Getting Started with LEGO® MINDSTORMS® Education EV3 MicroPython

Version 1.0.0

TABLE OF CONTENTS

This guide shows you how to get started with writing MicroPython programs for your LEGO® MINDSTORMS® EV3 robots. You'll learn to do this in two steps:

- Installation: First you'll prepare your computer and your EV3 Brick by collecting and installing the required tools. You'll also learn how to switch the EV3 Brick on and off, and how to navigate the on-screen menu.
- Creating and running programs: Next, you'll learn how to create a program and download it to the EV3 Brick. You'll also learn how to start that program from your computer or from the EV3 Brick.

After you've run the first demo program, you'll be ready to try out the example programs and start inventing your own programs.

CHAPTER

ONE

INSTALLATION

This page guides you through the steps for collecting and installing everything you need to start programming.

1.1 What is needed?

To get started, you'll need:

- A Windows 10 or Mac OS computer
- Internet access and administrator access

  This is only required during the installation process. You will not need special access in order to write and run programs later on.
- A micro SD card

  You'll need a card with a minimum capacity of 4GB and a maximum capacity of 32GB. This type of micro SD card is also known as 'micro SDHC'. We recommend using cards with the Application Performance Class A1 rating.
- A micro SD card slot in your computer, or a card reader.

  If your computer does not have a (micro) SD card slot, you can use an external USB (micro) SD card reader.
- A mini USB cable, like the one included in your EV3 set.

The typical configuration of this equipment is summarised in Figure 1.1.

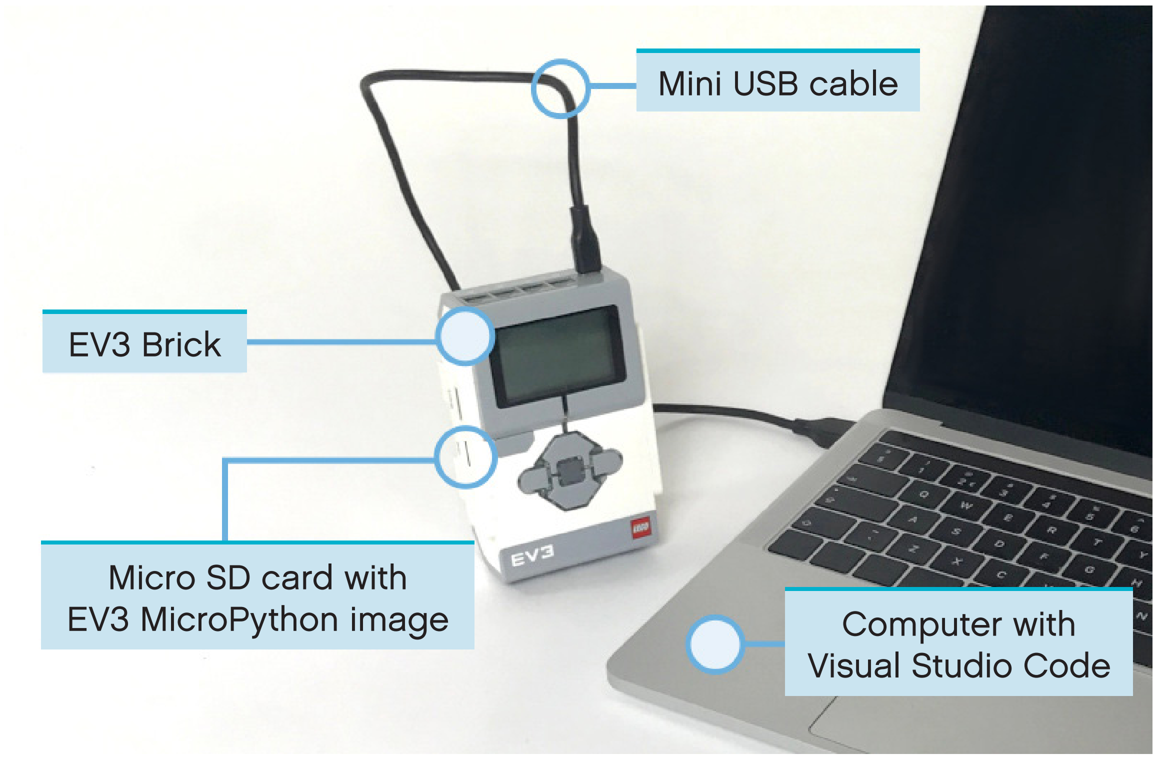

Mini USB cable
EV3 Brick
Micro SD card with EV3 MicroPython image
Computer with Visual Studio Code

Figure 1.1: Setup overview.

1.2 Preparing your computer

You'll write your MicroPython programs using Visual Studio Code. Follow the steps below to download, install and configure this application.

1. Download Visual Studio Code.
2. Follow the on-screen instructions to complete installation.
3. Launch Visual Studio Code.
4. Open the 'extensions' tab.
5. Install the EV3 MicroPython extension as shown in Figure 1.2.

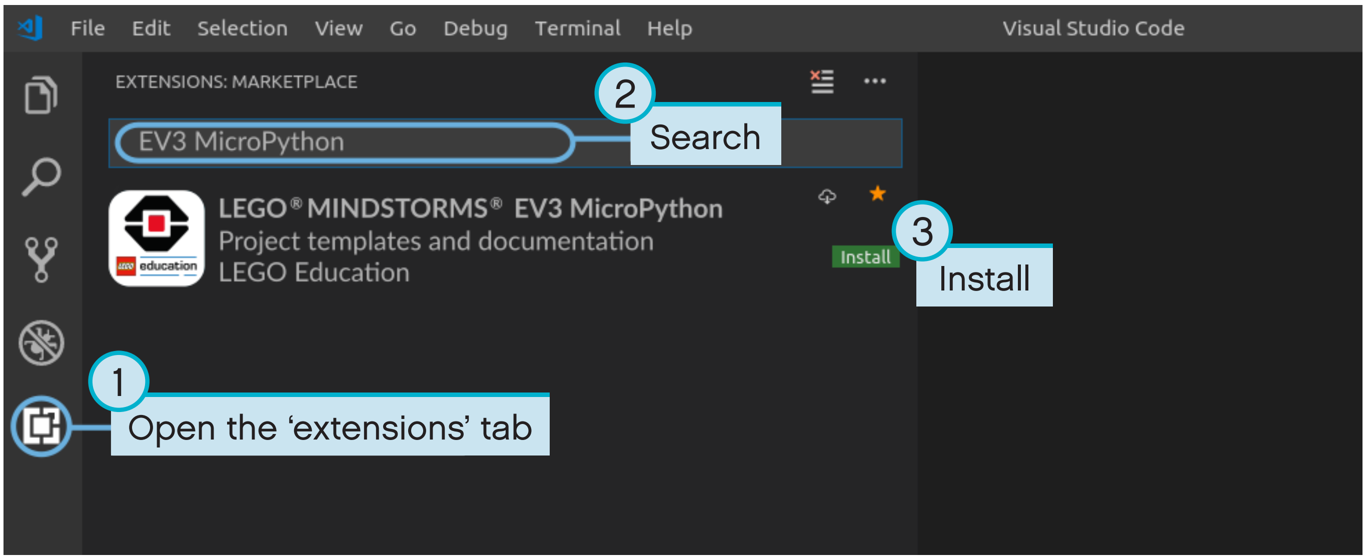


Figure 1.2: Installing the extension from the Visual Studio Code Marketplace.

1.3 Preparing the micro SD card

Now you'll learn how to install the required tools onto your micro SD card. This will make it possible to run MicroPython programs on your EV3 Brick.

If the micro SD card contains files you wish to keep, make sure you create a backup of its contents before installing the tools. If necessary, please refer to 'Managing files on the EV3 Brick' to learn how to back up your previous MicroPython programs.

This process erases everything on your micro SD card, including any previous MicroPython programs that are stored on it.

To install the MicroPython tools on your micro SD card:

1. Download the EV3 MicroPython micro SD card image and save it to a convenient location. This file is approximately 360 MB. You do not need to unzip the file.
2. Download and install a micro SD card flashing tool such as Etcher.
3. Insert the micro SD card into your computer or card reader.
4. Launch the flashing tool and follow the steps shown on your screen to install the file you have just downloaded. If you use Etcher, you can follow the instructions below, as shown in Figure 1.3.
   a. Select the EV3 MicroPython micro SD card image file you have just downloaded.
   b. Select your micro SD card. Make sure that the device and size shown correspond to your micro SD card.
   c. Start the flashing process. This may take several minutes. Do not remove the card until the flashing process is complete.

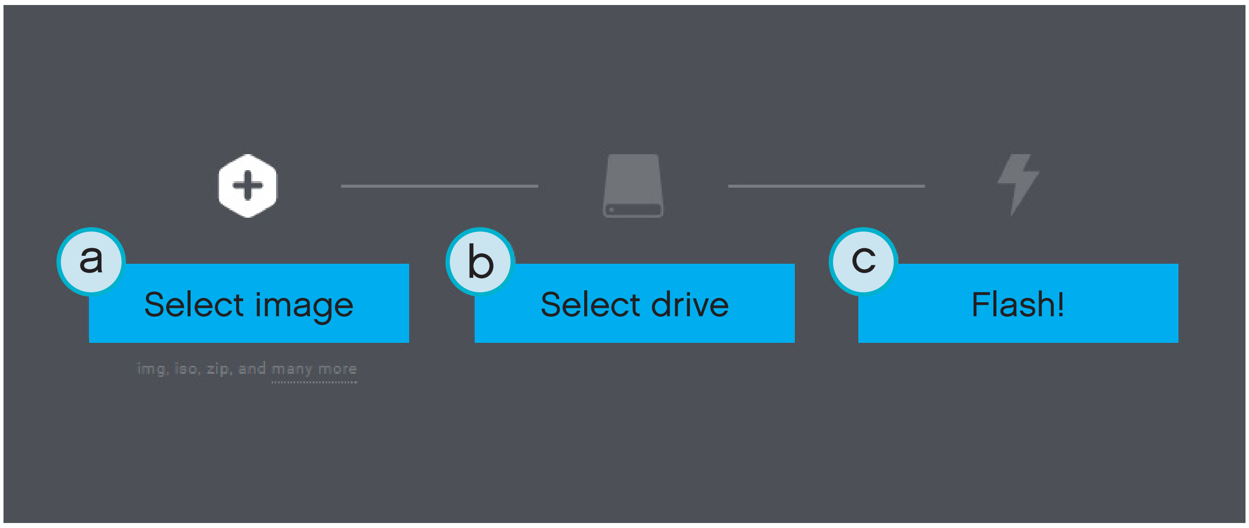


Figure 1.3: Using Etcher to flash the EV3 MicroPython micro SD card image.

