Robotics Group IV

MILESTONE: 3

Karabo Linala (577593)
Kelo Letsoal (577613)
Johannes Gerhardus Van Wyk (578007)
Kamahelo Mototo (577715)
Jordan Miguel Barradas (577848)
Jamie Sepp Butow (577588)

Executive Summary

The following report details the groups effort in milestone 3 of the robotic arm project. The report begins where the last milestone left off, namely, at the implementation of the user interface (mobile app) and the necessary API (application program interface) connection between it and the robot arm system. The report elaborates on the finished mobile application, showing the user interface and how it displays different statuses of the API and the robot arm. It also details how this mobile application connects to the API express server and the general architecture of the API system. The report then touches on implementation details of the API connection on the robot side. Moreover, the report details the object detection/ computer vision implementation of the robot system. It includes various code snippets showing the detection logic and the various classes used to implement the computer vision requirement of the project. Included with these snippets are explanations of the classes and what each snippet is detailing. Lastly, the report touches on the robotic arm movement logic which has been rewritten in python to help accommodate and integrate with the rest of the robotic system. As with the other code snippets, explanations of them are included with the snippets to help explain the logic and functionality.

Table of Contents

Table of Figures

Section A – Continuation of Mobile App API User Interface

This section will pick up where the last milestone ended in regard to the mobile app and API (application program interface – user story 5) which will work together to act as the user interface to the robotics system. Last milestone (milestone 2 document) up to this point covered the wire frame medium fidelity prototype of the mobile application and some pseudo code for the API. The following subsections will clarify changes to user stories necessary to make project deadlines, the high level design of the API, and then implementation of both the mobile application as well as the application programming interface used by the robotic system.

User Story Changes

Sprint 2 (Phone API User Interface) Sprint Backlog		
User Story Number	**Name**	**Description**
5	**Mobile App Control**	Allows the sorting process to be started and stopped remotely through a mobile interface.
~~**6**~~	~~**Object Detection Feedback**~~	Allows users to receive alerts ~~about object detection remotely~~ through a mobile interface.
~~**7**~~	~~**Error Handling and Alerts**~~	Allows users to receive alerts about errors detected during the sorting process remotely though a mobile interface

The focus of this sprint is on user story 5 – Mobile Application Control, as this is what is strictly necessary for the user to interact with the robotics system. This is done in order to ensure that the project meets the deadline currently set for it. At a later stage, user stories 6 and 7 can be addressed and brought to full functionality however as they are “nice to have” functionality this is less of a priority than user story 5 which is user critical, furthermore, user stories 6 and 7 will be built off of user story 5’s functionality, again highlighting the importance of it to the system.

User Interface High Level Design

The following section will detail the overall design and specify the components used to bring the user interface of the robotics system to fruition:

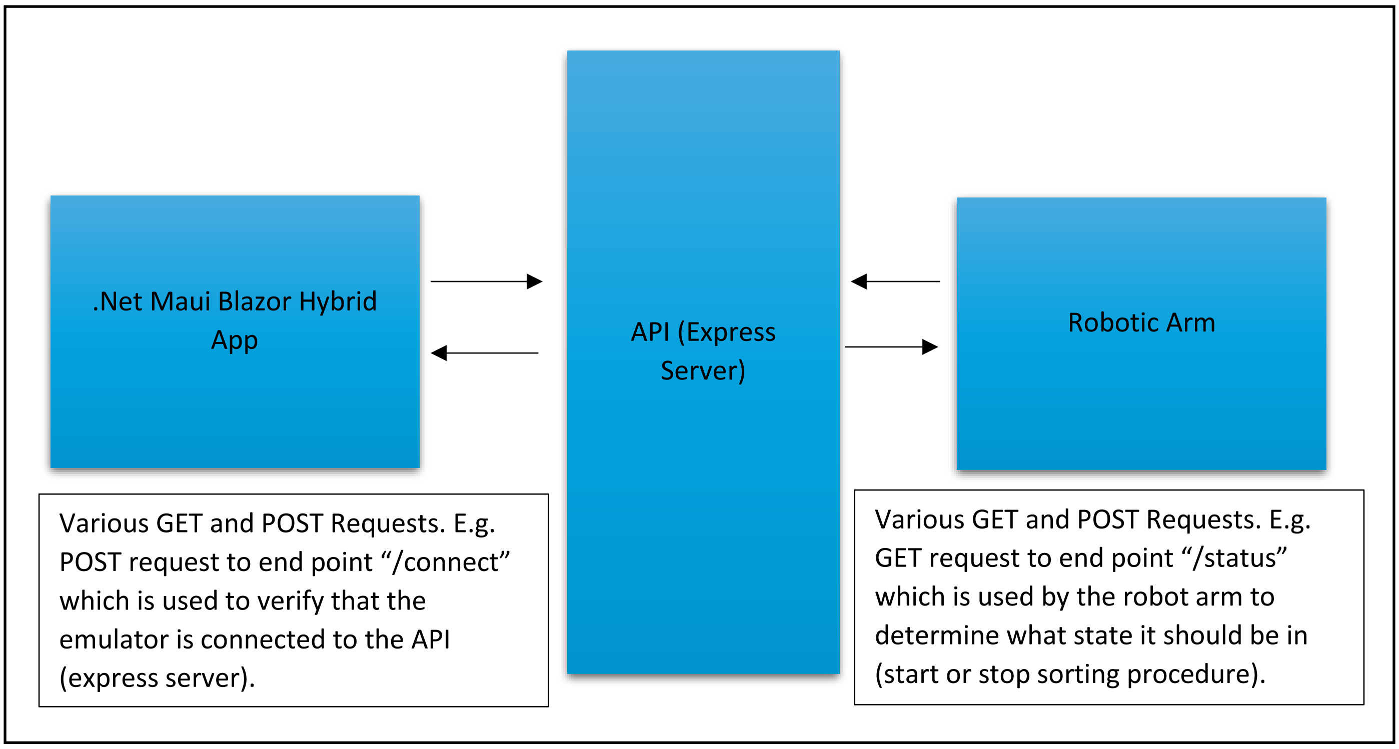

.Net Maui Blazor Hybrid App
API (Express Server)
Robotic Arm
Various GET and POST Requests. E.g. POST request to end point “/connect” which is used to verify that the emulator is connected to the API (express server).
Various GET and POST Requests. E.g. GET request to end point “/status” which is used by the robot arm to determine what state it should be in (start or stop sorting procedure).

Figure 1: High Level Overview of API and User Interface Architecture

The following sections will describe each of the above components in detail, highlighting important parts of each of them and how together the system achieves loose coupling but high cohesion.

.Net Maui Blazor Hybrid App

.Net Maui Blazor Hybrid App – emulator which emulates a google pixel 5 smart phone and is what is used to support the sorting app. Written in C#, this is the component which delivers the user interface to the user. It consists of a start and stop button as well as areas which detail the current status the app (if it’s connected to the API and thus the robot arm or not) and the status of the robot arm (i.e. “stopped” or “running”).
Using this interface a user is able to start the sorting process, stop the sorting process as well as view the status of the app’s connection to the API as well as the current status of the robot arm. This system although simplistic gives the user a way to interface with the system.

The following will detail with the aid of diagrams important aspects of the app:

- Every 2 seconds the app checks for the status of the API ("online" or "offline") and if the express server (or API) is running it automatically connects to it and displays the connection status to the user:

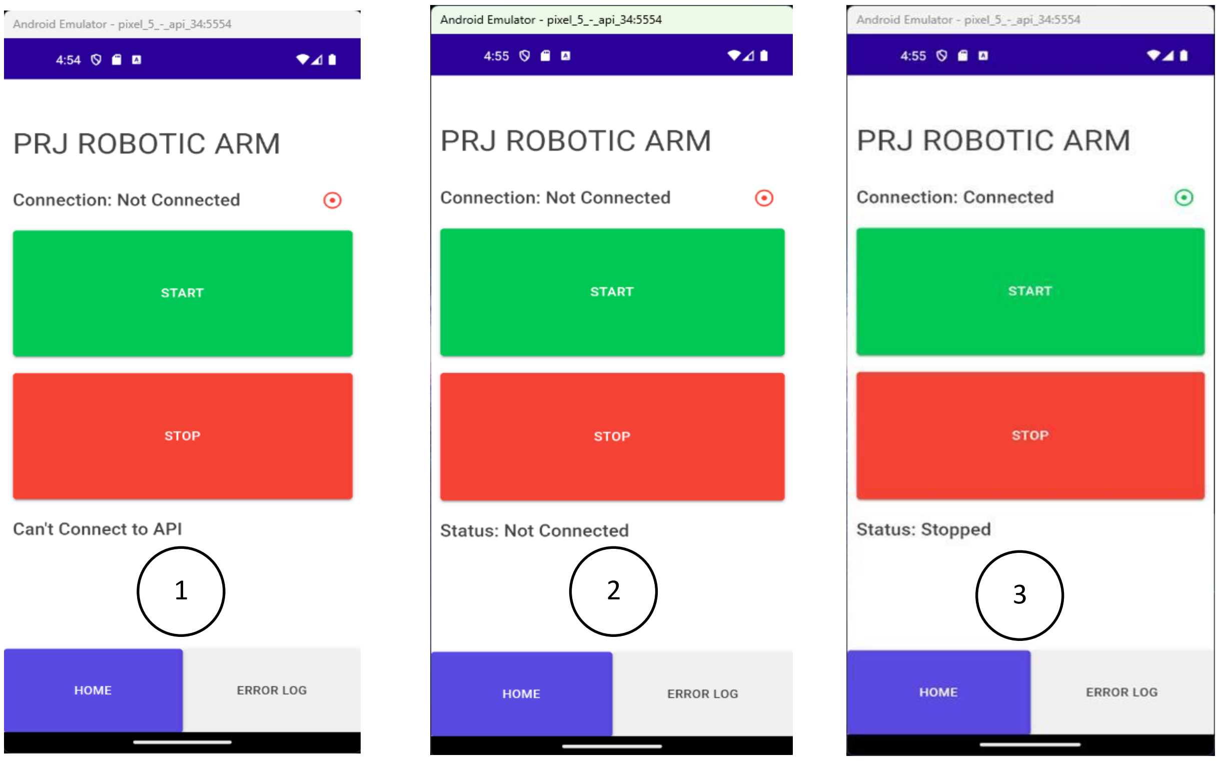


Figure 2: Diagram showing App UI - (1) when API and robot arm not connected, (2) API connected but robot arm not connected, (3) API and robot arm connected

