Python Library Documentation

For the duration of Project 3, your team will use the functions listed below to design a computer program for sorting and recycling containers of varying materials. This will be achieved through interfacing with the Quanser Interactive Labs (Q-Labs) environment.

The following is a breakdown of the categories those functions fall under.

Q-arm: Movement

By default, the Q-arm starts in the home position where all joint angles (base, shoulder, elbow, wrist) are set to zero. Otherwise, any of the following functions can be used to move the Q-arm as desired.

rotate_base(deg)

It rotates the base joint by the number of degrees specified when the function is called. Function calls are relative to each other, meaning when rotating the base by 35 degrees at first then later by 40 degrees, the final position will be at 75 degrees and not 40 degrees with movement limitation of +/- 175 degrees.

Example: The first function call rotates the base joint 56 degrees counter-clockwise and second function call rotates the joint 35 degrees in the clockwise direction.

```
>>> arm.rotate_base(56)
>>> arm.rotate_base(-35)
```

rotate_shoulder(deg)

It rotates the shoulder joint by the number of degrees specified when the function is called. Function calls are relative to each other, meaning when rotating the shoulder by 20 degrees at first then later by -15 degrees, the final position will be at 5 degrees and not -15 degrees with movement limitation of +/- 90 degrees.

Example: The first line rotates the shoulder joint 45 degrees upwards while the second rotates it 67 degrees downwards.

```
>>> arm.rotate_shoulder(-45)
>>> arm.rotate_shoulder(67)
```

rotate_elbow(deg)

It rotates the elbow joint by the number of degrees specified when the function is called. Function calls are relative to each other, meaning when rotating the elbow by 10 degrees at first then later by -20 degrees, the final position will be at -10 degrees and not -20 degrees with movement limitation of +90 degrees in the downward direction and -80 degrees in the upward direction.

Example: The first line rotates the elbow joint 10 degrees upward while the second rotates it 5 degrees downward.

```
>>> arm.rotate_elbow(-10)
>>> arm.rotate_elbow(5)
```

rotate_wrist(deg)

It rotates the wrist joint by the number of degrees specified when the function is called. Function calls are relative to each other, meaning when rotating the wrist by 5 degrees at first then later by 15 degrees, the final position will be at 20 degrees and not 15 degrees with movement limitation of +/-170 degrees.

Example: The function calls rotate the wrist 45 degrees counter-clockwise then an additional 30 degrees in the same direction.

```
>>> arm.rotate_wrist(45)
>>> arm.rotate_wrist(30)
```

move_arm(x, y, z)

It moves the Q-arm to target location based on a cartesian coordinate input, taking in three input arguments corresponding to x, y, and z coordinates in 3D space.

Example: The Q-arm's joints are rotated to move the arm to the specified xyz location.

```
>>> arm.move_arm(-0.6097, 0.2463, 0.3643)
```

home()

It moves the Q-arm to default position in the environment, corresponding to all joints being at 0 degrees and the gripper being fully open. When used, this function takes no arguments.

Example: Calling the home function takes the arm back to the default position.

```
>>> arm.home()
```

Q-arm: Gripper

Objects are picked up or dropped off by opening or closing the Q-arm's gripper.

control_gripper(deg)

It controls the opening and closing of the gripper with degrees. A value of zero corresponding to fully open and 45 degrees correspond to fully closed. Function calls are relative, meaning when passing an angle greater than zero, the gripper will partially close but to fully open it again, you must pass the same number but with a negative sign to have a sum of zero.

Example: The gripper is first fully closed and then fully opened.

```
>>> arm.control_gripper(45)
>>> arm.control_gripper(-45)
```

Q-arm: Coordinates

It returns the Q-arm coordinates in 3D space.

effector_position()

It returns the xyz coordinates of the Q-arm's location in 3D space as a 3-item list.

Example: The function returns the cartesian location of the Q-arm.

```
>>> arm.effector_position()
(0.4064, 0.0, 0.4826)
```

Q-bot: Movement

The following functions can be used to move the Q-bot around the QuanserSim Environment as desired.

forward_time(time)

When this function is called, the Q-bot will move forward for the set amount of time in seconds at the speed that has been set when initializing the Q-bot.

