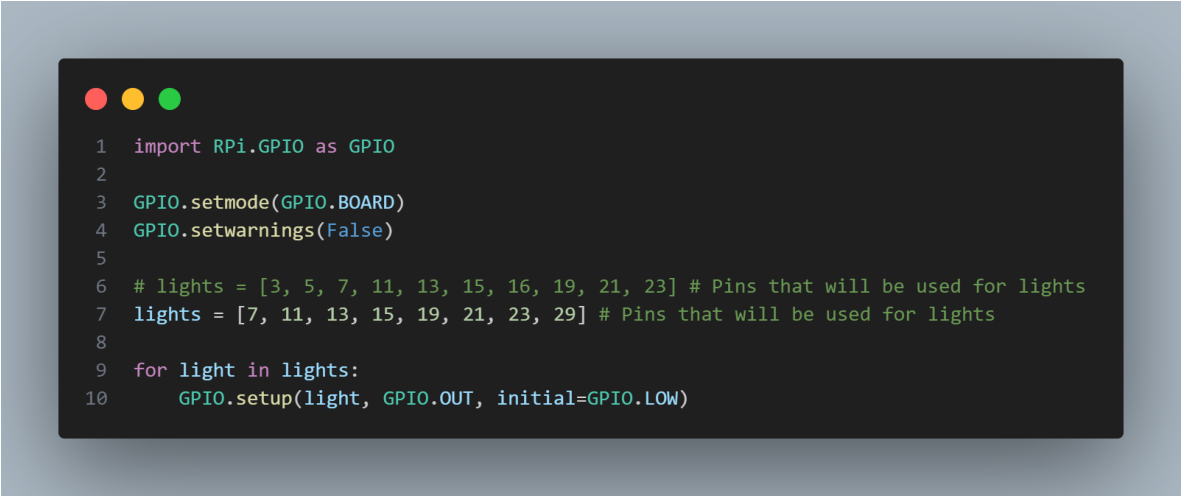


```
1  def open_gates():
2      GPIO.setup(32, GPIO.OUT)
3      p = GPIO.PWM(32, 50)
4      p.start(0)
5      p.ChangeDutyCycle(3)
6      sleep(0.25)
7      p.stop()
8
9  def close_gates():
10     GPIO.setup(32, GPIO.OUT)
11     p = GPIO.PWM(32, 50)
12     p.start(0)
13     p.ChangeDutyCycle(12)
14     sleep(0.25)
15     p.stop()
16
```

Lights.py


Primero, se configura el modo de los pines GPIO a GPIO.BOARD, lo que significa que los números de los pines se refieren a los números de los pines físicos de la placa. Luego, se desactivan las advertencias para evitar que se muestren mensajes de advertencia innecesarios.

Se define una lista llamada `lights` que contiene los números de los pines que se utilizarán para las luces. Luego, en un bucle `for`, se configura cada pin de luz como salida (`GPIO.OUT`) y se inicializa en estado bajo (`GPIO.LOW`), lo que significa que las luces estarán apagadas al inicio.

A screenshot of a terminal window with a dark background and light-colored text. The code is written in Python and is used to initialize GPIO pins for a Raspberry Pi. It includes comments in Spanish explaining the purpose of the code. The code is as follows:

```
1 import RPi.GPIO as GPIO
2
3 GPIO.setmode(GPIO.BOARD)
4 GPIO.setwarnings(False)
5
6 # lights = [3, 5, 7, 11, 13, 15, 16, 19, 21, 23] # Pins that will be used for lights
7 lights = [7, 11, 13, 15, 19, 21, 23, 29] # Pins that will be used for lights
8
9 for light in lights:
10     GPIO.setup(light, GPIO.OUT, initial=GPIO.LOW)
```