- When start button is clicked, a POST http request is made to the express server and the robot arm status is changed from "stopped" to "running" and if successful is specified on the "Status" label of the UI. Similarly, when the stop button is clicked, a POST http request is again made to the express server and the robot arm status is changed from "running" to "stopped" and this change is also reflected on the app UI:

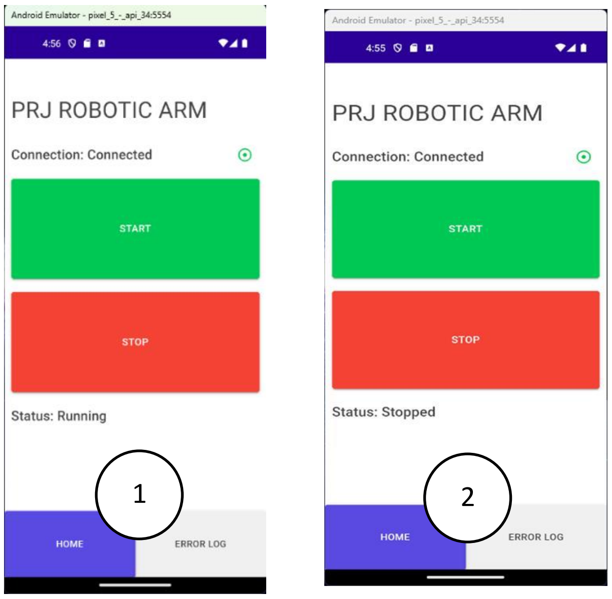
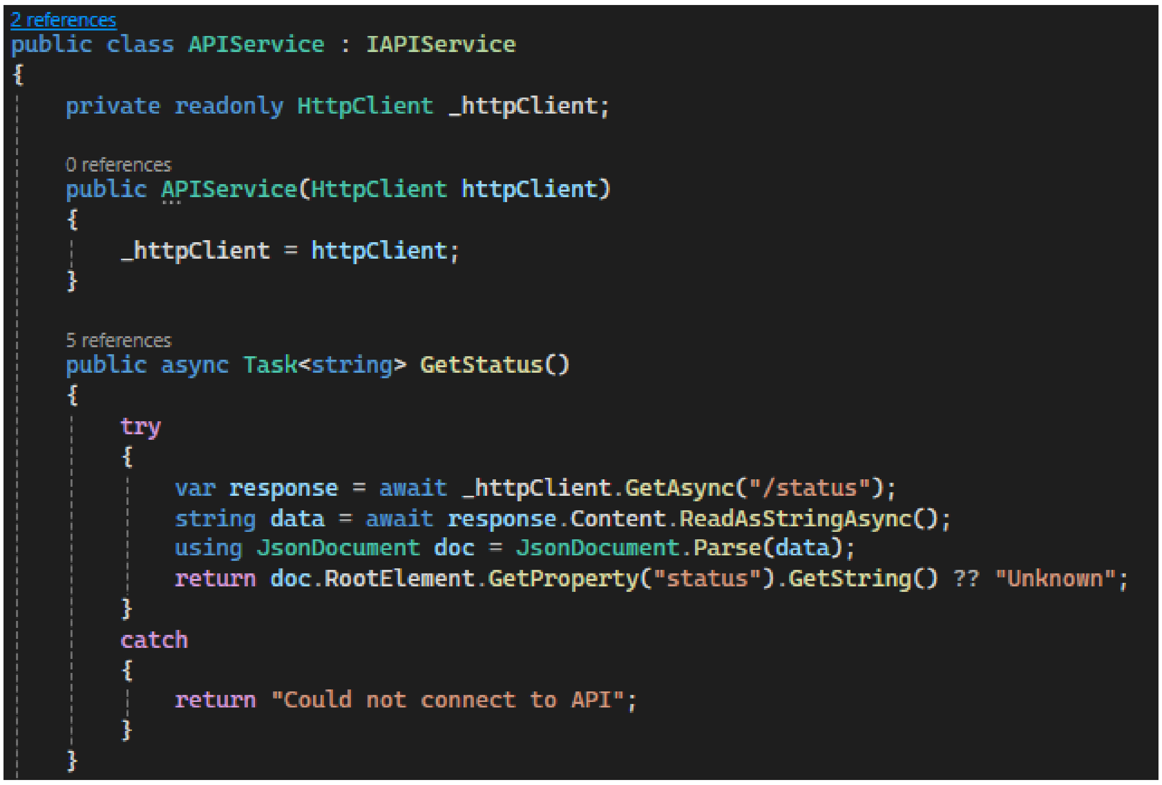


Figure 3: Diagram showing UI when (1) Start button is clicked and (2) Stop button is clicked

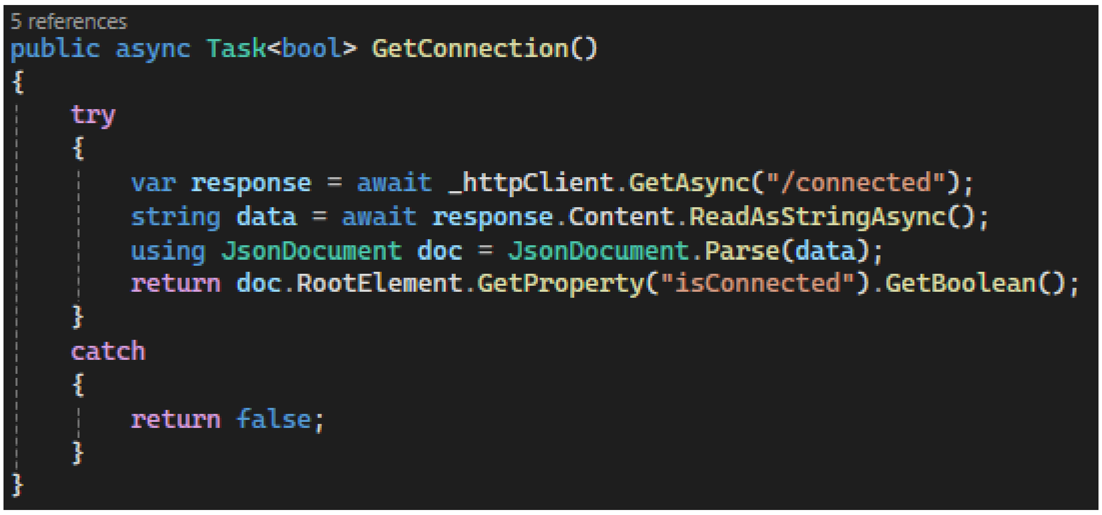
Code snippets of important parts of the .Net Maui Blazor Hybrid App:



```
2 references
public class APIService : IAPIService
{
    private readonly HttpClient _httpClient;

    0 references
    public APIService(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    5 references
    public async Task<string> GetStatus()
    {
        try
        {
            var response = await _httpClient.GetAsync("/status");
            string data = await response.Content.ReadAsStringAsync();
            using JsonDocument doc = JsonDocument.Parse(data);
            return doc.RootElement.GetProperty("status").GetString() ?? "Unknown";
        }
        catch
        {
            return "Could not connect to API";
        }
    }
```

Figure 4: Code Snippet of APIService class

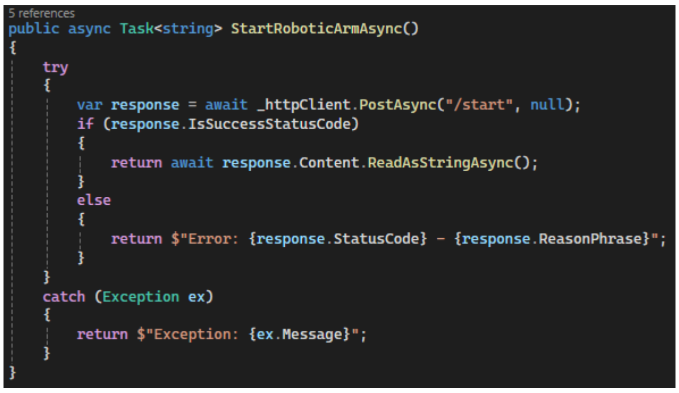
The above method is tasked with sending a GET http request to the endpoint "/status" of the express API which returns the current status of the robot arm, either "Stopped" or "Running".