1.4 Updating the micro SD card

To update the micro SD card, use the link above to download a new image file and flash it to the micro SD card as described above. Be sure to back up any MicroPython programs that you wish to save.

You do not need to erase the contents of the micro SD card first. This is done automatically when you flash the new image file.

1.5 Using the EV3 Brick

Make sure the EV3 Brick is switched off. Insert the micro SD card you have prepared into the micro SD card slot on the EV3 Brick, as shown in Figure 1.4.

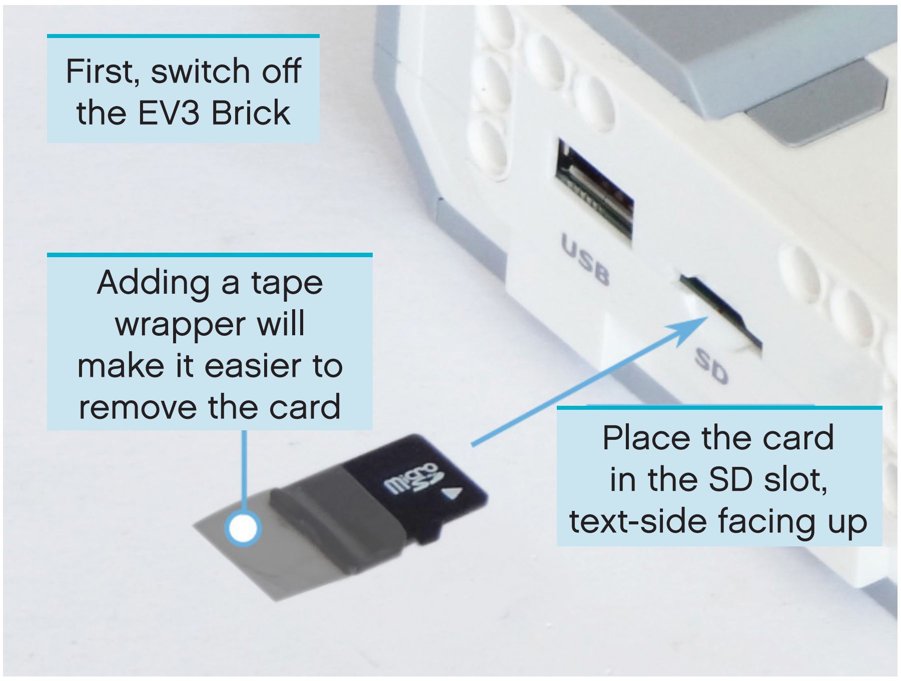

First, switch off the EV3 Brick
Adding a tape wrapper will make it easier to remove the card
Place the card in the SD slot, text-side facing up
USB
SD

Figure 1.4: Inserting the flashed micro SD card into the EV3 Brick

1.5.1 Switching the EV3 Brick on and off

Switch the EV3 Brick on by pressing the dark grey Centre Button.

The booting up process may take several minutes. While booting, the EV3 Brick Status Light turns orange and blinks intermittently and you'll see quite a bit of text on the EV3 Display. The EV3 Brick is ready for use when the Brick Status Light turns green.

To switch off the EV3 Brick, use the Back Button to open the shutdown menu and then select 'Power Off' using the Centre Button, as shown in Figure 1.5.

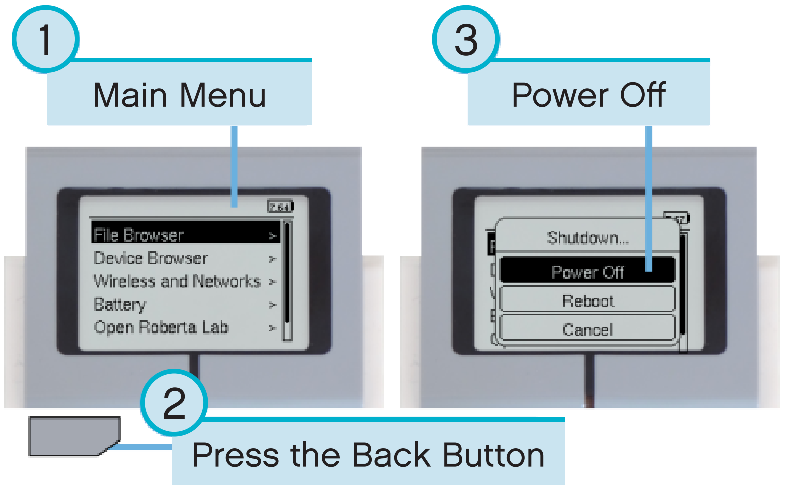


Figure 1.5: Switching off the EV3 Brick

1.5.2 Viewing motor and sensor values

When you're not running a program, you can use the device browser to view the motor and sensor values, as shown in Figure 1.6.

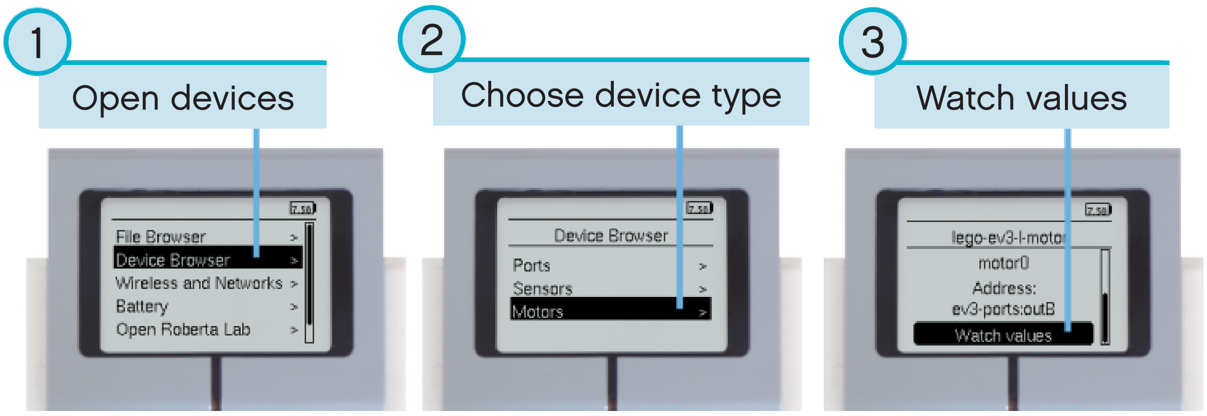


Figure 1.6: Viewing motor and sensor values

1.5.3 Reverting to the original firmware

You can revert to the LEGO® firmware and your LEGO programs at any time. To do so:

1. Switch off the EV3 Brick as shown above.
2. Wait for the Display and Brick Status Light to switch off.
3. Remove the micro SD card.
4. Switch on the EV3 Brick.

CHAPTER

TWO

CREATING AND RUNNING PROGRAMS

Now that you've set up your computer and EV3 Brick, you're ready to start writing programs.

To make it easier to create and manage your programs, let's first have a quick look at how the MicroPython projects and programs for your EV3 robots are organised.

The programs are organised into project folders, as shown in Figure 2.1. A project folder is a directory on your computer that contains the main program (**main.py**) and other optional scripts or files. This project folder and all of its contents will be copied to the EV3 Brick, where the main program will be run.

This page shows you how to create a project and how to transfer it to the EV3 Brick.

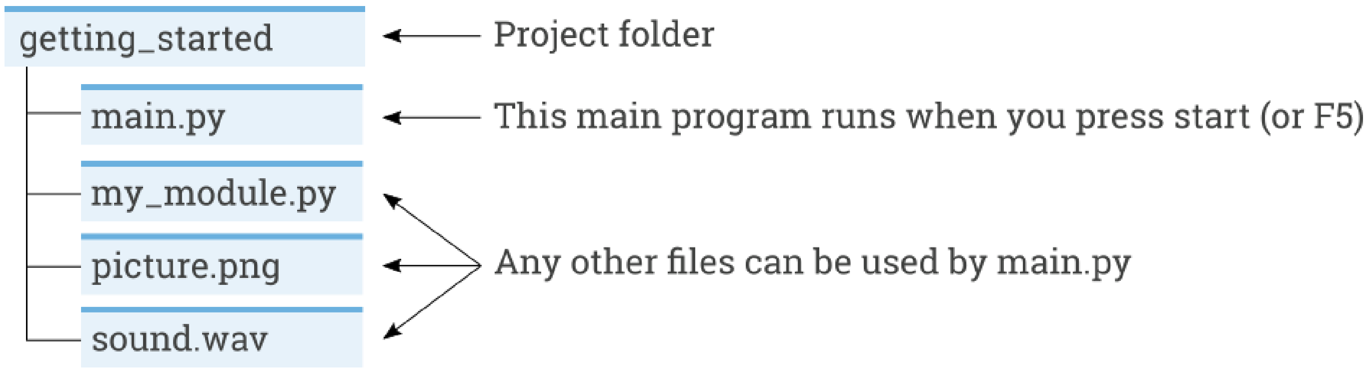

getting_started
Project folder
main.py
This main program runs when you press start (or F5)
my_module.py
picture.png
sound.wav
Any other files can be used by main.py

Figure 2.1: A project contains a program called **main.py** and optional resources like sounds or MicroPython modules.