Example: This moves the Q-bot forward for 4 seconds.

```
>>> bot.forward_time(4)
```

forward_distance(distance)

When this function is called, the Q-bot will move forward for the set distance in meters from its current position, regardless of what speed was set for the Q-bot.

Example: This moves the Q-bot 2 meters forward

```
>>> bot.forward_distance(2)
```

travel_forward(threshold)

The Q-bot will continue to drive forward until it judges the distance between it and the object in front of it is less than the defined threshold. The camera angle of the Q-bot may need to be changed to improve the results of this function.

Example: The Q-bot moves forward until it reaches a depth of 0.25 m away from an obstacle.

```
>>> bot.travel_forward(0.25)
Depth (m):  0.25121568627450974
```

set_wheel_speeds(speeds)

When this function is called, the Q-bot will move forward at the given velocity until the function stop() is called. Valid input is in the form of a 2-item list where the first item is the speed of the left wheel and the second item is the speed of the right wheel of the Q-bot.

Example: The first line sets both wheels speed at 0.1m/s for a constant forward movement. The second line moves the Q-bot in ever widening circles as the left wheel moves at 0.1 m/s and the right wheel at 0.2 m/s

```
>>> bot.set_wheel_speed([0.1,0.1])
>>> bot.set_wheel_speed([0.1,0.2])
```

rotate(deg)

The Q-bot will rotate on the spot by the given number of degrees. A positive value will result in a clockwise rotation and a negative value will result in a counter-clockwise rotation. Function calls are relative to each other meaning that a rotation of 50 degrees and then a rotation of -30 degrees will result in a final rotation of 20 degrees clockwise.

Example: The Q-bot rotates clockwise 50 degrees and then counter-clockwise 20 degrees.

```
>>> bot.rotate(50)
>>> bot.rotate(-20)
```

stop()

The Q-bot stops moving.

```
>>> bot.stop()
```

Q-bot: Position

It returns the Q-bot position in the QuanserSim Environment.

position()

It returns the xyz coordinates of the Q-bot's location in 3D space at the time the function was called. Towards the bins is y and towards the arm is x.

Example: The function returns the cartesian location of the Q-bot beside Bin 4 as a 3-item list.

```
>>> bot.position()
(-1.01, 1.12, -0.01)
```

depth()

Reads and returns how far the Q-bot is from an object e.g., walls.

Example: The Q-bot measures the depth from it to the nearest wall that it is facing.

```
>>> bot.depth()
0.25121568627450974
```

Q-bot: Line Following

The following function can be used to follow the yellow line in the QuanserSim Environment.

line_following_sensors()

Uses two IR Proximity sensors mounted on the front of the Q-bot to sense the presence of the yellow line. Calling this function will return a two-item list consisting of the left IR sensor reading and right IR sensor reading. Each reading will either be a high value of 1, indicating the presence of the yellow line, or a low value of 0 indicating the absence of the yellow line. For this function to function correctly, the camera angle of the Q-bot **must** be set to -21.5 degrees.

This function in combination with *set_wheel_speeds* can be used to program a simple line following algorithm which you must program.

Example 1: Calling the function while the Q-bot is on the yellow line.

```
>>> bot.line_following_sensors()
[1, 1]
```

Example 2: Calling the function while the Q-bot is partially on the yellow line. Only the left IR sensor reads a high value.

```
>>> bot.line_following_sensors()
[1, 0]
```

Q-bot: Dumping Mechanism

It dumps the containers that are held in the box on top of the Q-bot.

dump()

Dumps the containers along a generic pre-defined motion if the actuator has been activated. Either Actuator must be activated in order to run this function (see section below).