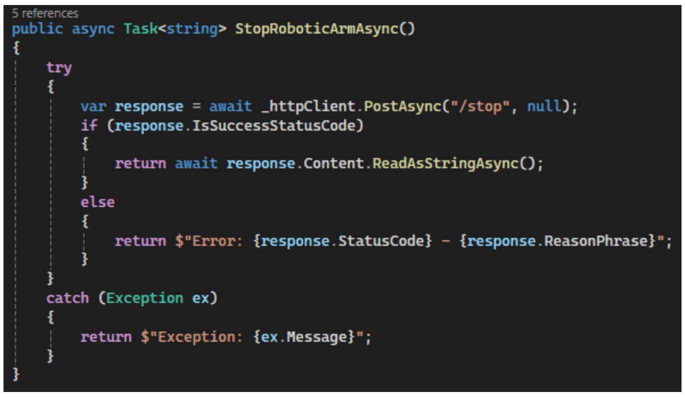
```
5 references
public async Task<bool> GetConnection()
{
    try
    {
        var response = await _httpClient.GetAsync("/connected");
        string data = await response.Content.ReadAsStringAsync();
        using JsonDocument doc = JsonDocument.Parse(data);
        return doc.RootElement.GetProperty("isConnected").GetBoolean();
    }
    catch
    {
        return false;
    }
}
```

Figure 5: Code Snippet of GetConnection() method

The above method is tasked with sending a GET http request to the endpoint "/connected" of the express API which returns the current connection state of the robot arm, either "true" or "false".



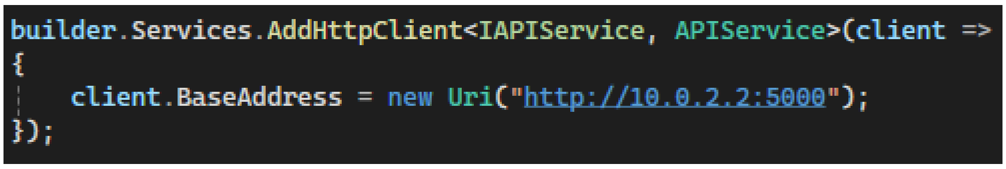
```
5 references
public async Task<string> StartRoboticArmAsync()
{
    try
    {
        var response = await _httpClient.PostAsync("/start", null);
        if (response.IsSuccessStatusCode)
        {
            return await response.Content.ReadAsStringAsync();
        }
        else
        {
            return $"Error: {response.StatusCode} - {response.ReasonPhrase}";
        }
    }
    catch (Exception ex)
    {
        return $"Exception: {ex.Message}";
    }
}
```



```
5 references
public async Task<string> StopRoboticArmAsync()
{
    try
    {
        var response = await _httpClient.PostAsync("/stop", null);
        if (response.IsSuccessStatusCode)
        {
            return await response.Content.ReadAsStringAsync();
        }
        else
        {
            return $"Error: {response.StatusCode} - {response.ReasonPhrase}";
        }
    }
    catch (Exception ex)
    {
        return $"Exception: {ex.Message}";
    }
}
```

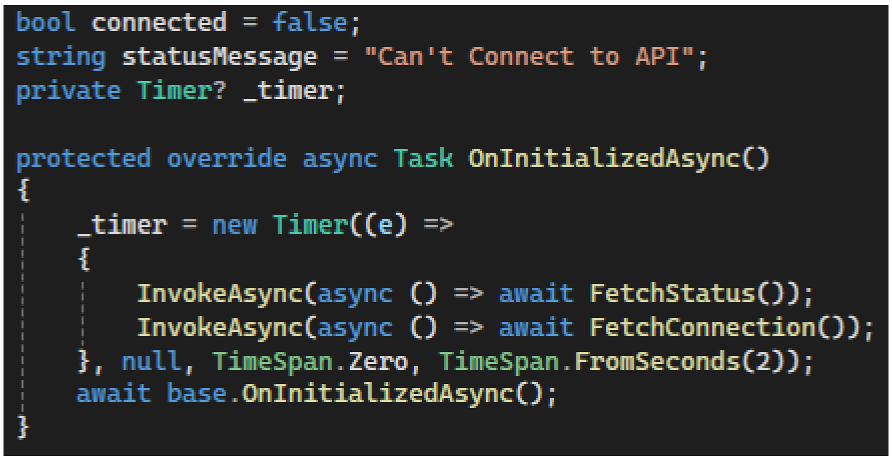
Figure 6: Code Snippets of StartRoboticArmSync() and StopRoboticArmSync() methods

The above methods are tasked with sending POST http requests to the express server to the endpoints "/start" and "/stop" respectively in order to begin and end the sorting process.



```
builder.Services.AddHttpClient<IAPIService, APIService>(client =>
{
    client.BaseAddress = new Uri("http://10.0.2.2:5000");
});
```

Figure 7: Code Snippet Showing HTTP Client Set Up



```
bool connected = false;
string statusMessage = "Can't Connect to API";
private Timer? _timer;

protected override async Task OnInitializedAsync()
{
    _timer = new Timer((e) =>
    {
        InvokeAsync(async () => await FetchStatus());
        InvokeAsync(async () => await FetchConnection());
    }, null, TimeSpan.Zero, TimeSpan.FromSeconds(2));
    await base.OnInitializedAsync();
}
```

Figure 8: Code Snippet Showing Home Page Initialization

The above method is tasked with running every 2 seconds to get and display the current status of the API (connected or not connected) and the status of the robot arm (stopped or running or not connected).

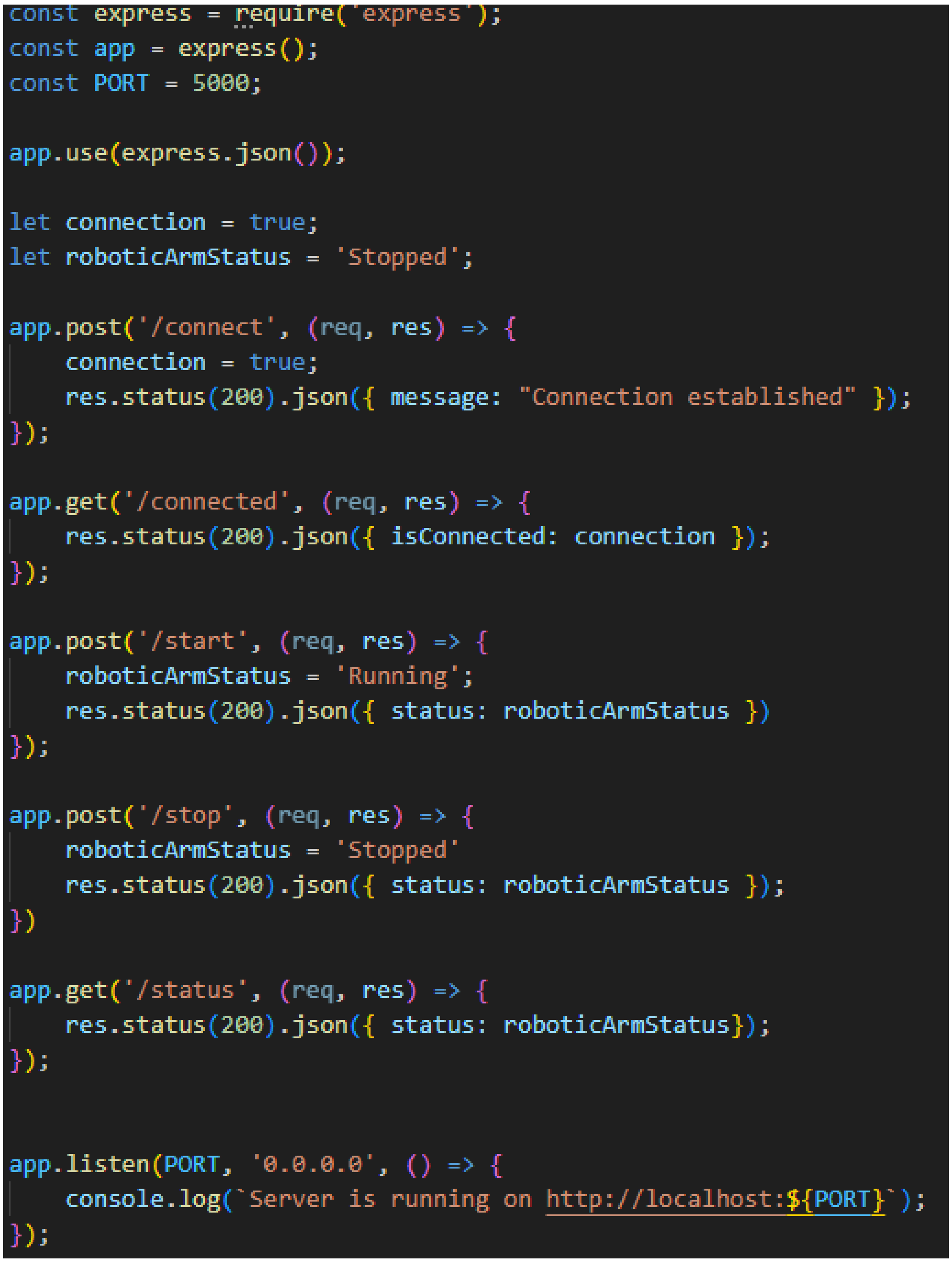
System API – Express Server

The express server made using Node.JS acts as the application programming interface of the system, allowing both the mobile app and the robotic arm to communicate with each other. Written in JavaScript, it contains a serious of endpoints which are accessed by both the mobile app and the robot arm in order to implement the start and stop logic, thereby allowing the user through the user interface (mobile app) the ability to control the robotic system.