2.1 Creating a new project

To create a new project, open the 'EV3 MicroPython' tab and click *create a new project*, as shown in Figure 2.2. Enter the name of the project in the text field that appears and press *Enter*. When prompted, choose a location in which to store this program and confirm by clicking *choose folder*.

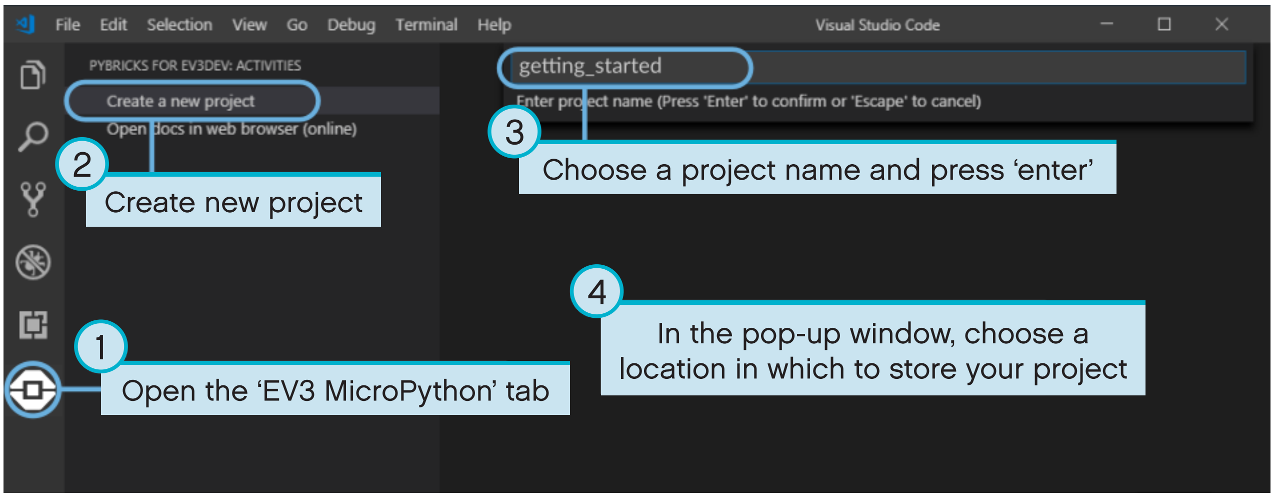
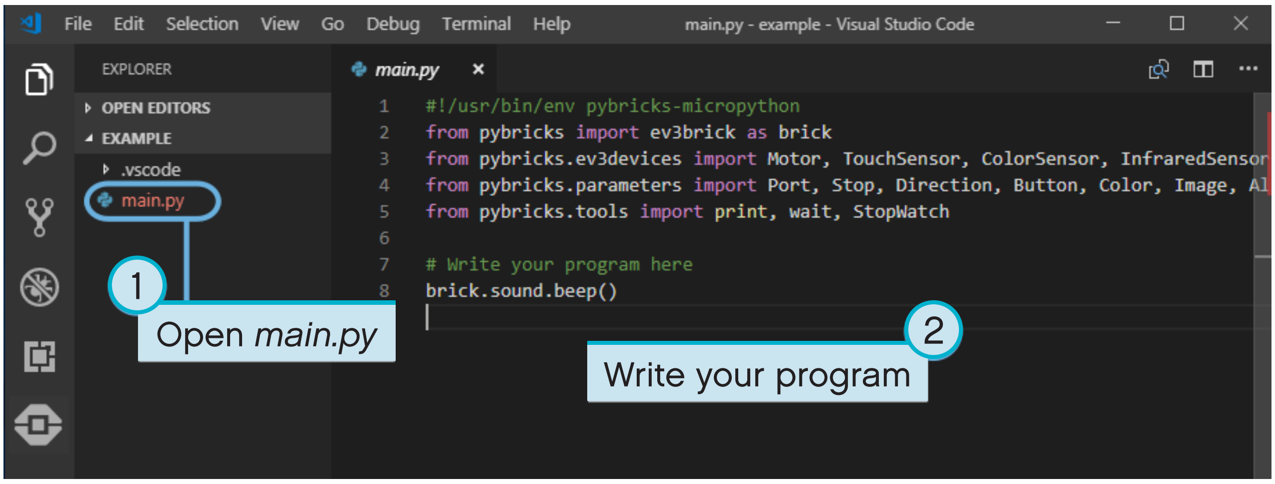

File Edit Selection View Go Debug Terminal Help
Visual Studio Code
PYBRICKS FOR EV3DEV: ACTIVITIES
Create a new project
Open docs in web browser (online)
getting_started
Enter project name (Press 'Enter' to confirm or 'Escape' to cancel)
1 Open the 'EV3 MicroPython' tab
2 Create new project
3 Choose a project name and press 'enter'
4 In the pop-up window, choose a location in which to store your project

Figure 2.2: Creating a new project. This example is called *getting_started* but you can choose any name.

When you create a new project, it already includes a file called *main.py*. To see its contents and to modify it, open it from the file browser as shown in Figure 2.3. This is where you'll write your programs.

If you're new to MicroPython programming, we recommend you keep the existing code in place and add your code to it.


File Edit Selection View Go Debug Terminal Help
main.py - example - Visual Studio Code
EXPLORER
OPEN EDITORS
EXAMPLE
.vscode
main.py
1 Open *main.py*
2 Write your program

```
#!/usr/bin/env pybricks-micropython
from pybricks import ev3brick as brick
from pybricks.ev3devices import Motor, TouchSensor, ColorSensor, InfraredSensor
from pybricks.parameters import Port, Stop, Direction, Button, Color, Image, Al
from pybricks.tools import print, wait, StopWatch

# Write your program here
brick.sound.beep()
```

Figure 2.3: Opening the default main.py program.

2.2 Opening an existing project

To open a project that you've previously created, click *File* then *Open Folder*, as shown in Figure 2.4. Next, navigate to your previously created project folder and click *OK*. You can also open your recently used projects using the *Open Recent* menu option.

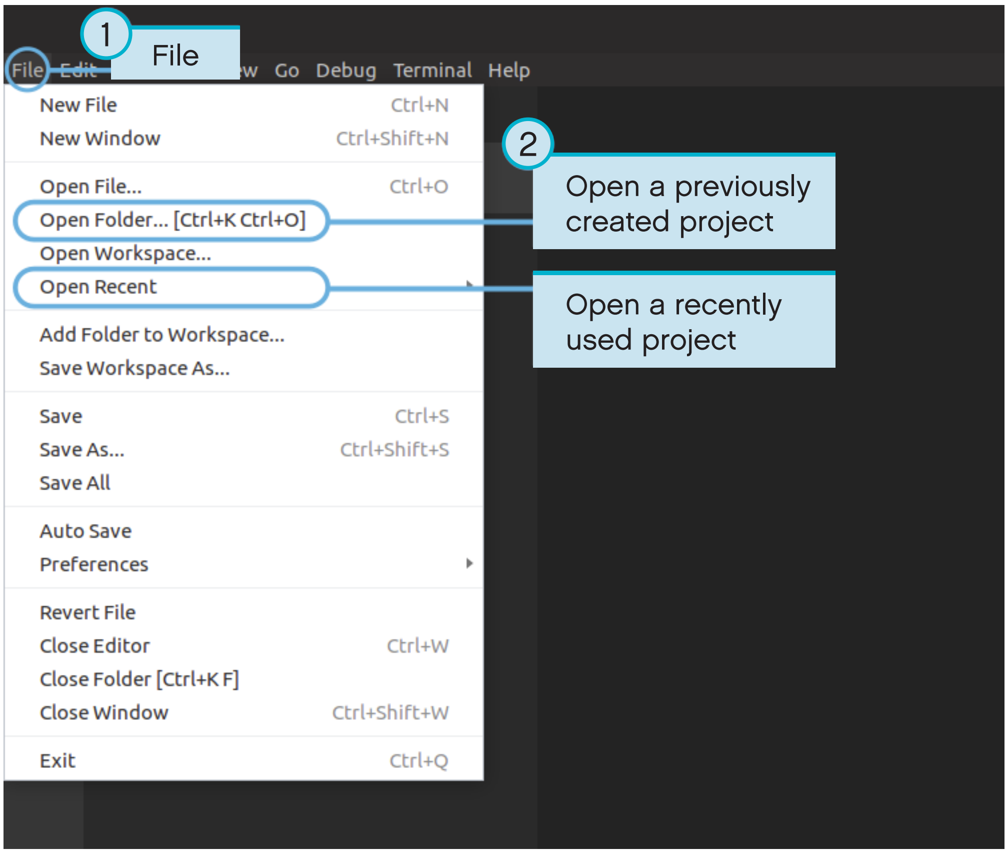


Figure 2.4: Opening a previously created project.

2.3 Connecting to the EV3 Brick with Visual Studio Code

In order to be able to transfer your code to the EV3 Brick, you'll first need to connect the EV3 Brick to your computer using the mini USB cable and configure the connection with Visual Studio Code. To do so:

- Switch on the EV3 Brick.
- Connect the EV3 Brick to your computer using the mini USB cable.
- Configure the USB connection as shown in Figure 2.5.