Example: Q-bot tilts the box to dump the contents.

```
>>> bot.dump()
```

Q-bot: Actuator

The following functions can be used to control the actuator that drives the device used to deposit containers into the recycling bin.

activate_linear_actuator()

Calling this function activates the linear actuator and resets the box's position and rotation.

```
>>> bot.activate_linear_actuator()
Actuator activated.
```

deactivate_linear_actuator()

Calling this function deactivates the linear actuator and resets the box's position and rotation.

```
>>> bot.deactivate_linear_actuator()
Actuator deactivated.
```

activate_stepper_motor()

Calling this function activates the rotary actuator and resets the box's position and rotation.

```
>>> bot.activate_stepper_motor()
Actuator activated.
```

deactivate_stepper_motor()

Calling this function deactivates the rotary actuator and resets the box's position and rotation.

```
>>> bot.deactivate_stepper_motor()
Actuator deactivated.
```

Q-Labs ONLY

rotate_hopper(angle)

Calling this function rotates the recycling hopper about its axis to a specified angle. Either Actuator must be activated in order to run this function (but not both). The specified angle must be positive in the range $0 \leq angle \leq 90$.

Example: The first line rotates the hopper 30 degrees about its axis. The second line rotates the hopper about its axis completely at 90 degrees. The third line sets the hopper back at the resting position at 0 degrees.

```
>>> bot.rotate_hopper(30)
>>> bot.rotate_hopper(90)
>>> bot.rotate_hopper(0)
```

Hardware ONLY

linear_actuator_out(time_extend)

Calling this function extends the physical linear actuator for a specified amount of time. Valid input is time in seconds, with the actuator extending at a rate of 10mm/s, for a maximum extension of 50mm. Attempting to actuate past this max extension will give a warning, and no actuation will occur. For this function to work, the linear actuator must be activated and hardware should be set to **True** in the project_sample.py setup.

Example: Attempting to extend the actuator for 5 seconds in the simulation environment

```
>>> bot.linear_actuator_out(5)
Please use simulation acctuator methods.
```

linear_actuator_in(time_extend)

Calling this function retracts the physical linear actuator for a specified amount of time. Valid input is time in seconds, with the actuator retracting at a rate of 10mm/s, for a maximum retraction limit of 0mm. Attempting to actuate past this max retraction will give a warning, and no actuation will occur. For this function to work, the linear actuator must be activated, and hardware should be set to **True** in the project_sample.py setup.

Example: Calling the function to retract the actuator for 5 seconds

```
>>> bot.linear_actuator_in(5)
```

rotate_stepper_cw(time_rotate)

Calling this function rotates the rotary actuator (stepper motor) clockwise at a rate of 30RPM, or 180 deg/s for a specified amount of time in seconds. Valid input is time in seconds. The rotary actuator must be activated for this function to work, and hardware should be set to **True** in the project_sample.py setup.

Example: Attempting to rotate the rotary actuator clockwise for 5 seconds in the simulation environment

```
>>> bot.rotate_stepper_cw(5)
Please use simulation acctuator methods.
```

rotate_stepper_ccw(time_rotate)

Calling this function rotates the rotary actuator (stepper motor) counter-clockwise at a rate of 30RPM, or 180 deg/s for a specified amount of time in seconds. Valid input is time in seconds. The rotary actuator must be activated for this function to work, and hardware should be set to **True** in the project_sample.py setup.

Example: Calling the function to rotate the rotary actuator counter-clockwise for 5 seconds

```
>>> bot.rotate_stepper_ccw(5)
```

Q-bot: Sensors

To help with distinguishing between the different bins in the Recycling station, you have been provided with several sensors that you can activate in addition to the sensors already provided on the Q-bot. The available sensors include Ultrasonic, Active Infrared (IR), Light Dependent Resistor (LDR), and Color.

Ultrasonic Sensor

activate_ultrasonic_sensor()

Activates the Ultrasonic sensor on the Q-bot.

```
>>> bot.activate_ultrasonic_sensor()
Ultrasonic sensor activated
```

deactivate_ultrasonic_sensor()

Deactivates the Ultrasonic sensor on the Q-bot.

```
>>> bot.deactivate_ultrasonic_sensor()
Ultrasonic sensor deactivated
```

read_ultrasonic_sensor()

Calling this function (once the sensor is activated) outputs the distance in meters from the Q-bot's bumper to the front face of the nearest bin. If the distance is not in range that the Ultrasonic sensor can operate, the function will return 0.