The express server allows for requests to the following endpoints:

- POST "/connect" – sets the connection field of the server to true and is used by robot arm when it connects to the API.
- GET "/connection" – gets the current value of the connection field of the server (true or false) and is used by the mobile app to inform users of the current status of the connection of the robot arm (connected or not connected).
- GET "/status" – gets the current value of the RobotArmStatus field of the server ("Running" or "Stopped") and is used by the mobile to inform users of the current sorting status of the robot arm. It is also used by the robot arm to establish whether it should begin or stop sorting.
- POST "/start" – sets the value of the RobotArmStatus field to "Running". When the robot arm (in a loop continuously checking) uses the GET "/status" end point and it is set to "Running" it will begin the sorting process.
- POST "/stop" – sets the value of the RobotArmStatus field to "Stopped". When the robot arm (in a loop continuously checking) uses the GET "/status" end point and it is set to "Stopped" it will end / stop the sorting process.

The express application makes use of JSON (Java Script Object Notation) for any and all messaging that is required.

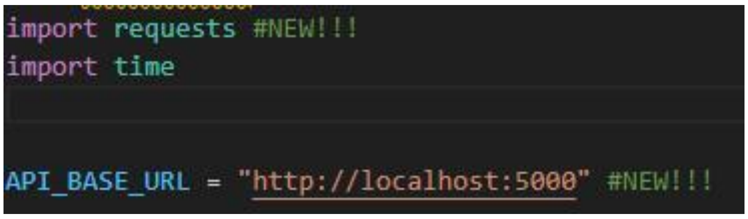


```javascript
const express = require('express');
const app = express();
const PORT = 5000;

app.use(express.json());

let connection = true;
let roboticArmStatus = 'Stopped';

app.post('/connect', (req, res) => {
    connection = true;
    res.status(200).json({ message: "Connection established" });
});

app.get('/connected', (req, res) => {
    res.status(200).json({ isConnected: connection });
});

app.post('/start', (req, res) => {
    roboticArmStatus = 'Running';
    res.status(200).json({ status: roboticArmStatus })
});

app.post('/stop', (req, res) => {
    roboticArmStatus = 'Stopped'
    res.status(200).json({ status: roboticArmStatus });
})

app.get('/status', (req, res) => {
    res.status(200).json({ status: roboticArmStatus});
});


app.listen(PORT, '0.0.0.0', () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

Figure 9: Code Snippet Showing Express API

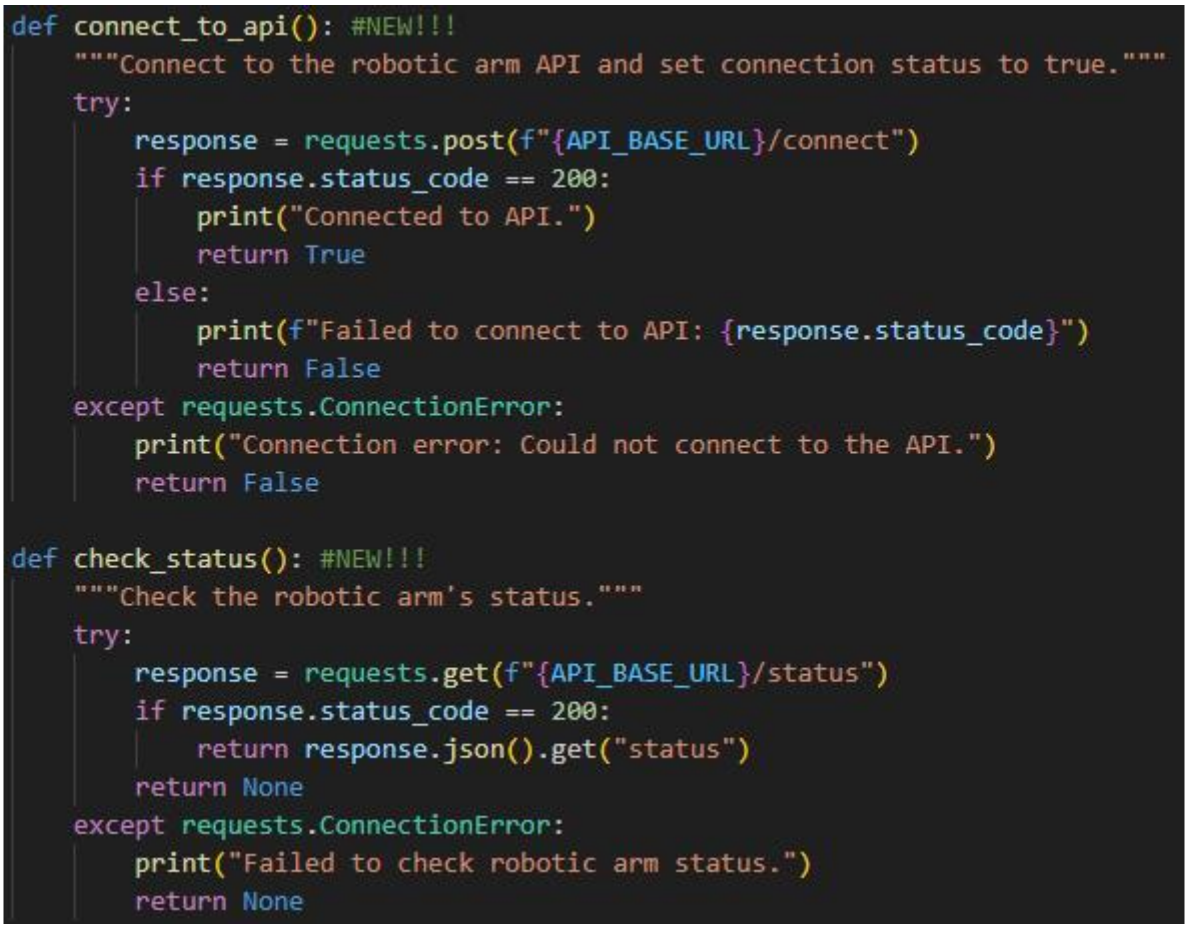
Robot Arm API Integration

The following section will detail the API integration on the robot side which is done in python.



```python
import requests #NEW!!!
import time


API_BASE_URL = "http://localhost:5000" #NEW!!!
```

Figure 10: Code snippet showing import of request library and the setting of a base URL



```python
def connect_to_api(): #NEW!!!
    """Connect to the robotic arm API and set connection status to true."""
    try:
        response = requests.post(f"{API_BASE_URL}/connect")
        if response.status_code == 200:
            print("Connected to API.")
            return True
        else:
            print(f"Failed to connect to API: {response.status_code}")
            return False
    except requests.ConnectionError:
        print("Connection error: Could not connect to the API.")
        return False