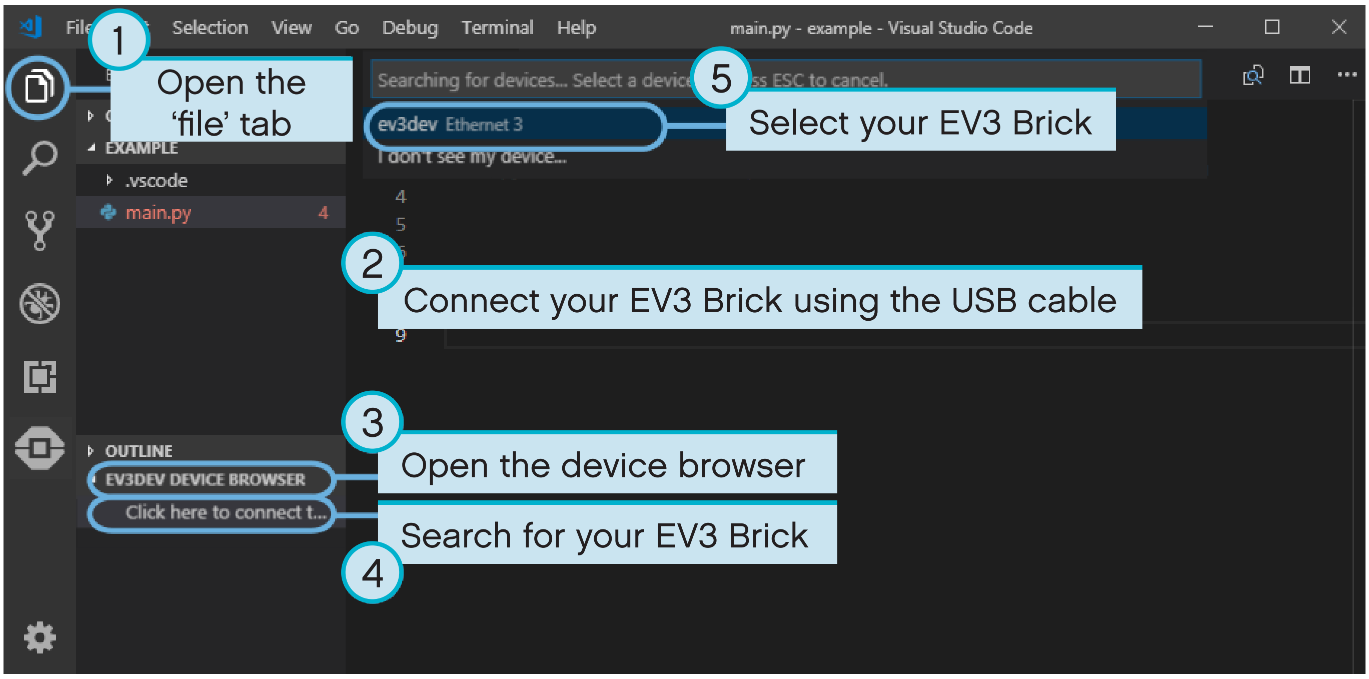


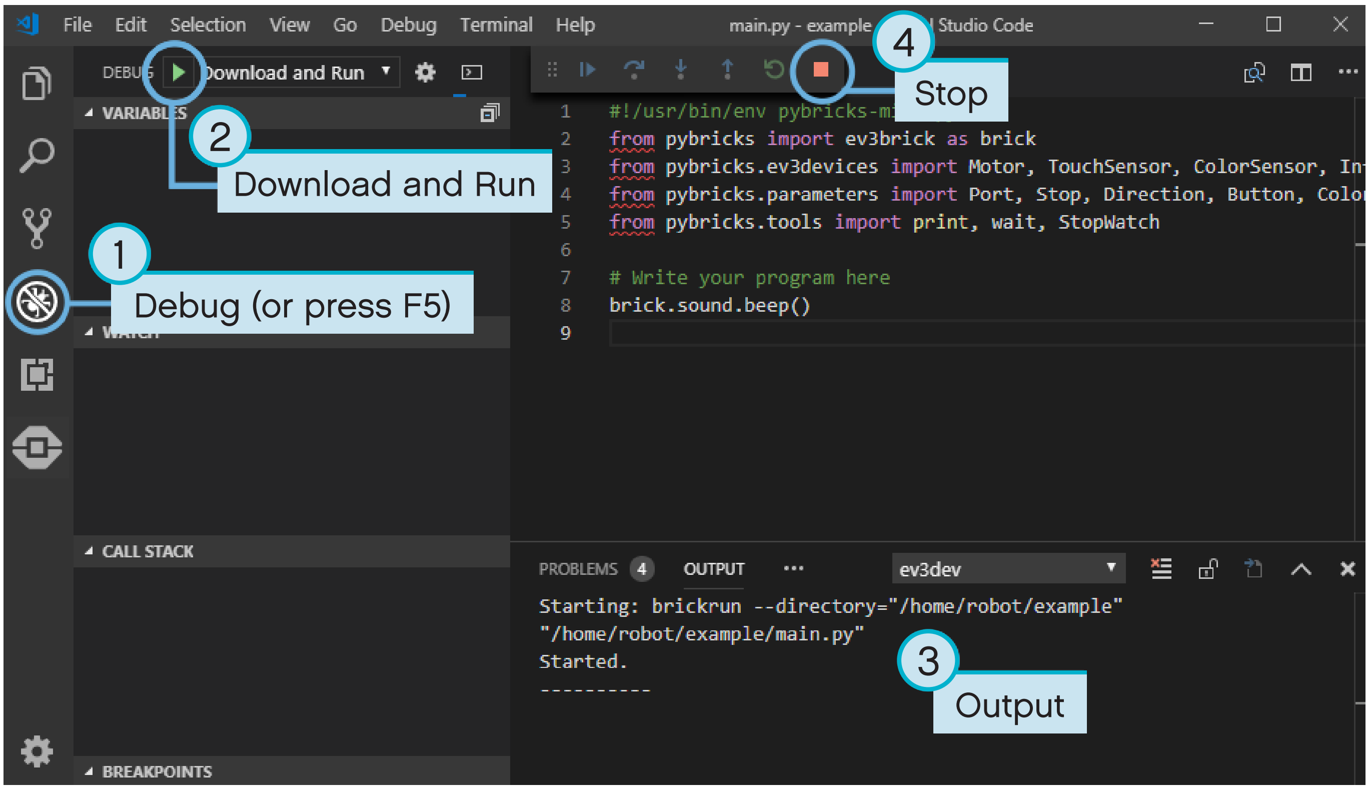
Figure 2.5: Configuring the USB connection between the computer and the EV3 Brick.

2.4 Downloading and running a program

Press the F5 key to run the program. Alternatively, you can start the program manually by going to the 'debug' tab and clicking the green 'start' arrow, as shown in Figure 2.6.

When the program starts, a pop-up toolbar allows you to stop the program if necessary. You can also stop the program at any time by using the Back Button on the EV3 Brick.

If your program produces any output with the *print* command, this is shown in the output window.



```
#!/usr/bin/env pybricks-mi
from pybricks import ev3brick as brick
from pybricks.ev3devices import Motor, TouchSensor, ColorSensor, In
from pybricks.parameters import Port, Stop, Direction, Button, Colo
from pybricks.tools import print, wait, StopWatch

# Write your program here
brick.sound.beep()
```

```
Starting: brickrun --directory="/home/robot/example"
"/home/robot/example/main.py"
Started.
----------
```

Figure 2.6: Running a program.

2.5 Expanding the example program

Now that you've run the basic code template, you can expand the program to make a motor move. First, attach a Large Motor to Port B on the EV3 Brick, as shown in Figure 2.7.

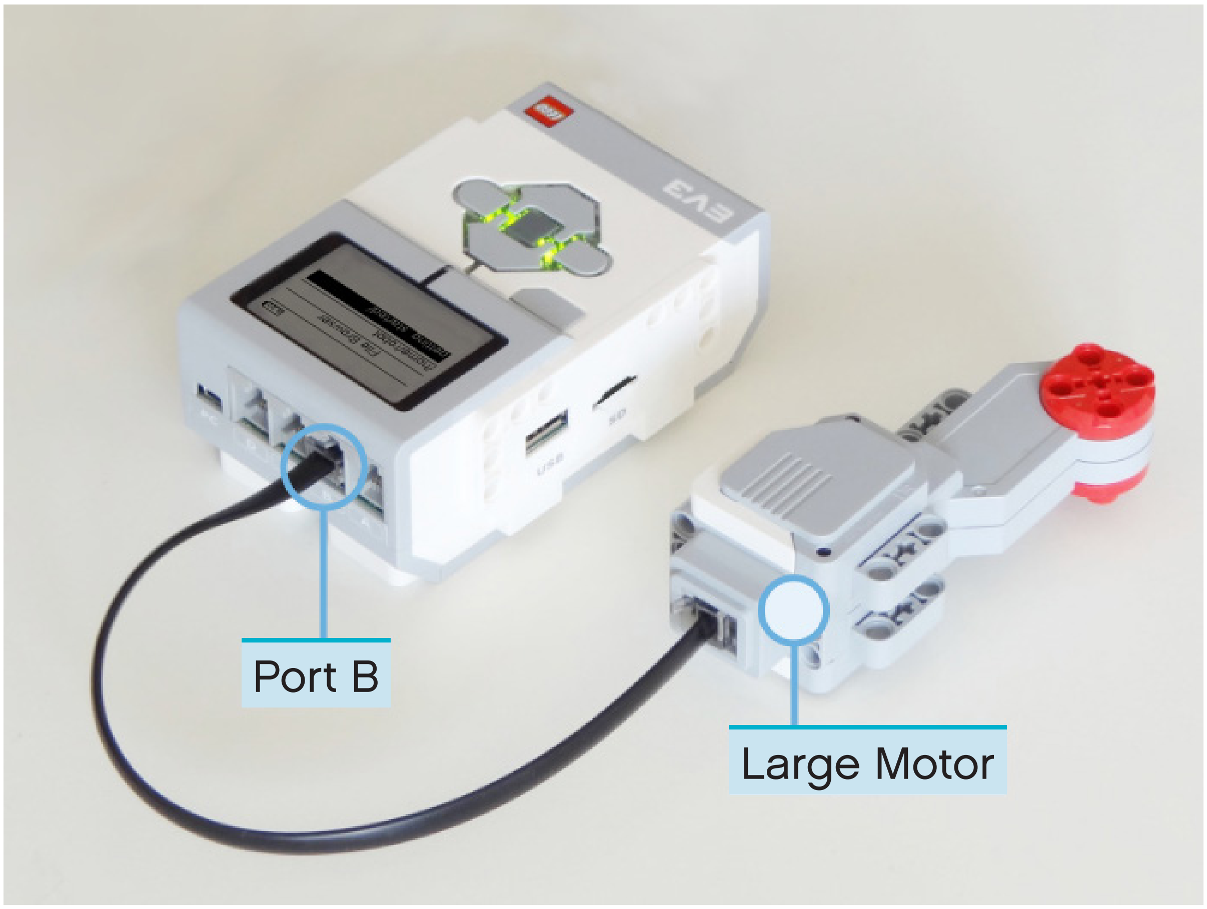


Figure 2.7: The EV3 Brick with a Large Motor attached to Port B.