La función `turn_light(number, on)` se utiliza para encender o apagar una luz específica. El primer parámetro es el índice en la lista `lights` para la luz que se quiere controlar, y el segundo parámetro es un booleano que indica si la luz debe estar encendida (`True`) o apagada (`False`). Dentro de la función, se verifica si el número proporcionado está dentro del rango válido. Si es así, se cambia el estado de la luz correspondiente. Si el número es 10, se llama a la función `turn_all_lights(on)` para encender o apagar todas las luces. Si el número no es válido, se imprime un mensaje de error.




```

1  # First parameter is the index in the array for the light [number]
2  # Second parameter dictates if it is turned on or off [on]
3  def turn_light(number, on):
4      try:
5          number = int(number)
6          if 0 <= number <= 8:
7              if on:
8                  GPIO.output(lights[number], GPIO.HIGH)
9              else:
10                 GPIO.output(lights[number], GPIO.LOW)
11             elif number == 10:
12                 turn_all_lights(on)
13             else:
14                 print("No existe esa luz, intente con otra")
15         except:
16             print("No ingreso un numero valido")
17

```

La función `turn_all_lights(on)` se utiliza para encender o apagar todas las luces. Recibe un parámetro booleano que indica si las luces deben estar encendidas (True) o apagadas (False). Dentro de la función, se recorre la lista `lights` y se cambia el estado de cada luz según el valor del parámetro `on`.



```

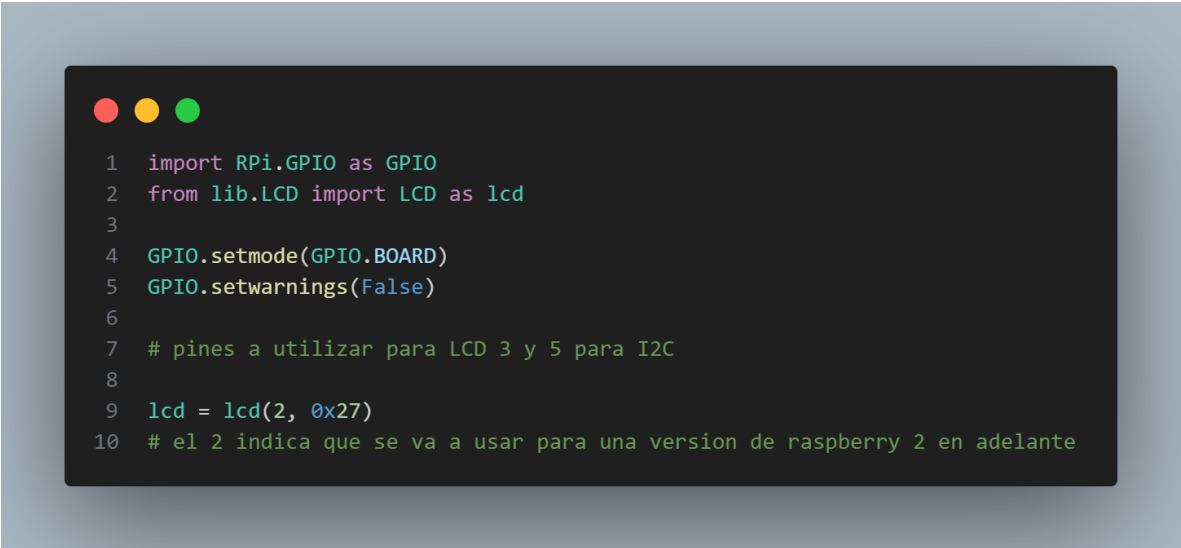
1  # Parameter on checks if it turn on or off.
2  def turn_all_lights(on):
3      if on:
4          for light in lights:
5              GPIO.output(light, GPIO.HIGH)
6      else:
7          for light in lights:
8              GPIO.output(light, GPIO.LOW)
9

```

Screen.py

Primero, importa la clase LCD del módulo lib.LCD y la renombra como lcd para su uso en el código. Luego, configura el modo de los pines GPIO de la Raspberry Pi para que se refieran por su número de pin físico en la placa (esto es lo que hace GPIO.setmode(GPIO.BOARD)). Desactiva las advertencias de GPIO con GPIO.setwarnings(False).

Después, crea una instancia de la clase lcd con los argumentos 2 y 0x27. El 2 indica que se va a usar para una versión de Raspberry Pi 2 en adelante. El 0x27 es probablemente la dirección I2C del LCD.



```
1 import RPi.GPIO as GPIO
2 from lib.LCD import LCD as lcd
3
4 GPIO.setmode(GPIO.BOARD)
5 GPIO.setwarnings(False)
6
7 # pines a utilizar para LCD 3 y 5 para I2C
8
9 lcd = lcd(2, 0x27)
10 # el 2 indica que se va a usar para una version de raspberry 2 en adelante
```

La función show_text(action) toma una cadena de texto como entrada, la divide en palabras y las muestra en la pantalla LCD. Primero, borra cualquier texto existente en la pantalla con lcd.clear(). Luego, divide la cadena de entrada en palabras individuales con action.split(" ").