def check_status(): #NEW!!!
    """Check the robotic arm's status."""
    try:
        response = requests.get(f"{API_BASE_URL}/status")
        if response.status_code == 200:
            return response.json().get("status")
        return None
    except requests.ConnectionError:
        print("Failed to check robotic arm status.")
        return None
```

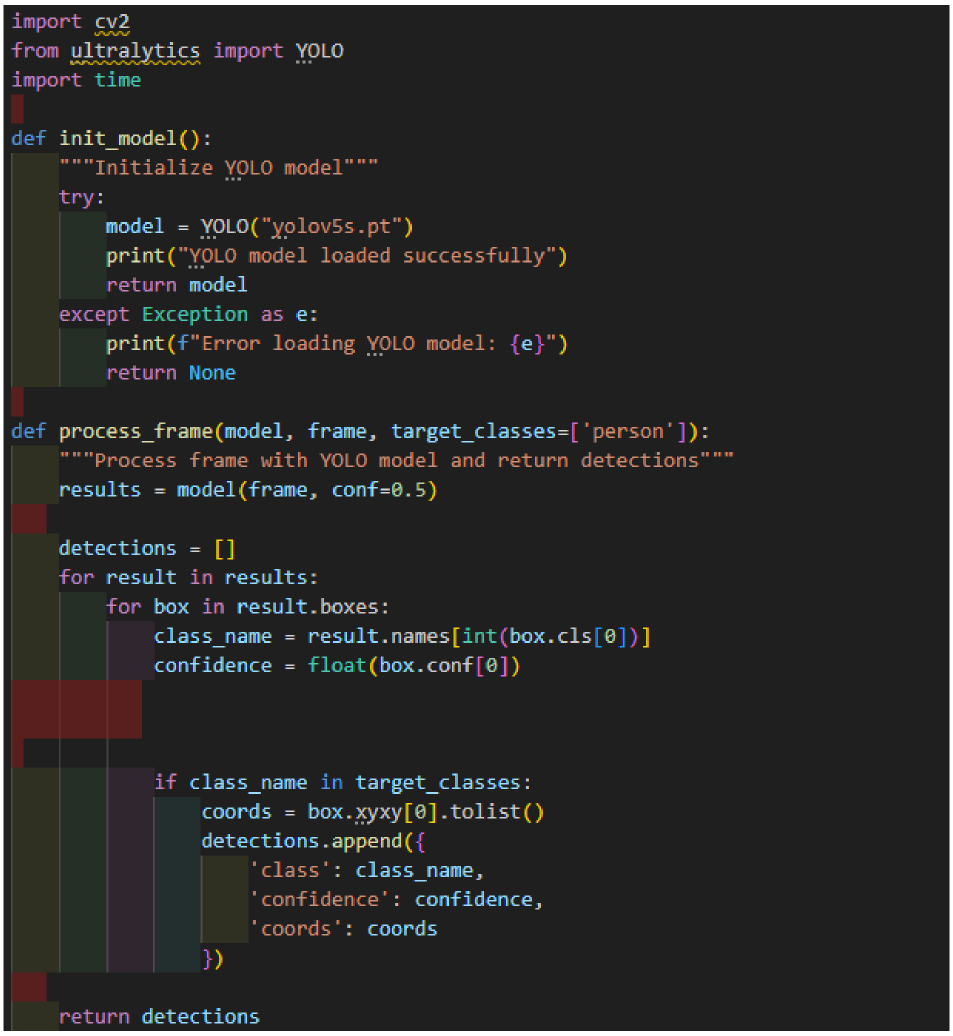
Figure 11: Code snippet showing logic that connects to API and checks status of start command

The above code snippet connects the robot arm to the API and by extension to the start and stop buttons of the user interface (the mobile app). It first connects to the API and sets its connection to true. After this, the check_status() function can be called which checks the robotic arms status variable of the API, it will await until it is set to “Running” by the user clicking the start button on the mobile app before starting the sorting process.

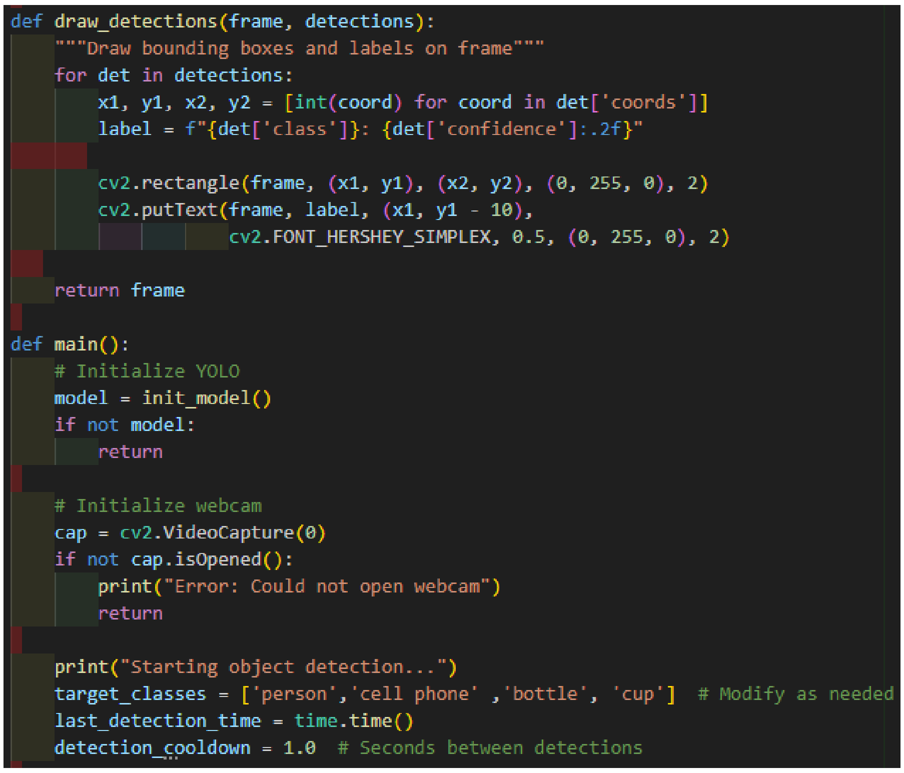
Section B – Object Detection and Computer Vision

Code Snippets

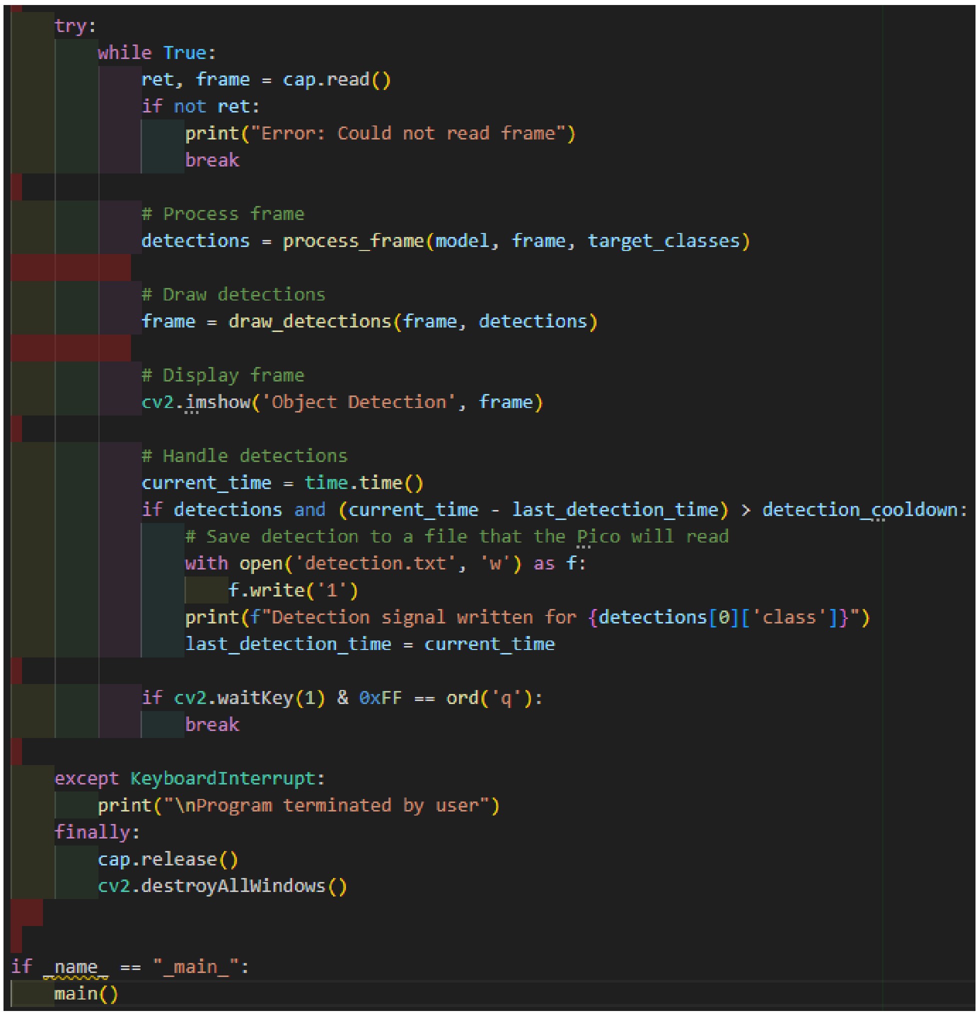
This code performs real-time object detection, identifies specified target objects and visually highlights them on the display, making it useful for applications where specific object tracking and logging are required.



```python
import cv2
from ultralytics import YOLO
import time

def init_model():
    """Initialize YOLO model"""
    try:
        model = YOLO("yolov5s.pt")
        print("YOLO model loaded successfully")
        return model
    except Exception as e:
        print(f"Error loading YOLO model: {e}")
        return None

def process_frame(model, frame, target_classes=['person']):
    """Process frame with YOLO model and return detections"""
    results = model(frame, conf=0.5)

    detections = []
    for result in results:
        for box in result.boxes:
            class_name = result.names[int(box.cls[0])]
            confidence = float(box.conf[0])


            if class_name in target_classes:
                coords = box.xyxy[0].tolist()
                detections.append({
                    'class': class_name,
                    'confidence': confidence,
                    'coords': coords
                })

    return detections
```

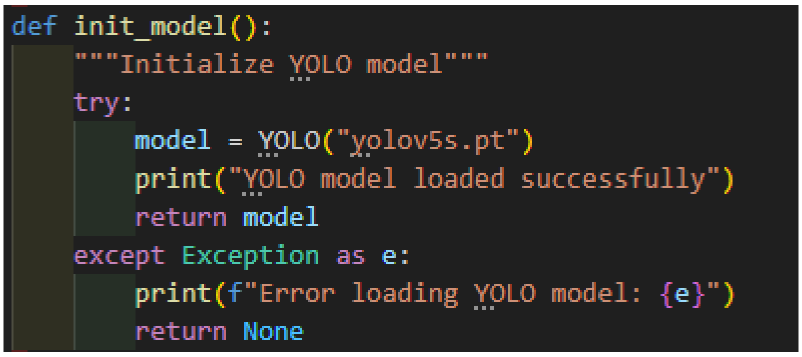


```python
def draw_detections(frame, detections):
    """Draw bounding boxes and labels on frame"""
    for det in detections:
        x1, y1, x2, y2 = [int(coord) for coord in det['coords']]
        label = f"{det['class']}: {det['confidence']:.2f}"

        cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
        cv2.putText(frame, label, (x1, y1 - 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

    return frame

def main():
    # Initialize YOLO
    model = init_model()
    if not model:
        return

    # Initialize webcam
    cap = cv2.VideoCapture(0)
    if not cap.isOpened():
        print("Error: Could not open webcam")
        return

    print("Starting object detection...")
    target_classes = ['person','cell phone' ,'bottle', 'cup']  # Modify as needed
    last_detection_time = time.time()
    detection_cooldown = 1.0  # Seconds between detections
```



```python
    try:
        while True:
            ret, frame = cap.read()
            if not ret:
                print("Error: Could not read frame")
                break