Range: $0 \leq distance\ (m) \leq 2.5$

Example: The distance in meters from the Q-bot's bumper to the front face of the nearest bin collected by the Ultrasonic sensor.

```
>>> bot.read_ultrasonic_sensor()
0.49
```

Active Infrared (IR) Sensor

activate_ir_sensor()

Activates the Active Infrared (IR) sensor on the Q-bot.

```
>>> bot.activate_ir_sensor()
Active IR sensor activated
```

deactivate_ir_sensor()

Deactivates the Active Infrared (IR) sensor on the Q-bot.

```
>>> bot.deactivate_ir_sensor()
Active IR sensor deactivated
```

read_ir_sensor()

Calling this function (once the Active Infrared sensor is activated) outputs high voltage readings a bin is within proximity to the Q-bot and the bin is within the sensor's range. If the distance is not in range that the Active IR sensor can operate, the function will return low voltage readings.

Range: $0 \leq distance\ (m) \leq 0.25$

Example: The voltage readings collected by the Active IR sensor within range of a bin.

```
>>> bot.read_ir_sensor()
4.899888719129141
```

Light Dependent Resistor (LDR) Sensor

activate_ldr_sensor()

Activates the Light Dependent Resistor (LDR) sensor on the Q-bot.

```
>>> bot.activate_ldr_sensor()
LDR sensor activated
```

deactivate_ldr_sensor()

Deactivates the Light Dependent Resistor (LDR) sensor on the Q-bot.

```
>>> bot.deactivate_ldr_sensor()
LDR sensor deactivated
```

read_ldr_sensor()

Calling this function (once the Light Dependent Resistor (LDR) sensor is activated) outputs high voltage if a light is sensed around the Q-bot.

Range: $0 \leq distance\ (m) \leq 0.25$

Example 1: The high voltage reading collected by the LDR sensor within the range of a bin

```
>>> bot.read_ldr_sensor()
0.94
```

Color Sensor

activate_color_sensor()

Activates a Color sensor on the Q-bot.

```
>>> bot.activate_color_sensor()
Color sensor activated
```

deactivate_color_sensor()

Deactivates the Color sensor on the Q-bot.

```
>>> bot.deactivate_color_sensor()
Color sensor deactivated
```

read_color_sensor()

Calling this function outputs converted RGB values and raw sensor values of the nearest bin under the condition the bin is within the sensors range. The output is a list of three item lists, where the first list is the converted RGB values of the sensed color, and the second list is raw values read from the sensor.

Range: $0 \leq distance\ (m) \leq 0.25$

Example 1: The output collected by the Color sensor near Bin 1 which has been changed to the color red.

```
>>> bot.read_color_sensor()
([1, 0, 0], [605.7, 123.6, 159.2])
```

Example: The output collected by the Color sensor near Bin 2 which has been changed to the color green.

```
>>> bot.read_color_sensor()
([0, 1, 0], [211.5, 635.0, 159.7])
```

Container Properties

Each container has colour, material, and classification that is used to determine the bin to which it needs to be delivered. Container colour can be either clear, red, or blue and material can be either plastic, metal, or paper. Lastly, the classification of the container will be either contaminated or non-contaminated.

dispense_container(value, properties)

When this function is called, a container is spawned onto the table depending on the *value* variable inputted. A valid value is a number between 1 and 6 (inclusive) and each value coincides to a container with distinct properties. 1, 2 & 3 corresponds to plastic, metal, and paper containers, respectively, that are non-contaminated, and 4, 5 & 6 corresponds to plastic, metal, and paper containers, respectively, that are contaminated.

Setting *properties* to the Boolean True will also return the container material, mass (in grams) of the container, as well as the bin ID (where the container needs to be dropped off). By default, the function returns no properties.

Example 1: Dispenses a non-contaminated plastic container onto the table.

```
>>> table.dispense_container(1)
```

Example 2: Dispenses a contaminated paper container onto the table and outputs the material, mass, and bin ID for container 6 as a 3-item list.

```
>>> table.dispense_container(6,True)
('paper', 59.32860054161556, 'Bin04')
```