El código luego itera sobre cada palabra en la lista de palabras. Si la longitud de la palabra y el texto actual en row1 es menor o igual a 15 y row1 aún no está llena (rowsFull es False), añade la palabra a row1. Si row1 está llena o la adición de la palabra haría que row1 exceda 15 caracteres, cambia rowsFull a True y comienza a añadir palabras a row2.

Finalmente, muestra row1 y row2 en la pantalla LCD en las líneas 1 y 2 respectivamente con lcd.text(row1, 1) y lcd.text(row2, 2).

```

1 def show_text(action):
2     lcd.clear()
3     texts = action.split(" ")
4     rowIsFull = False
5     row1=""
6     row2=""
7     for text in texts:
8         if (len(row1) + len(text) <= 15 and not rowIsFull):
9             row1 += text + " "
10        else:
11            rowIsFull = True
12            row2 += text + " "
13    lcd.text(row1, 1)
14    lcd.text(row2, 2)

```

Supervisor.py


El código comienza configurando el modo de los pines GPIO y desactivando las advertencias. Luego, se configuran tres pines como entradas, que se utilizarán para leer los estados de los sensores.

```

1 import RPi.GPIO as GPIO
2 from rpi_lcd import LCD
3 import threading
4 from Raspberry.Gates import open_gates, close_gates
5 from Raspberry.Lights import turn_light
6 from Raspberry.ConvBelt import start_motor, stop_motor
7 from Raspberry.Alarm import turn_on_buzzer_with_timmer
8 from Raspberry.Clients import show_binary
9 from signal import signal, SIGTERM, SIGHUP
10 from time import sleep
11
12 GPIO.setmode(GPIO.BOARD)
13 GPIO.setwarnings(False)
14
15 GPIO.setup(18, GPIO.IN) #primer sensor (este irá en la entrada)
16 GPIO.setup(22, GPIO.IN) #segundo sensor (este irá adentro)
17 GPIO.setup(31, GPIO.IN) #sensor perimetral
18 #GPIO.setup(33, GPIO.IN) #sensor de alarma

```

La función `safe_exit` se define para manejar las señales `SIGTERM` y `SIGHUP`, que son señales del sistema operativo que indican que el programa debe terminar. Cuando se recibe una de estas señales, el programa se cierra con un código de salida de 1.



```
1 def safe_exit(signum, frame):  
2     exit(1)  
3  
4 signal(SIGTERM, safe_exit)  
5 signal(SIGHUP, safe_exit)
```

La clase `butler` es el núcleo del sistema de supervisión. En su método `__init__`, inicializa un objeto `LCD`, y establece contadores para los clientes, el estado de las puertas y las luces.



```
1  def __init__(self):  
2      self.lcd = LCD()  
3      self.clients = 0  
4      self.gate_state = 0  
5      self.all_lights = 0
```

El método `light_action` controla las luces. Si se pasa -1 como el número de luz, todas las luces se encienden o apagan. De lo contrario, se enciende o apaga la luz especificada.