            # Process frame
            detections = process_frame(model, frame, target_classes)

            # Draw detections
            frame = draw_detections(frame, detections)

            # Display frame
            cv2.imshow('Object Detection', frame)

            # Handle detections
            current_time = time.time()
            if detections and (current_time - last_detection_time) > detection_cooldown:
                # Save detection to a file that the Pico will read
                with open('detection.txt', 'w') as f:
                    f.write('1')
                print(f"Detection signal written for {detections[0]['class']}")
                last_detection_time = current_time

            if cv2.waitKey(1) & 0xFF == ord('q'):
                break

    except KeyboardInterrupt:
        print("\nProgram terminated by user")
    finally:
        cap.release()
        cv2.destroyAllWindows()

if _name_ == "_main_":
    main()
```

init_model()

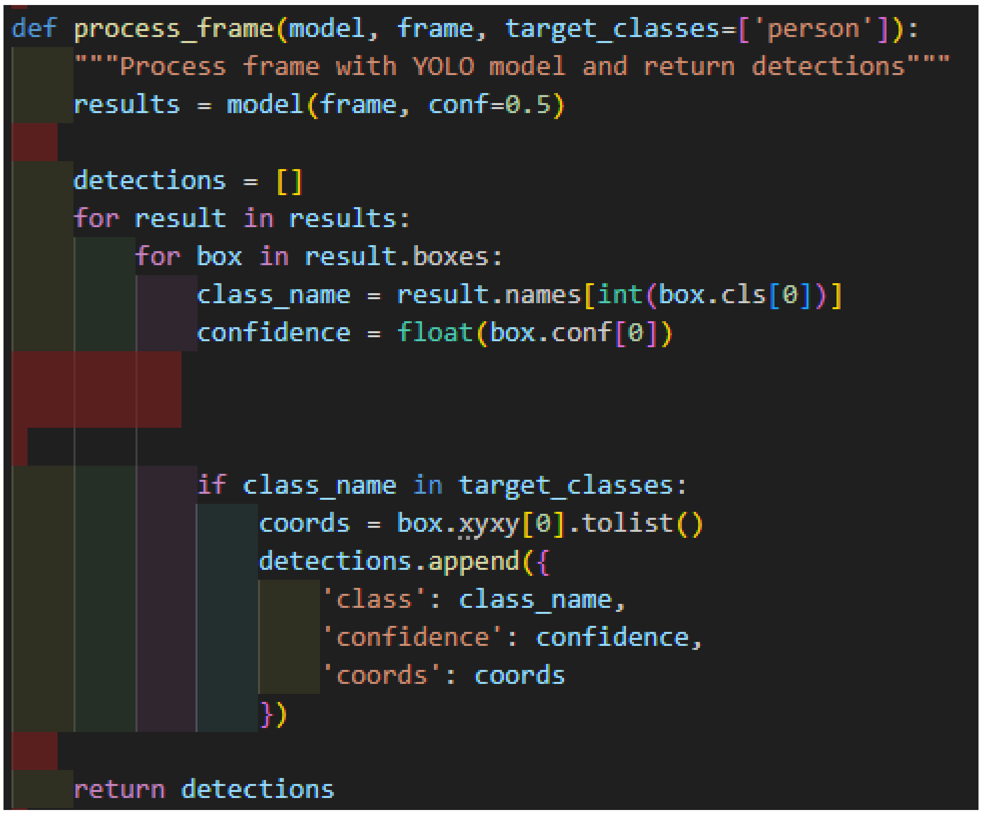


```python
def init_model():
    """Initialize YOLO model"""
    try:
        model = YOLO("yolov5s.pt")
        print("YOLO model loaded successfully")
        return model
    except Exception as e:
        print(f"Error loading YOLO model: {e}")
        return None
```

Purpose : Initializes the YOLO model for object detection.

Using a pre-trained model file (*yolov5s.pt*), this functions tries to load the YOLO model. The model object is returned if the loading of the model is successful. If not, it detects any exception and indicates that the model did not load by printing an error message and returning "*None*".

process_frame()



```python
def process_frame(model, frame, target_classes=['person']):
    """Process frame with YOLO model and return detections"""
    results = model(frame, conf=0.5)

    detections = []
    for result in results:
        for box in result.boxes:
            class_name = result.names[int(box.cls[0])]
            confidence = float(box.conf[0])


            if class_name in target_classes:
                coords = box.xyxy[0].tolist()
                detections.append({
                    'class': class_name,
                    'confidence': confidence,
                    'coords': coords
                })

    return detections
```

Purpose : Processes a single video frame with the YOLO Model, extracting detections for specified target classes.

The YOLO Model is applied to the frame, and detections with a confidence level of at least 0.5 are returned. For each detection(bounding box) in the results, it extracts :

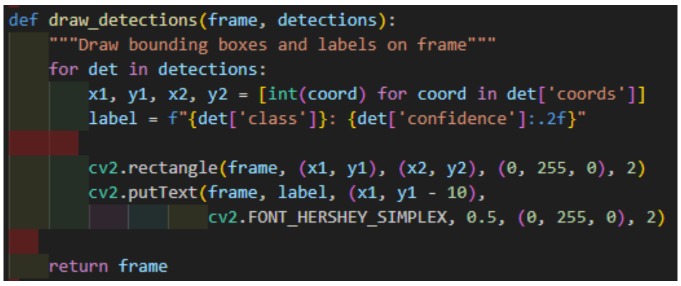
- ***class_name*** : The detected class("*person*")
- ***confidence*** : The detection confidence score

- ***coords*** : The bounding box co-ordinates

If the detected class is in *target_classes*, it appends a dictionary containing the class, confidence, and co-ordinates to *detection*.

Output : Returns a list of detected objects in the specified classes.

draw_detections()

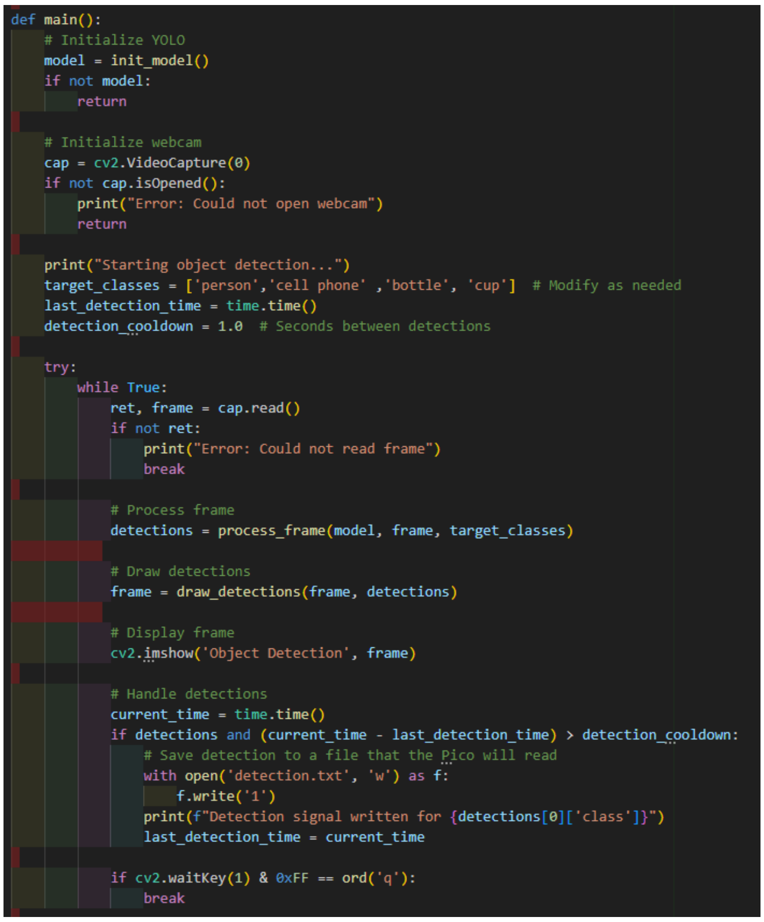


```python
def draw_detections(frame, detections):
    """Draw bounding boxes and labels on frame"""
    for det in detections:
        x1, y1, x2, y2 = [int(coord) for coord in det['coords']]
        label = f"{det['class']}: {det['confidence']:.2f}"

        cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
        cv2.putText(frame, label, (x1, y1 - 10),
                   cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

    return frame
```

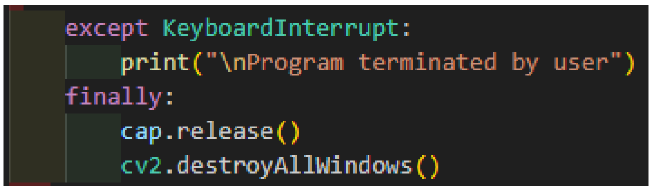
Purpose : Draws bounding boxes and labels on the video frame.

This program extracts the co-ordinates and confidence score for every object that is detected. The discovered object is surrounded by a green rectangle that bears the class name and confidence score. After that, the altered frame with labels and bounding boxes is sent back.

main()



```python
def main():
    # Initialize YOLO
    model = init_model()
    if not model:
        return

    # Initialize webcam
    cap = cv2.VideoCapture(0)
    if not cap.isOpened():
        print("Error: Could not open webcam")
        return

    print("Starting object detection...")
    target_classes = ['person','cell phone' ,'bottle', 'cup']  # Modify as needed
    last_detection_time = time.time()
    detection_cooldown = 1.0  # Seconds between detections

    try:
        while True:
            ret, frame = cap.read()
            if not ret:
                print("Error: Could not read frame")
                break