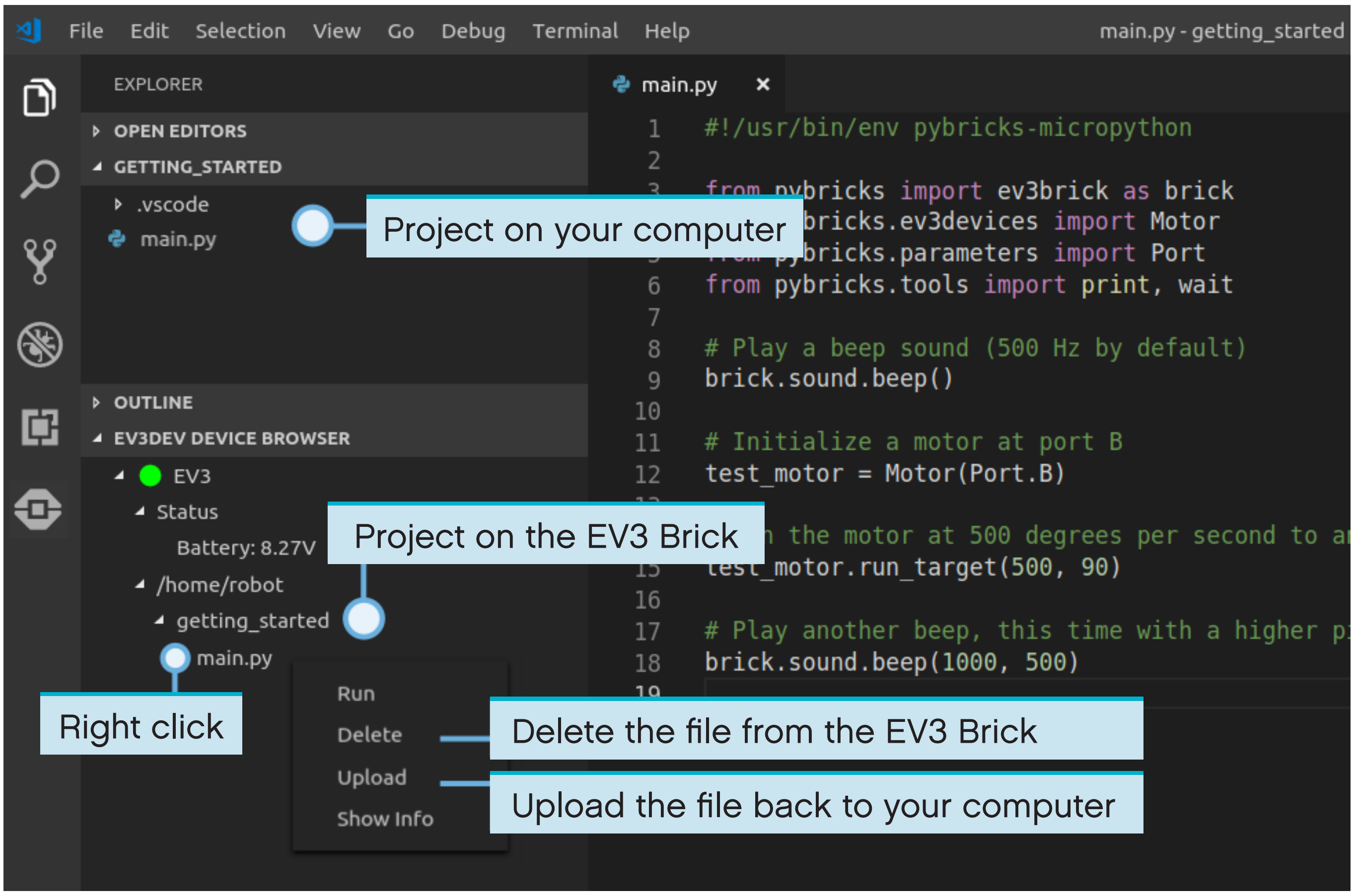
Next, edit *main.py* so that it looks like this:

```
#!/usr/bin/env pybricks-micropython

from pybricks import ev3brick as brick
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

# Play a sound.
brick.sound.beep()

# Initialize a motor at port B.
test_motor = Motor(Port.B)

# Run the motor up to 500 degrees per second. To a target angle of 90 degrees.
test_motor.run_target(500, 90)

# Play another beep sound.
# This time with a higher pitch(1000 Hz) and longer duration(500 ms).
brick.sound.beep(1000, 500)
```

This program makes your robot beep, rotate the motor and beep again at a higher-pitched tone.
Run the program to make sure it works as you expected.

2.6 Managing files on the EV3 Brick

After you've downloaded a project to the EV3 Brick, you can run, delete or back up the programs stored on it using the device browser as shown in Figure 2.8.



```
#!/usr/bin/env pybricks-micropython

from pybricks import ev3brick as brick
     bricks.ev3devices import Motor
     bricks.parameters import Port
from pybricks.tools import print, wait

# Play a beep sound (500 Hz by default)
brick.sound.beep()

# Initialize a motor at port B
test_motor = Motor(Port.B)

 n the motor at 500 degrees per second to a
test_motor.run_target(500, 90)

# Play another beep, this time with a higher p
brick.sound.beep(1000, 500)
```

Figure 2.8: Using the EV3 device browser to manage files on your EV3 Brick.

2.7 Running a program without a computer

You can run previously downloaded programs directly from the EV3 Brick.

To do so, find the program using the *file browser* on the EV3 Display and press the Centre Button to start the program as shown in Figure 2.9.

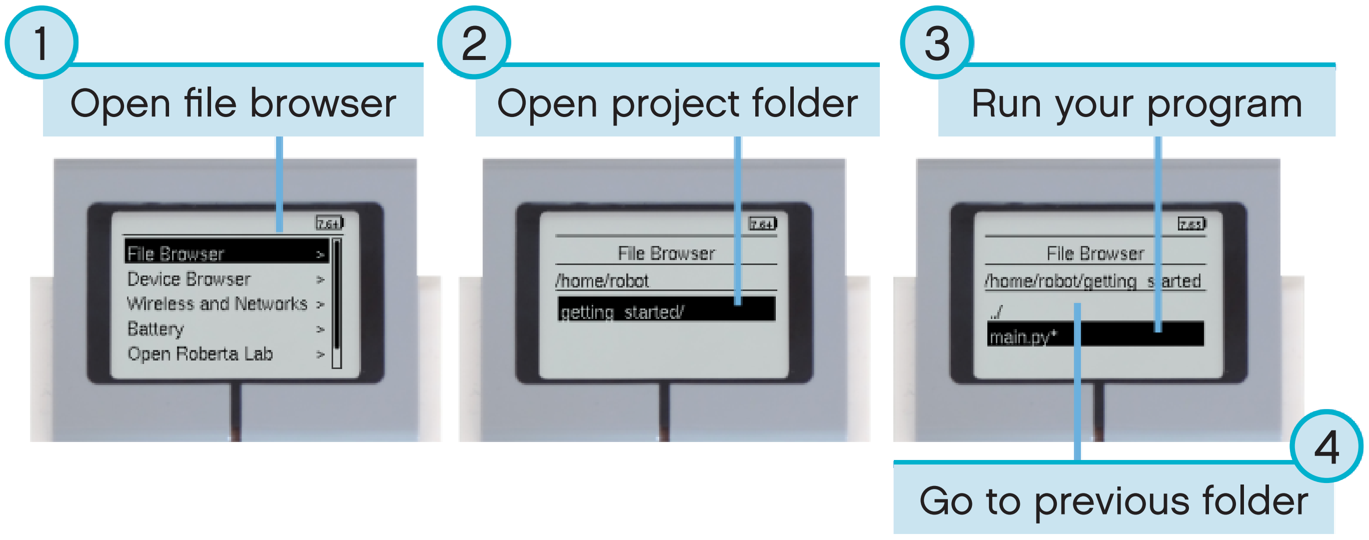


Figure 2.9: Starting a program using the buttons on the EV3 Brick.

CHAPTER THREE

`EV3 BRICK` – THE EV3 PROGRAMMABLE BRICK

3.1 Buttons

`buttons` ()

Check which buttons are currently pressed on the EV3 Brick.

Returns List of pressed buttons.

Return type List of *`Buttons`*

Examples:

```
# Do something if the left button is pressed
if Button.LEFT in brick.buttons():
    print("The left button is pressed.")
```

```
# Wait until any of the buttons are pressed
while not any(brick.buttons()):
    wait(10)

# Wait until all buttons are released
while any(brick.buttons()):
    wait(10)
```

3.2 Light

`light`(*color*)

Set the colour of the brick light.

Parameters **`color`** (`Color`) – Colour of the light. Choose `Color.BLACK` or `None` to switch the light off.

Example:

```
# Make the light red
brick.light(Color.RED)

# Turn the light off
brick.light(None)
```

3.3 Sound

classmethod sound.**beep** (*frequency=500, duration=100, volume=30*)

Play a beep/tone.

Parameters

- **frequency** (*frequency: Hz*) – Frequency of the beep (*Default*: 500).
- **duration** (*time: ms*) – Duration of the beep (*Default*: 100).
- **volume** (*percentage: %*) – Volume of the beep (*Default*: 30).

Example:

```
# A simple beep
brick.sound.beep()

# A high pitch(1500 Hz) for one second(1000 ms) at 50% volume
brick.sound.beep(1500, 1000, 50)
```

classmethod sound.**beeps** (*number*)

Play a number of default beeps with a brief pause in between.