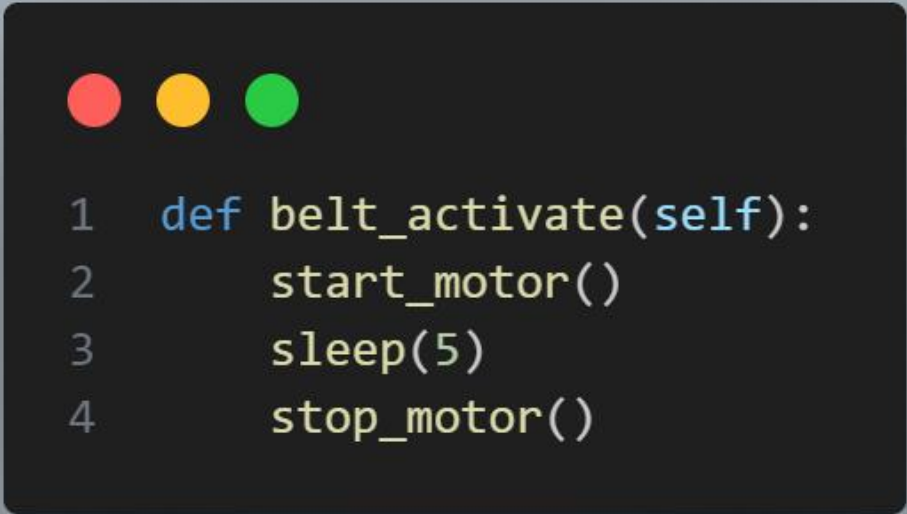
```
1 def light_action(self, lights, state):  
2     if lights == -1:  
3         turn_light(10, state)  
4         self.all_lights = state  
5     else:  
6         turn_light(lights, state)
```

El método `gate_action` controla las puertas. Si se pasa 1 como estado, las puertas se cierran. De lo contrario, las puertas se abren.



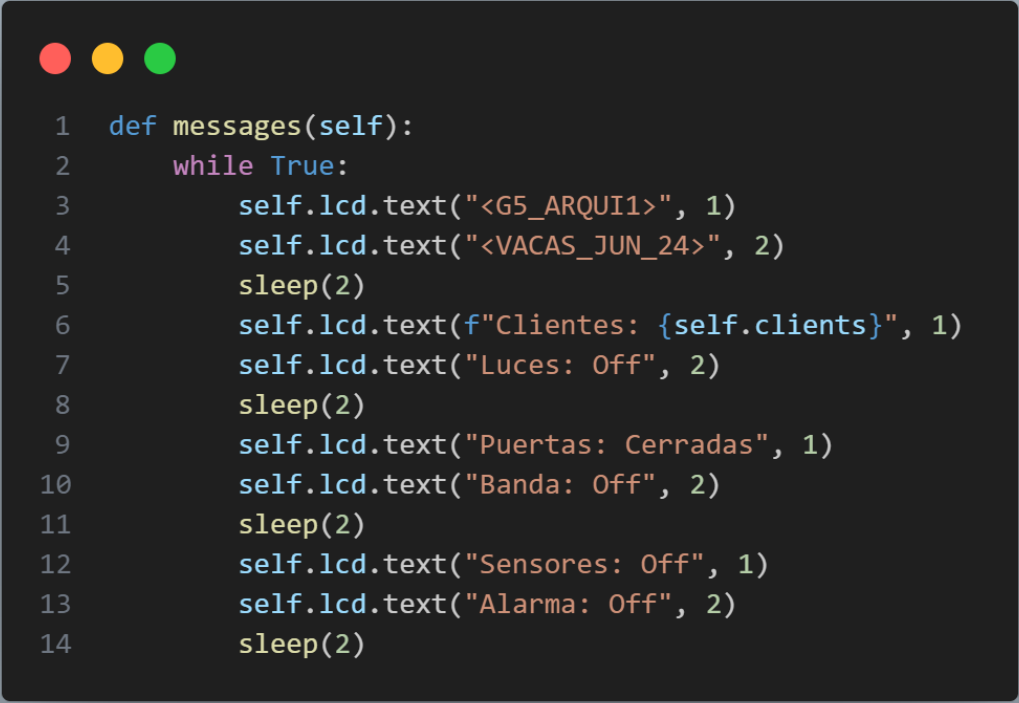
```
1
2  def gate_action(self, state):
3      if state == 1:
4          close_gates()
5          self.gate_state = 0
6      else:
7          open_gates()
8          self.gate_state = 1
```

El método `belt_activate` inicia el motor de la cinta transportadora, espera 5 segundos y luego lo detiene.




```
1  def belt_activate(self):  
2      start_motor()  
3      sleep(5)  
4      stop_motor()
```