            # Process frame
            detections = process_frame(model, frame, target_classes)

            # Draw detections
            frame = draw_detections(frame, detections)

            # Display frame
            cv2.imshow('Object Detection', frame)

            # Handle detections
            current_time = time.time()
            if detections and (current_time - last_detection_time) > detection_cooldown:
                # Save detection to a file that the Pico will read
                with open('detection.txt', 'w') as f:
                    f.write('1')
                print(f"Detection signal written for {detections[0]['class']}")
                last_detection_time = current_time

            if cv2.waitKey(1) & 0xFF == ord('q'):
                break
```



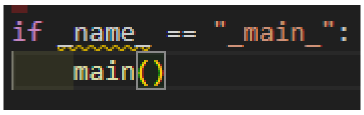
```
except KeyboardInterrupt:
    print("\nProgram terminated by user")
finally:
    cap.release()
    cv2.destroyAllWindows()
```

Purpose : The primary function to set up the YOLO Model, capture webcam input, process each frame, and display detections.

- **Initialize the YOLO Model** : Calls the *init_model()* to load the YOLO Model.
- **Initialize Webcam** : Opens the webcam for real-time video capture.
- **Process Loop** : A loop continuously reads frames from the webcam, detects objects, draws bounding boxes, and displays the updated frame.
- **Detection Cooldown** : Manages the timing between detections, using a *detection_cooldown* of 1 second to avoid overloading the output file.
- **Write Detection Signal** : If an object is detected and the cooldown period has passed, writes a signal (*1*) to a file (*detection.txt*) for external use, such as communication with a microcontroller like a Raspberry Pi Pico.

Exit Condition : The program ends if the webcam fails, the user presses '*q*', or if there is an interruption (like a keyboard interrupt).

name



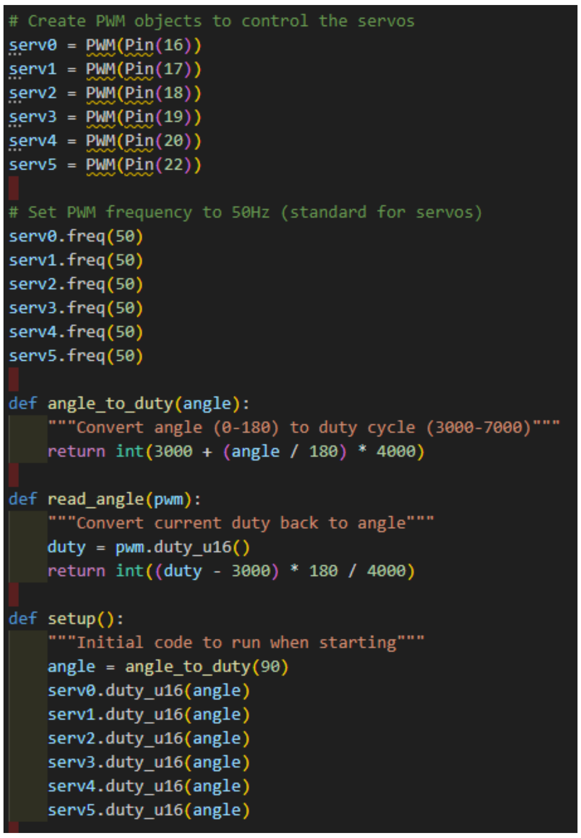
```
if _name_ == "_main_":
    main()
```

Purpose : Checks if the scripts is being run directly and, if so, calls *main()*.

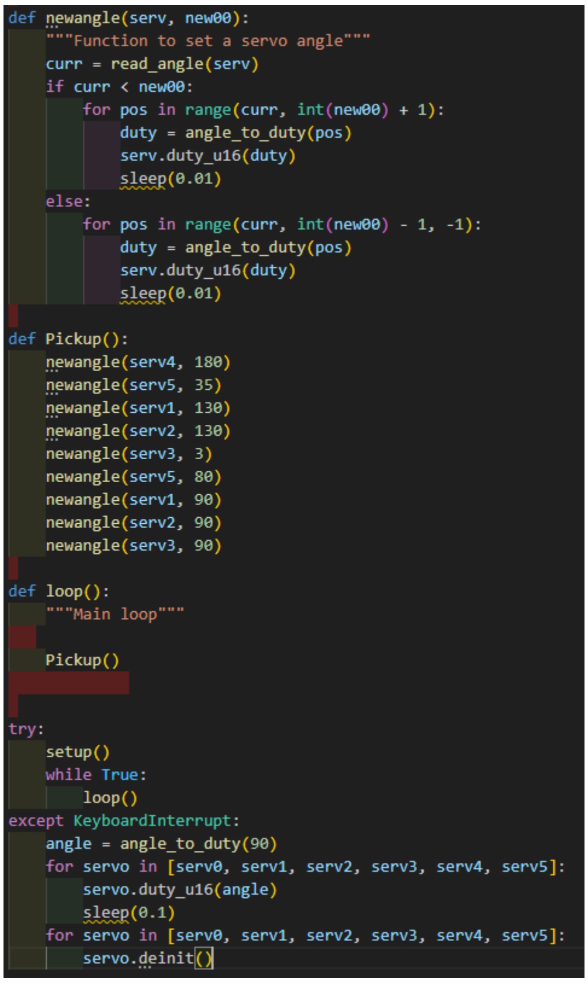
This condition ensures that *main()* is only called if the script executed directly, preventing it from running if the script is imported as a module in another program.

Section C - Robotic Arm Snippets and Brief Explanation

With the help of a microcontroller's Pulse Width Modulation(PWM) signal, this code manages several servos and enables accurate angle adjustments. Applications like a Robotic Arm carrying out repetitive operations that call for accurate, synchronized motions of several servos are the focus of this code.

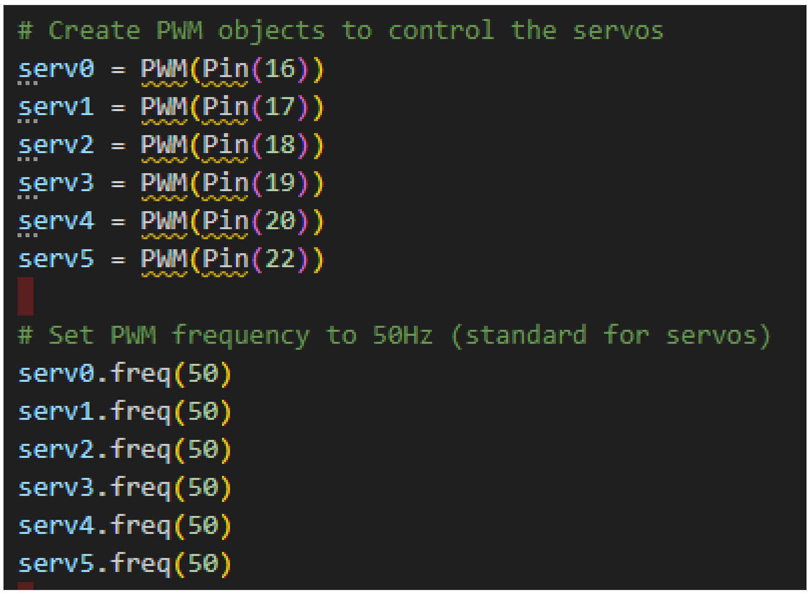


```
# Create PWM objects to control the servos
serv0 = PWM(Pin(16))
serv1 = PWM(Pin(17))
serv2 = PWM(Pin(18))
serv3 = PWM(Pin(19))
serv4 = PWM(Pin(20))
serv5 = PWM(Pin(22))

# Set PWM frequency to 50Hz (standard for servos)
serv0.freq(50)
serv1.freq(50)
serv2.freq(50)
serv3.freq(50)
serv4.freq(50)
serv5.freq(50)

def angle_to_duty(angle):
    """Convert angle (0-180) to duty cycle (3000-7000)"""
    return int(3000 + (angle / 180) * 4000)

def read_angle(pwm):
    """Convert current duty back to angle"""
    duty = pwm.duty_u16()
    return int((duty - 3000) * 180 / 4000)

def setup():
    """Initial code to run when starting"""
    angle = angle_to_duty(90)
    serv0.duty_u16(angle)
    serv1.duty_u16(angle)
    serv2.duty_u16(angle)
    serv3.duty_u16(angle)
    serv4.duty_u16(angle)
    serv5.duty_u16(angle)
```



```python
def newangle(serv, new00):
    """Function to set a servo angle"""
    curr = read_angle(serv)
    if curr < new00:
        for pos in range(curr, int(new00) + 1):
            duty = angle_to_duty(pos)
            serv.duty_u16(duty)
            sleep(0.01)
    else:
        for pos in range(curr, int(new00) - 1, -1):
            duty = angle_to_duty(pos)
            serv.duty_u16(duty)
            sleep(0.01)

def Pickup():
    newangle(serv4, 180)
    newangle(serv5, 35)
    newangle(serv1, 130)
    newangle(serv2, 130)
    newangle(serv3, 3)
    newangle(serv5, 80)
    newangle(serv1, 90)
    newangle(serv2, 90)
    newangle(serv3, 90)

def loop():
    """Main loop"""

    Pickup()


try:
    setup()
    while True:
        loop()
except KeyboardInterrupt:
    angle = angle_to_duty(90)
    for servo in [serv0, serv1, serv2, serv3, serv4, serv5]:
        servo.duty_u16(angle)
        sleep(0.1)
    for servo in [serv0, serv1, serv2, serv3, serv4, serv5]:
        servo.deinit()
```

Servo Initialization and Frequency Setup

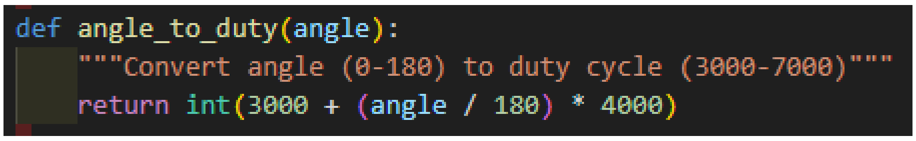