Parameters number(*int***)** – Number of beeps.

Example:

```
# Make 5 simple beeps
brick.sound.beeps(5)
```

classmethod sound.**file** (*file_name, volume=100*)

Play a sound file.

Parameters

- **file_name** (*str*) – Path to the sound file, including extension.
- **volume** (*percentage: %*) – Volume of the sound (*Default*: 100).

Example:

```
# Play one of the built-in sounds
brick.sound.file(SoundFile.HELLO)

# Play a sound file from your project folder
brick.sound.file('mysound.wav')
```

3.4 Display

```
                      X
          ---------------->
(0, 0)  _______________________
        |                     |
    |   |                     |
y   |   |        Hello        |
    |   |        World        |
    v   |                     |
        |_____________________| (177, 127)
```

classmethod display.**clear**()

Clear everything on the Display.

classmethod display.**text**(*text, coordinate=None*)

Display text.

Parameters

- **text** (*str*) – The text to display.
- **coordinate** (*tuple*) – (x, y) Coordinate tuple. It is the top-left corner of the first character. If no coordinate is specified, it is printed on the next line.

Example:

```
# Clear the display
brick.display.clear()

# Print ``Hello`` near the middle of the screen
brick.display.text("Hello",(60, 50))

# Print ``World`` directly underneath it
brick.display.text("World")
```

classmethod display.**image**(*file_name, alignment=Align.CENTER, coordinate=None, clear=True)*

Show an image file.

You can specify its placement using either `alignment` or by specifying a `coordinate`, but not both.

Parameters

- **file_name** (*str*) – Path to the image file. Paths may be absolute or relative from the project folder.
- **alignment** (Align) – Where to place the image (*Default*: Align.CENTER).
- **coordinate** (*tuple*) – (x, y) Coordinate tuple. It is the top-left corner of the image (*Default*: None).
- **clear** (*bool*) – Whether to clear the Display before showing the image (*Default*: True).

Example:

```
# Show a built-in image of two eyes looking upward
brick.display.image(ImageFile.UP)

# Display a custom image from your project folder
brick.display.image('pybricks.png')

# Display a custom image at the top right of the screen, without clearing
# the screen first
brick.display.image('arrow.png', Align.TOP_RIGHT, clear=False)
```

3.5 Battery

classmethod battery.**voltage**()

Retrieve the voltage of the battery.

Returns Battery voltage.

Return type *voltage: mV*

Examples:

```
# Play a warning sound when the battery voltage
# is below 7 Volt(7000 mV = 7V)
if brick.battery.voltage() < 7000:
    brick.sound.beep()
```

classmethod battery.**current**()

Retrieve the current supplied by the battery.

Returns Battery current.

Return type *current: mA*

CHAPTER

FOUR

`EV3 DEVICES` – EV3 MOTORS AND SENSORS

4.1 Motors

`class Motor`(*port, direction=Direction.CLOCKWISE, gears=None*)

LEGO® MINDSTORMS® EV3 Medium or Large Motor.

Element 99455/6148292 or 95658/6148278, contained in:

- 31313: LEGO MINDSTORMS EV3 (2013)
- 45544: LEGO MINDSTORMS Education EV3 Core Set (2013)
- 45503 or 45502: Separate part (2013)

Parameters

- **port** (`Port`) – Port to which the motor is connected.
- **direction** (`Direction`) – Positive speed direction (*Default*: Direction.CLOCKWISE).
- **gears** (*`list`*) – List of gears that are linked to the motor (*Default*: `None`).

  For example: [`12, 36`] represents a gear train with a 12-tooth and a 36-tooth gear. See ratio for illustrated examples.

  Use a list of lists for multiple gear trains, such as `[[12, 36], [20, 16, 40]]`.

  When you specify a gear train, all motor commands and settings are automatically adjusted to account for the resulting gear ratio. The direction of the motor remains unchanged regardless of the number of gears you choose.

  For example, with `gears=[12, 36]`, the gear ratio is 3, which means that the output is mechanically slowed by a factor of 3. To compensate, the motor will automatically turn 3 times as fast and 3 times as far when you give a motor command. So when you choose `run_angle(200, 90)`, your mechanism output simply turns at 200 deg/s for 90 degrees.

  The same holds for the documentation below. When it states 'motor angle' or 'motor speed', you can read this as 'mechanism output angle', 'mechanism output speed' and so on, as the gear ratio is automatically accounted for.

  The `gears` setting is only available for motors with Rotation Sensors.

Example:

```
# Initialize a motor(by default this means clockwise, without any gears).
example_motor = Motor(Port.A)

# Initialize a motor where positive speed values should go counterclockwise
right_motor = Motor(Port.B, Direction.COUNTERCLOCKWISE)

# Initialize a motor with a gear train
robot_arm = Motor(Port.C, Direction.CLOCKWISE, [12, 36])
```

Methods for Motors without Rotation Sensors

dc(*duty*)
Set the duty cycle of the motor.

Parameters **duty**(*percentage: %*) – The duty cycle (-100.0 to 100).

Example:

```
# Set the motor duty cycle to 75%.
example_motor.duty(75)
```

Methods for Motors with Rotation Sensors

angle()
Get the rotation angle of the motor.

Returns Motor angle.

Return type *angle: deg*

reset_angle (*angle*)
Reset the accumulated rotation angle of the motor.

Parameters **angle**(*angle: deg*) – Value to which the angle should be reset.

speed()
Get the speed (angular velocity) of the motor.

Returns Motor speed.

Return type *rotational speed: deg/s*

stop(*stop__type=Stop.COAST*)
Stop the motor.

Parameters **stop_type**(Stop) – Whether to coast, brake or hold (*Default*: *Stop.COAST*).

run(*speed*)
Keep the motor running at a constant speed (angular velocity).

The motor will accelerate towards the requested speed and the duty cycle is automatically adjusted in order to keep the speed constant, even under some load. This continues in the background until you give the motor a new command or the program stops.

Parameters **speed**(*rotational speed: deg/s*) – Speed of the motor.

run_time(*speed, time, stop__type=Stop.COAST, wait=True*)
Run the motor at a constant speed (angular velocity) for a given length of time.

The motor will accelerate towards the requested speed and the duty cycle is automatically adjusted in order to keep the speed constant, even under some load. It begins to decelerate just in time to reach a standstill after the specified duration.

Parameters

- **`speed`** (*rotational speed: deg/s*) – Speed of the motor.
- **`time`** (*time: ms*) – Duration of the manoeuvre.
- **`stop_type`** (`Stop`) – Whether to coast, brake or hold after coming to a standstill (*Default*: *`Stop.COAST`*).
- **`wait`** (*`bool`*) – Wait for the manoeuvre to complete before continuing with the rest of the program (*Default*: `True`). This means your program waits for the specified `time`.

`run_angle` (*speed, rotation_angle, stop_type=Stop.COAST, wait=True*)
Run the motor at a constant speed (angular velocity) by a given angle.

The motor will accelerate towards the requested speed and the duty cycle is automatically adjusted in order to keep the speed constant, even under some load. It begins to decelerate just in time to come to a standstill after traversing the given angle.

Parameters

- **`speed`** (*rotational speed: deg/s*) – Speed of the motor.
- **`rotation_angle`** (*angle: deg*) – Angle by which the motor should rotate.
- **`stop_type`** (`Stop`) – Whether to coast, brake or hold after coming to a standstill (*Default*: *`Stop.COAST`*).
- **`wait`** (*`bool`*) – Wait for the manoeuvre to complete before continuing with the rest of the program (*Default*: `True`). This means that your program waits until the motor has travelled precisely the requested angle.

`run_target` (*speed, target_angle, stop_type=Stop.COAST, wait=True*)
Run the motor at a constant speed (angular velocity) towards a given target angle.

The motor will accelerate towards the requested speed and the duty cycle is automatically adjusted in order to keep the speed constant, even under some load. It begins to decelerate just in time to come to a standstill at the given target angle.

The direction of rotation is automatically selected based on the target angle.

Parameters

- **`speed`** (*rotational speed: deg/s*) – Absolute speed of the motor. The direction will be selected automatically based on the target angle: it makes no difference whether you specify a positive or negative speed.
- **`target_angle`** (*angle: deg*) – Target angle that the motor should rotate to, regardless of its current angle.
- **`stop_type`** (`Stop`) – Whether to coast, brake or hold after coming to a standstill (*Default*: *`Stop.COAST`*).
- **`wait`** (*`bool`*) – Wait for the manoeuvre to complete before continuing with the rest of the program (*Default*: `True`). This means that your program waits until the motor has reached the target angle.

Advanced Methods for Motors with Rotation Sensors

`track_target` (*target_angle*)

Track a target angle that varies in time.

This function is quite similar to *`run_target()`* but speed and acceleration settings are ignored. It will move to the target angle as quickly as possible. Instead, you adjust speed and acceleration by choosing how quickly or slowly you vary the `target_angle`.

This method is useful in fast loops where the motor target changes continuously.

Parameters **`target_angle`** (*angle: deg*) – Target angle that the motor should rotate to.

Example:

```
# Initialize motor and timer
from math import sin
motor = Motor(Port.A)
watch = StopWatch()
amplitude = 90

# In a fast loop, compute a reference angle
# and make the motor track it.

while True:

    # Get the time in seconds
    seconds = watch.time()/1000
    # Compute a reference angle. This produces
    # a sine wave that makes the motor move
    # smoothly between -90 and +90 degrees.
    angle_now = sin(seconds)*amplitude
    # Make the motor track the given angle
    motor.track_target(angle_now)
```

`stalled`()

Check whether the motor is currently stalled.

A motor is stalled when it cannot move even with the maximum torque. For example, when something is blocking the motor or your mechanism simply cannot turn any further.

Specifically, the motor is stalled when the duty cycle computed by the PID controllers has reached the maximum (so `duty = duty_limit`) and still the motor cannot reach a minimal speed (so `speed < stall_speed`) for a period of at least `stall_time`.