El método messages muestra mensajes en el LCD. Los mensajes se actualizan cada 2 segundos.




```
1  def messages(self):
2      while True:
3          self.lcd.text("<G5_ARQUI1>", 1)
4          self.lcd.text("<VACAS_JUN_24>", 2)
5          sleep(2)
6          self.lcd.text(f"Clientes: {self.clients}", 1)
7          self.lcd.text("Luces: Off", 2)
8          sleep(2)
9          self.lcd.text("Puertas: Cerradas", 1)
10         self.lcd.text("Banda: Off", 2)
11         sleep(2)
12         self.lcd.text("Sensores: Off", 1)
13         self.lcd.text("Alarma: Off", 2)
14         sleep(2)
```

El método `clients_counter` cuenta el número de clientes. Utiliza dos sensores para determinar si un cliente está entrando o saliendo, y ajusta el contador de clientes en consecuencia.



```
1  def clients_counter(self):
2      a_state = 0
3      a_prev = 0
4      b_state = 0
5      b_prev = 0
6
7      while True:
8          a_state = not GPIO.input(18)
9          b_state = not GPIO.input(22)
10
11         if a_state == 1 and a_prev == 0:
12             if b_prev == 1:
13                 self.clients = self.clients - 1
14                 b_prev = 0
15             else:
16                 a_prev = 1
17                 print("Contador: ", self.clients)
18
19         if b_state == 1 and b_prev == 0:
20             if a_prev == 1:
21                 self.clients = self.clients + 1
22                 a_prev = 0
23             else:
24                 b_prev = 1
25                 print("Contador: ", self.clients)
26         show_binary(self.clients)
27         sleep(1)
```

El método `perimeter_alarm` activa una alarma si se detecta una entrada en el perímetro.




```

1
2  def perimeter_alarm(self):
3      while True:
4          if GPIO.input(31):
5              turn_on_buzzer_with_timmer(10)

```

Finalmente, el método `start_supervisor` inicia tres hilos para ejecutar los métodos `messages`, `clients_counter` y `perimeter_alarm` en paralelo.



```

1  def start_supervisor(self):
2      th1 = threading.Thread(target=self.messages, args=())
3      th2 = threading.Thread(target=self.clients_counter, args=())
4      th3 = threading.Thread(target=self.perimeter_alarm, args=())
5      th1.start()
6      th2.start()
7      th3.start()

```

Parte del front

Solo se van a explicar cosas sencillas, para esto se utilizo django y flask (es el archivo `app.py` que se explico al principio)

Views.py

Las clases `MainData` y `Light` son estructuras de datos. `MainData` almacena información sobre el número total de clientes que entran y salen, el estado de la cinta transportadora, la puerta y la alarma. `Light` representa una luz individual con un identificador y un estado.

```
1 import requests
2 from django.shortcuts import render
3 from django.http import JsonResponse
4
5 # Create your views here.
6 class MainData:
7     def __init__(self, totalClientIn, totalClientOut, clientsIn, clientsOut, conditionConveyorBelt, gate, alarm):
8         self.totalClientsIn = totalClientIn
9         self.totalClientsOut = totalClientOut
10        self.clientsIn = clientsIn
11        self.clientsOut = clientsOut
12        self.conditionConveyorBelt = conditionConveyorBelt
13        self.gate = gate
14        self.alarm = alarm
15
16
17 class Light:
18     def __init__(self, id, state):
19         self.id = id
20         self.state = state
```

La función loadData se utiliza para cargar datos desde varias URL, cada una de las cuales devuelve un estado o un recuento de personas. Los datos se almacenan en la instancia global de MainData llamada data.


```

1  def loadData():
2      print("Loading data")
3      global data
4
5      #Load actual people
6      response = requests.get('http://127.0.0.1:5000/actual_people')
7      print(response.json())
8      data.totalClientsIn = response.json()['actual_people']
9
10     #counter people
11     response = requests.get('http://127.0.0.1:5000/counter_people')
12     print(response.json())
13     data.clientsIn = response.json()['counter_people']
14
15     #transport band
16     response = requests.get('http://127.0.0.1:5000/transport_band_status')
17     print(response.json())
18     state = response.json()['state']
19     if state == 1:
20         data.conditionConveyorBelt = True
21     else:
22         data.conditionConveyorBelt = False
23
24
25     #gate
26     response = requests.get('http://127.0.0.1:5000/garage_status')
27     print(response.json())
28     state = response.json()['state']
29     if state == 1:
30         data.gate = True
31     else:
32         data.gate = False
33
34     #alarm
35     response = requests.get('http://127.0.0.1:5000/alarm_status')
36     print(response.json())
37     state = response.json()['state']
38     if state == 1:
39         data.gate = True
40     else:
41         data.gate = False
42