```python
# Create PWM objects to control the servos
serv0 = PWM(Pin(16))
serv1 = PWM(Pin(17))
serv2 = PWM(Pin(18))
serv3 = PWM(Pin(19))
serv4 = PWM(Pin(20))
serv5 = PWM(Pin(22))

# Set PWM frequency to 50Hz (standard for servos)
serv0.freq(50)
serv1.freq(50)
serv2.freq(50)
serv3.freq(50)
serv4.freq(50)
serv5.freq(50)
```

Purpose : Initializes PWN objects on specific pins to control 6 servos, each corresponding to a joint or part of a robotic arm.

Each servo is assigned to a PWM pin. The frequency is set to 50Hz, which is the standard for most servos, enabling the control of servo positions based on the duty cycle.

angle_to_duty()

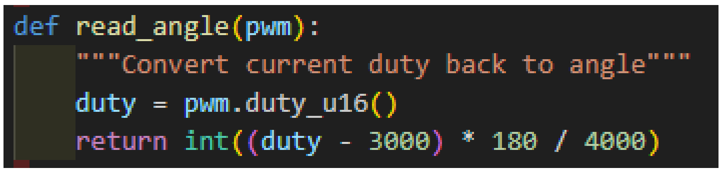


```python
def angle_to_duty(angle):
    """Convert angle (0-180) to duty cycle (3000-7000)"""
    return int(3000 + (angle / 180) * 4000)
```

Purpose : Converts an angle (in degrees) to the PWM duty cycle needed to achieve it.

The usual range of operation for servo control is 0° - 180°. With 3000 denoting 0° and 7000 180°, the duty cycle value is transferred from 3000 – 7000. The servos can move to the required location because this function determines the proper duty cycle for a given angle.

read_angle()

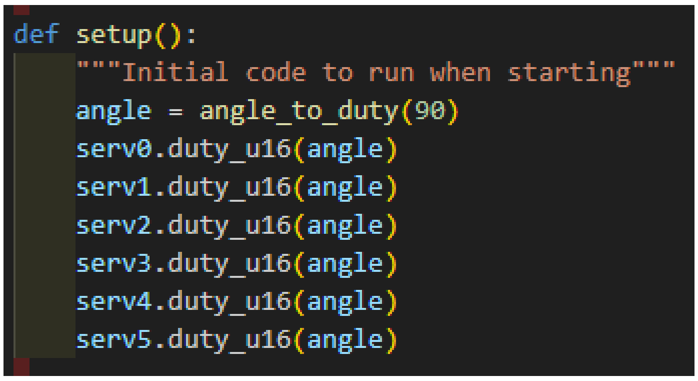


```python
def read_angle(pwm):
    """Convert current duty back to angle"""
    duty = pwm.duty_u16()
    return int((duty - 3000) * 180 / 4000)
```

Purpose : Reads the current angle of a servo based on its duty cycle.

This function returns an angle after retrieving the specified servo's current duty cycle. This facilitates real-time tracking of the servo's position.

setup()

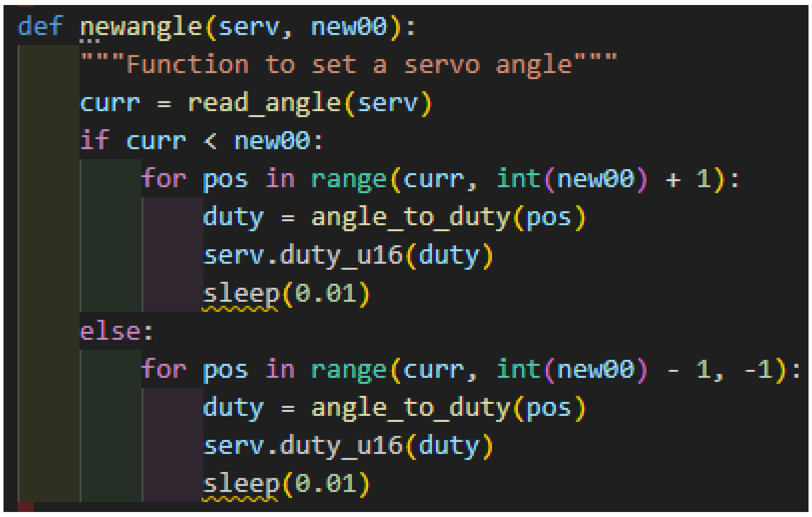


```python
def setup():
    """Initial code to run when starting"""
    angle = angle_to_duty(90)
    serv0.duty_u16(angle)
    serv1.duty_u16(angle)
    serv2.duty_u16(angle)
    serv3.duty_u16(angle)
    serv4.duty_u16(angle)
    serv5.duty_u16(angle)
```

Purpose : Sets each servo to a neutral position (90°).

The *setup* function initializes each servo to 90° angle, which typically represents a midpoint for many servo applications. This ensures all servos start in a known, central position.

newangle()

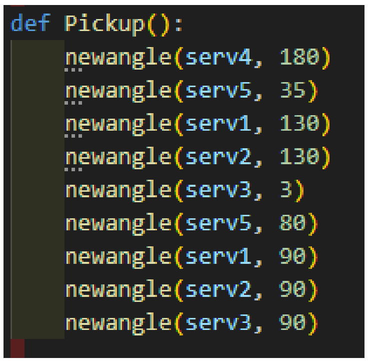


```python
def newangle(serv, new00):
    """Function to set a servo angle"""
    curr = read_angle(serv)
    if curr < new00:
        for pos in range(curr, int(new00) + 1):
            duty = angle_to_duty(pos)
            serv.duty_u16(duty)
            sleep(0.01)
    else:
        for pos in range(curr, int(new00) - 1, -1):
            duty = angle_to_duty(pos)
            serv.duty_u16(duty)
            sleep(0.01)
```

Purpose : Moves a servo to a new angle gradually, creating a smooth transition.

This function iteratively advances towards the intended *new00* angle after reading the servo's current angle. To provide a smooth motion, it moves in little steps, halting momentarily in between. By changing the direction of the iteration, the function permits movement in both clockwise and counterclockwise directions.

Pickup()

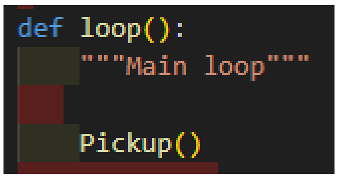


```python
def Pickup():
    newangle(serv4, 180)
    newangle(serv5, 35)
    newangle(serv1, 130)
    newangle(serv2, 130)
    newangle(serv3, 3)
    newangle(serv5, 80)
    newangle(serv1, 90)
    newangle(serv2, 90)
    newangle(serv3, 90)
```

Purpose : Controls the servos to perform a sequence of movements for picking up an object.

In order to carry out a sequence of motions (perhaps a pickup sequence for a robotic arm), this function determines precise angles for each servo. It creates a synchronized movement for grabbing an object by moving different joints to certain angles in a predetermined order.

loop()

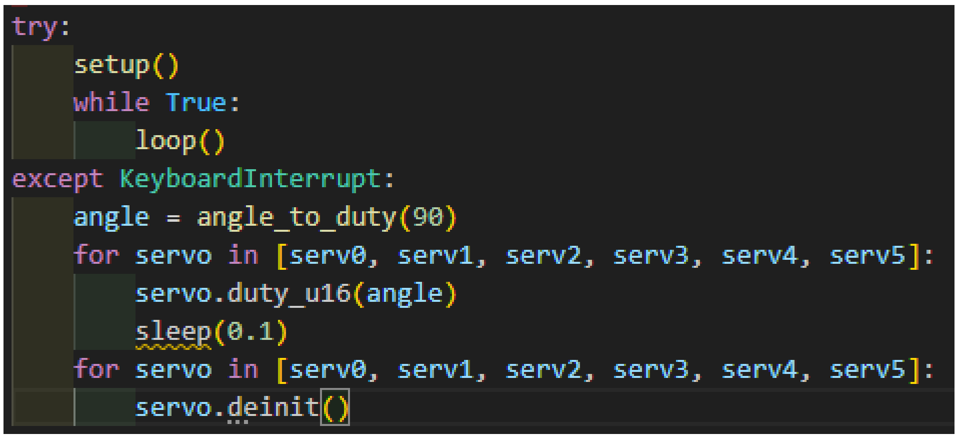


```python
def loop():
    """Main loop"""

    Pickup()
```

Purpose : Main loop that repeatedly calls the *Pickup* function.

This function is designed to continuously run the pickup sequence as part of the main program's repeated actions.

Main Program Execution



```python
try:
    setup()
    while True:
        loop()
except KeyboardInterrupt:
    angle = angle_to_duty(90)
    for servo in [serv0, serv1, serv2, serv3, serv4, serv5]:
        servo.duty_u16(angle)
        sleep(0.1)
    for servo in [serv0, serv1, serv2, serv3, serv4, serv5]:
        servo.deinit()
```

Purpose : Initializes and runs the servo control loop, with error handling for a safe shutdown.

After initializing the servo position using *setup()*, the program enters an endless loop and keeps running *loop()*. It de-initializes each servo to release the PWM resources and return each servo to 90° if it is interrupted (for example, pressing Ctrl + C).