You can change the `duty_limit`, `stall_speed` and `stall_time` settings using *`set_dc_settings()`* and *`set_pid_settings()`* in order to change the sensitivity to being stalled.

Returns `True` if the motor is stalled, `False` if it is not.

Return type bool

`run_until_stalled` (*speed, stop_type=Stop.COAST, duty_limit=default*)

Run the motor at a constant speed (angular velocity) until it stalls. The motor is considered stalled when it cannot move even with the maximum torque. See *`stalled()`* for a more precise definition.

The `duty_limit` argument lets you temporarily limit the motor torque during this manoeuvre. This is useful in order to avoid applying the full motor torque to a geared or lever mechanism.

Parameters

- **speed** (*rotational speed: deg/s*) – Speed of the motor.
- **stop_type** (Stop) – Whether to coast, brake or hold after coming to a standstill (*Default*: *Stop.COAST*).
- **duty_limit** (*percentage: %*) – Relative torque limit. This limit works just like *set_dc_settings()* but in this case the limit is temporary. It returns to its previous value after completing this command.

set_dc_settings (*duty_limit, duty_offset*)
Configure the settings to adjust the behaviour of the *dc()* command. This also affects all of the run commands, which use the *dc()* method in the background.

Parameters

- **duty_limit** (*percentage: %*) – Relative torque limit during subsequent motor commands. This sets the maximum duty cycle that is applied during any subsequent motor command. This reduces the maximum torque output to a percentage of the absolute maximum stall torque. This is useful in order to avoid applying the full motor torque to a geared or lever mechanism or to prevent your LEGO® train from unintentionally going at full speed. (*Default*: 100).
- **duty_offset** (*percentage: %*) – Minimum duty cycle given when you use *dc()*. This adds a small feed forward torque so that your motor will move even for very low duty cycle values, which can be useful when you create your own feedback controllers (*Default*: 0).

set_run_settings (*max_speed, acceleration*)
Configure the maximum speed and acceleration/deceleration of the motor for all run commands.

This applies to the run, run_time, run_angle, run_target or run_until_stalled commands that you give the motor. See also the default parameters for each motor.

Parameters

- **max_speed** (*rotational speed: deg/s*) – Maximum speed of the motor during a motor command.
- **acceleration** (*rotational acceleration: deg/s/s*) – Acceleration towards the target speed and deceleration towards standstill. This should be a positive value. The motor will automatically change the sign in order to decelerate as needed.

Example:

```
# Set the maximum speed to 200 deg/s and acceleration to 400 deg/s/s.
example_motor.set_run_settings(200, 400)

# Make the motor run for 5 seconds. Even though the speed argument is 300
# deg/s in this example, the motor will move at only 200 deg/s because of
# the settings above.
example_motor.run_time(300, 5000)
```

set_pid_settings (*kp, ki, kd, tight_loop_limit, angle_tolerance, speed_tolerance, stall_speed, stall_time*)
Configure the settings of the position and speed controllers. See also pid and the default parameters for each motor.

Parameters

- **kp** (int) – Proportional position (and integral speed) control constant.
- **ki** (int) – Integral position control constant.
- **kd** (int) – Derivative position (and proportional speed) control constant.

- **`tight_loop_limit`** (*time: ms*) – If you execute any of the run commands within this interval after starting the previous command, the controllers assume that you wish to control the speed directly. This means that it will ignore the acceleration setting and immediately begin tracking the speed that you give in the run command. This is useful in a fast loop, where you usually want the motors to respond quickly rather than to accelerate smoothly, for example with a line-following robot.
- **`angle_tolerance`** (*angle: deg*) – Allowed deviation from the target angle before motion is considered complete.
- **`speed_tolerance`** (*rotational speed: deg/s*) – Allowed deviation from zero speed before motion is considered complete.
- **`stall_speed`** (*rotational speed: deg/s*) – See `stalled()`.
- **`stall_time`** (*time: ms*) – See `stalled()`.

4.2 Sensors

4.2.1 Touch Sensor

`class TouchSensor` (*port*)

Element 95648/6138404

- 31313: LEGO MINDSTORMS EV3 (2013)
- 45544: LEGO MINDSTORMS Education EV3 Core Set (2013)
- 45507: As a separate part (2013)

Parameters **`port`** (`Port`) – Port to which the sensor is connected.

`pressed` ()

Check whether the sensor is pressed.

Returns `True` if the sensor is pressed, `False` if it is not pressed.

Return type `bool`

4.2.2 Colour Sensor

`class ColorSensor` (*port*)

Element 95650/6128869, contained in:

- 31313: LEGO MINDSTORMS EV3 (2013)
- 45544: LEGO MINDSTORMS Education EV3 Core Set (2013)
- 45506: As a separate part (2013)

Parameters **`port`** (`Port`) – Port to which the sensor is connected.

`color` ()

Measure the colour of a surface.

Returns `Color.BLACK`, `Color.BLUE`, `Color.GREEN`, `Color.YELLOW`, `Color.RED`, `Color.WHITE`, `Color.BROWN` or `None`.

Return type `Color`, or `None` if no colour is detected.

`ambient` ()
Measure the Ambient Light Intensity.

Returns Ambient Light Intensity, ranging from 0 (dark) to 100 (bright).

Return type *percentage: %*

`reflection` ()
Measure the reflection of a surface using a red light.

Returns Reflection, ranging from 0 (no reflection) to 100 (high reflection).

Return type *percentage: %*

`rgb` ()
Measure the reflection of a surface using a red, green and then a blue light.

Returns Reflection for red, green and blue light, each ranging from 0.0 (no reflection) to 100.0 (high reflection).

Return type tuple of three *percentages*

4.2.3 Infrared Sensor and Beacon

`class InfraredSensor` (*port*)

Elements 95654/6132629 and 72156/6127283, contained in:

- 31313: LEGO MINDSTORMS EV3 (2013)
- 45509 and 45508: As separate parts (2013)

Parameters **`port`** (`Port`) – Port to which the sensor is connected.

`distance` ()
Measure the relative distance between the sensor and an object using infrared light.

Returns Relative distance ranging from 0 (closest) to 100 (farthest).

Return type *relative distance: %*

`beacon` (*channel*)
Measure the relative distance and angle between the remote and the Infrared Sensor.

Parameters **`channel`** (*`int`*) – Channel number of the remote.

Returns Tuple of relative distance (0 to 100) and approximate angle (-75 to 75 degrees) between remote and Infrared Sensor.

Return type (*relative distance: %, angle: deg*) or (`None`, `None`) if no remote is detected.

`buttons` (*channel*)
Check which buttons are pressed on the infrared remote.

Parameters **`channel`** (*`int`*) – Channel number of the remote.

Returns List of pressed buttons on the remote on the specified channel.

Return type List of *`Buttons`*

4.2.4 Ultrasonic Sensor

class UltrasonicSensor (*port*)

Element 95652/6138403, contained in:

- 45544: LEGO MINDSTORMS Education EV3 Core Set (2013)
- 45504: As aeparate part (2013)

Parameters port (`Port`) – Port to which the sensor is connected.

distance (*silent=False*)

Measure the distance between the sensor and an object using ultrasonic sound waves.

Parameters silent (*bool*) – Choose `True` to turn the sensor off after measuring the distance.

Choose `False` to leave the sensor on (*Default*).

When you choose `silent=True`, the sensor does not emit sound waves except when it is taking the measurement. This reduces interference with other Ultrasonic Sensors, but switching off the sensor takes approximately 300 ms each time.

Returns Distance (millimetres).

Return type *distance: mm*

presence ()

Check for the presence of other Ultrasonic Sensors by detecting ultrasonic sounds.

If the other Ultrasonic Sensor is operating in silent mode, you can only detect the presence of that sensor while it is taking a measurement.

Returns `True` if ultrasonic sounds are detected, `False` if not.

Return type `bool`

4.2.5 Gyroscopic Sensor

class GyroSensor (*port, direction=Direction.CLOCKWISE*)

Element 99380/6138411, contained in:

- 45544: LEGO MINDSTORMS Education EV3 Core Set (2013)
- 45505: As a separate part (2013)

Parameters

- **port** (`Port`) – Port to which the sensor is connected.
- **direction** (`Direction`) – Positive rotation direction when looking at the red dot on top of the sensor (*Default*: Direction.CLOCKWISE).

speed ()

Get the speed (angular velocity) of the sensor.

Returns sensor angular velocity.

Return type *rotational speed: deg/s* `angle` ()

Get the accumulated angle of the sensor.

Returns rotation angle.

Return type *angle: deg*

`reset_angle` (*angle*)

Set the rotation angle of the sensor to a desired value.

Parameters `angle` (*angle: deg*) – Value to which the angle should be reset.

CHAPTER

FIVE

`PARAMETERS` – PARAMETERS AND CONSTANTS

Constant parameters/arguments for the Pybricks API.

`class Port`

Motor Ports:

`A`

`B`

`C`

`D`

Sensor Ports:

`S1`

`S2`

`S3`

`S4`

`class Direction`
Rotational direction for positive speed values: clockwise or anticlockwise.

`CLOCKWISE`
A positive speed value should make the motor move clockwise.

`COUNTERCLOCKWISE`
A positive speed value should make the motor move anticlockwise.

For all motors, this is defined when looking at the shaft, just like looking at a clock.

For NXT or EV3 motors, make sure you look at the motor with the red/orange shaft to the lower right.

Parameter	Positive Speed	Negative Speed
Direction.CLOCKWISE	clockwise	counterclockwise
Direction.COUNTERCLOCKWISE	counterclockwise	clockwise

```
Medium EV3 Motor:
    counterclockwise         clockwise
        _____                  ______
       /                             \
      /       _______________         \
     /        /               \         \
     |      |          _        |        |
     |      |      ___| |___    |        |
     v      |     |___ o ___|   |        v
            |         |_|       |
            |                   |
              \_______________/
Large EV3 Motor:
       _________
      /          \         ____    ____
     _|           \       /           \
     |               ----/------        \
     counterclockwise    |    ___\___    |  clockwise
       \____________     v  /       \    v
                     -------|   +   |
                            \______/
```

`class Stop`

Action after the motor stops: coast, brake or hold.