```

La función test es una vista que carga los datos utilizando loadData y luego renderiza una plantilla HTML llamada 'test.html', pasando los datos de las luces y la instancia de MainData.

```
1  #crear una lista de luces
2  lights = []
3
4  for i in range(10):
5      lights.append(Light(i, False))
6
7  data = MainData(0, 0, 0, 0, False, False, False)
8
9  def test(request):
10     global lights
11     global data
12     loadData()
13     return render(request, 'test.html', { 'lights': lights, 'data': data})
```

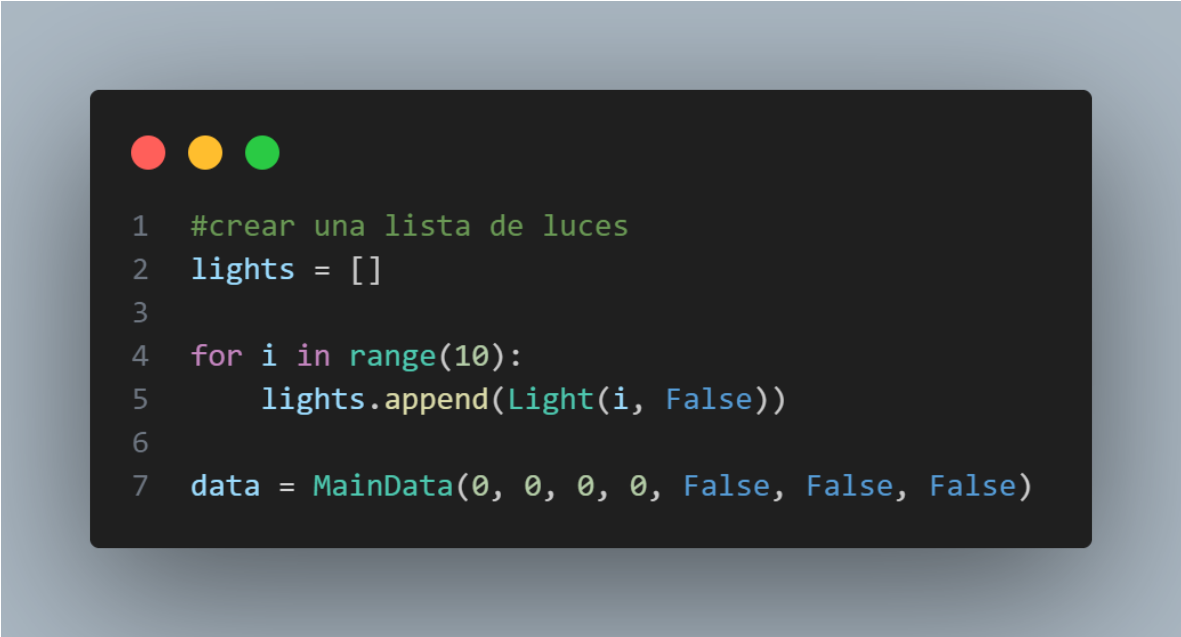
Las funciones `set_all_lights_states`, `set_lights_states`, `set_transport_band` y `set_garage` son vistas que manejan las solicitudes POST para cambiar el estado de las luces, la cinta transportadora y la puerta, respectivamente. Cada una de estas funciones recibe una acción ('on' o 'off') y realiza una solicitud POST a una URL específica para cambiar el estado correspondiente. Si la solicitud POST es exitosa, se renderiza la plantilla 'test.html' con los datos actualizados. Si la solicitud POST falla, se devuelve una respuesta JSON con un mensaje de error.

```

1  def set_lights_states(request):
2      global data
3      global lights
4      if request.method == 'POST':
5          print("Request received")
6          id_light = request.POST.get('id')
7          action = request.POST.get('action')
8          id_light = int(id_light)
9
10         print(f"ID: {id_light}, Action: {action}")
11         if action == 'on':
12             lights[id_light].state = True
13             dataResponse = {'num': id_light, 'state': 1}
14         elif action == 'off':
15             lights[id_light].state = False
16             dataResponse = {'num': id_light, 'state': 0}
17
18         response = requests.post('http://127.0.0.1:5000/light', json=dataResponse)
19         if response.status_code == 200:
20
21             return render(request, 'test.html', { 'lights': lights, 'data': data})
22         else:
23             return JsonResponse({'message': 'Failed to set lights state'}, status=500)
24
25     else:
26         return JsonResponse({'message': 'Invalid request method'}, status=400)
27
28 def set_transport_band(request):
29     global data
30     if request.method == 'POST':
31         print("Request received")
32         action = request.POST.get('action')
33         if action == 'on':
34             data.conditionConveyorBelt = True
35             dataResponse = {'state': 1}
36         elif action == 'off':
37             data.conditionConveyorBelt = False
38             dataResponse = {'state': 0}
39
40         response = requests.post('http://127.0.0.1:5000/transport_band', json=dataResponse)
41         if response.status_code == 200:
42             return render(request, 'test.html', { 'lights': lights, 'data': data})
43         else:
44             return JsonResponse({'message': 'Failed to set transport band state'}, status=500)
45
46     else:
47         return JsonResponse({'message': 'Invalid request method'}, status=400)
48
49 def set_garage(request):
50     global data
51     if request.method == 'POST':
52         print("Request received")
53         action = request.POST.get('action')
54         if action == 'on':
55             data.gate = True
56             dataResponse = {'state': 1}
57         elif action == 'off':
58             data.gate = False
59             dataResponse = {'state': 0}
60
61         response = requests.post('http://127.0.0.1:5000/Garage', json=dataResponse)
62         if response.status_code == 200:
63             return render(request, 'test.html', { 'lights': lights, 'data': data})
64         else:
65             return JsonResponse({'message': 'Failed to set garage state'}, status=500)
66     else:
67         return JsonResponse({'message': 'Invalid request method'}, status=400)
68

```

Por último, se crea una lista de luces `lights` con 10 luces, todas inicialmente apagadas, y una instancia de `MainData` llamada `data` con todos los valores inicialmente en cero o `False`.



```
1  #crear una lista de luces
2  lights = []
3
4  for i in range(10):
5      lights.append(Light(i, False))
6
7  data = MainData(0, 0, 0, 0, False, False, False)
```

Urls.py

En las primeras líneas, se importan varias vistas desde `webApp.views`. Las vistas son funciones que toman una solicitud web y devuelven una respuesta. En este caso, las vistas importadas son `test`, `set_all_lights_states`, `set_lights_states`, `set_transport_band` y `set_garage`.



```
1 from django.contrib import admin
2 from django.urls import path
3 from webApp.views import test
4
5 from webApp.views import set_all_lights_states
6
7 from webApp.views import set_lights_states
8 from webApp.views import set_transport_band
9 from webApp.views import set_garage
```

A continuación, se define una lista llamada `urlpatterns`. Esta lista es una parte crucial de la configuración de URL de Django. Cada elemento de la lista es una llamada a la función `path()`, que crea una instancia de `URLPattern`.

La función `path()` toma tres argumentos: una cadena que define la ruta URL, una vista que se invocará cuando se solicite esa URL, y opcionalmente un nombre para la ruta que se puede usar para referirse a ella en otras partes del código.

Por ejemplo, `path('set_all_lights_states/', set_all_lights_states, name='set_all_lights_states')` define una ruta URL que, cuando se solicita, invocará la vista `set_all_lights_states`. También se le da un nombre, `set_all_lights_states`, que se puede usar para referirse a esta ruta en otras partes del código, como en las plantillas de Django.

En resumen, este código define las rutas URL para una aplicación web Django, mapeando URL específicas a sus correspondientes vistas.



```
1 urlpatterns = [  
2     path('admin/', admin.site.urls),  
3     path('', test),  
4     path('set_all_lights_states/', set_all_lights_states, name='set_all_lights_states'),  
5     path('set_lights_states/', set_lights_states, name='set_lights_states'),  
6     path('set_transport_band/', set_transport_band, name='set_transport_band'),  
7     path('set_garage/', set_garage, name='set_garage'),  
8  
9 ]
```