`COAST`

Let the motor move freely.

`BRAKE`

Passively resist small external forces.

`HOLD`

Keep controlling the motor to hold it at the commanded angle. This is only available on motors with encoders.

The 'stop type' defines the resistance to motion after coming to a standstill:

Parameter	Resistance	Physical Meaning
Stop.COAST	Low	Friction
Stop.BRAKE	Medium	Friction + Torque opposite to motion
Stop.HOLD	High	Friction + Torque to hold commanded angle

`class Color`

Light or surface colour.

`BLACK`

`BLUE`

`GREEN`

`YELLOW`

`RED`

`WHITE`

`BROWN`

`ORANGE`

`PURPLE`

`class Button`

Buttons on a brick or remote:

`LEFT_DOWN`

`DOWN`

`RIGHT_DOWN`

`LEFT`

`CENTER`

`RIGHT`

`LEFT_UP`

`UP`

`BEACON`

`RIGHT_UP`

LEFT_UP	UP/BEACON	RIGHT_UP
LEFT	CENTER	RIGHT
LEFT_DOWN	DOWN	RIGHT_DOWN

`class Align`

Alignment of an image on the display.

`BOTTOM_LEFT`

`BOTTOM`

`BOTTOM_RIGHT`

`LEFT`

`CENTER`

`RIGHT`

`TOP_LEFT`

`TOP`

`TOP_RIGHT`

`class ImageFile`

Paths to standard EV3 images.

`Information`

`RIGHT`

`FORWARD`

`ACCEPT`

`QUESTION_MARK`

`STOP_1`

`LEFT`

`DECLINE`

`THUMBS_DOWN`

`BACKWARD`

`NO_GO`

`WARNING`

`STOP_2`

`THUMBS_UP`

LEGO

`EV3`

`EV3_ICON`

Objects

`TARGET`

Eyes

`BOTTOM_RIGHT`

`BOTTOM_LEFT`

`EVIL`

`CRAZY_2`

`KNOCKED_OUT`

`PINCHED_RIGHT`

`WINKING`

`DIZZY`

`DOWN`

`TIRED_MIDDLE`

`MIDDLE_RIGHT`

`SLEEPING`

`MIDDLE_LEFT`

`TIRED_RIGHT`

`PINCHED_LEFT`

`PINCHED_MIDDLE`

`CRAZY_1`

`NEUTRAL`

`AWAKE`

`UP`

`TIRED_LEFT`

`ANGRY`

`class SoundFile`

Paths to standard EV3 sounds.

Expressions

`SHOUTING`

`CHEERING`

`CRYING`

`OUCH`

`LAUGHING_2`

`SNEEZING`

`SMACK`

`BOING`

`BOO`

`UH_OH`

`SNORING`

`KUNG_FU`

`FANFARE`

`CRUNCHING`

`MAGIC_WAND`

`LAUGHING_1`

Information

`LEFT`

`BACKWARDS`

`RIGHT`

`OBJECT`

`COLOR`

`FLASHING`

`ERROR`

`ERROR_ALARM`

`DOWN`

`FORWARD`

`ACTIVATE`

`SEARCHING`

`TOUCH`

`UP`

`ANALYZE`

`STOP`

`DETECTED`

`TURN`

`START`

Communication

`MORNING`

`EV3`

`GO`

`GOOD_JOB`

`OKEY_DOKEY`

`GOOD`

`NO`

`THANK_YOU`

`YES`

`GAME_OVER`

`OKAY`

`SORRY`

`BRAVO`

`GOODBYE`

`HI`

`HELLO`

`MINDSTORMS`

`LEGO`

`FANTASTIC`

Movements

`SPEED_IDLE`

`SPEED_DOWN`

`SPEED_UP`

Colour

`BROWN`

`GREEN`

`BLACK`

`WHITE`

`RED`

`BLUE`

`YELLOW`

Mechanical

`TICK_TACK`

`HORN_1`

`BACKING_ALERT`

`MOTOR_IDLE`

`AIR_RELEASE`

`AIRBRAKE`

`RATCHET`

`MOTOR_STOP`

`HORN_2`

`LASER`

`SONAR`

`MOTOR_START`

Animals

`INSECT_BUZZ_2`

`ELEPHANT_CALL`

`SNAKE_HISS`

`DOG_BARK_2`

`DOG_WHINE`

`INSECT_BUZZ_1`

`DOG_SNIFF`

`T_REX_ROAR`

`INSECT_CHIRP`

`DOG_GROWL`

`SNAKE_RATTLE`

`DOG_BARK_1`

`CAT_PURR`

Numbers

`ZERO`

`ONE`

`TWO`

`THREE`

`FOUR`

`FIVE`

`SIX`

`SEVEN`

`EIGHT`

`NINE`

`TEN`

System

`READY`

`CONFIRM`

`GENERAL_ALERT`

`CLICK`

`OVERPOWER`

CHAPTER

SIX

TOOLS – TIMING AND DATALOGGING

Common tools for timing and datalogging.

print (*value, ..., sep, end, file, flush*)

Print values on the terminal or a stream.

Example:

```
# Print some text
print("Hello, world")

# Print some text and a number
print("Value:", 5)
```

wait (*time*)

Pause the user program for a specified length of time.

Parameters **time** (*time: ms*) – How long to wait.

class StopWatch

A stopwatch to measure time intervals. Similar to the stopwatch feature on your smartphone.

time ()

Get the current time of the stopwatch.

Returns Elapsed time.

Return type *time: ms*

pause ()

Pause the stopwatch.

resume ()

Resume the stopwatch.

reset ()

Reset the stopwatch time to '0'.

The run state is unaffected:

- If it was paused, it stays paused (but now at '0').
- If it was running, it stays running (but starting again from '0').

CHAPTER

SEVEN

ROBOTICS – ROBOTICS MODULE

Robotics module for the Pybricks API.

class DriveBase (*left_motor, right_motor, wheel_diameter, axle_track*)
Class representing a robotic vehicle having two powered wheels and optional castor(s).

Parameters

- **left_motor** (Motor) – The motor that drives the left wheel.
- **right_motor** (Motor) – The motor that drives the right wheel.
- **wheel_diameter** (*dimension: mm*) – Diameter of the wheels.
- **axle_track** (*dimension: mm*) – Distance between the midpoints of the two wheels.

Example:

```
# Initialize two motors and a drive base
left = Motor(Port.B)
right = Motor(Port.C)
robot = DriveBase(left, right, 56, 114)
```

drive (*speed, steering*)
Start driving at the specified speed and turn rate, both measured at the centre point between the wheels of the robot.

Parameters

- **speed** (*speed: mm/s*) – Forward speed of the robot.
- **steering** (*rotational speed: deg/s*) – Turn rate of the robot.

Example:

```
# Initialize two motors and a drive base
left = Motor(Port.B)
right = Motor(Port.C)
robot = DriveBase(left, right, 56, 114)

# Initialize a sensor
sensor = UltrasonicSensor(Port.S4)

# Drive forward until an object is detected
robot.drive(100, 0)
while sensor.distance() > 500:
    wait(10)
robot.stop()
```

`drive_time` (*speed, steering, time*)

Drive at the specified speed and turn rate for a given length of time and then stop.

Parameters

- **`speed`** (*speed: mm/s*) – Forward speed of the robot.
- **`steering`** (*rotational speed: deg/s*) – Turn rate of the robot.
- **`time`** (*time: ms*) – Duration of the manoeuvre.

Example:

```
# Drive forward at 100 mm/s for two seconds
robot.drive(100, 0, 2000)

# Turn at 45 deg/s for three seconds
robot.drive(0, 45, 3000)
```

`stop` (*stop__type=Stop.COAST*)

Stop the robot.

Parameters **`stop_type`** (`Stop`) – Whether to coast, brake or hold (*Default*: `Stop.COAST`).

CHAPTER EIGHT

SIGNALS AND UNITS

Many commands allow you to specify arguments in terms of well-known physical quantities. This page gives an overview of each quantity and its unit.

8.1 time: ms

All time and duration values are measured in milliseconds (ms).

For example, the duration of motion with `run_time`, the duration of `wait` or the time values returned by the `StopWatch` are specified in milliseconds.

8.2 angle: deg

All angles are measured in degrees (deg). One full rotation corresponds to 360 degrees.

For example, the angle values of a `Motor` and the `Gyro Sensor` are expressed in degrees.

8.3 rotational speed: deg/s

Rotational speed or *angular velocity* describes how fast something rotates, expressed as the number of degrees per second (deg/s).

For example, the rotational speed values of a `Motor` and the `Gyro Sensor` are expressed in degrees per second.

While we recommend working with degrees per second in your programs, you can use the following table to convert between commonly used units.

	deg/s	rpm
1 deg/s =	1	1/6=0.167
1 rpm =	6	1

8.4 distance: mm

Distances are expressed in millimetres (mm) whenever possible.

For example, the distance value of the `Ultrasonic Sensor` is measured in millimetres.

While we recommend working with millimetres in your programs, you can use the following table to convert between commonly used units.

	mm	cm	inch
1 mm =	1	0.1	0.0394
1 cm =	10	1	0.394
1 inch =	25.4	2.54	1

8.5 dimension: mm

Dimensions are expressed in millimetres (mm) whenever possible, just like distances.

For example, the diameter of a wheel is measured in millimetres.

8.6 relative distance: %

Some distance measurements do not provide an accurate value with a specific unit but instead range from very close (0%) to very far (100%). These are referred to as *relative distances*.

For example, the distance value of the `Infrared Sensor` is a relative distance.

8.7 speed: mm/s

Linear speeds are expressed as millimetres per second (mm/s).

For example, the speed of a robotic vehicle is expressed in mm/s.

8.8 rotational acceleration: deg/s/s

Rotational acceleration or *angular acceleration* describes how quickly the rotational speed changes. This is expressed as the change of the number of degrees per second, during one second (deg/s/s). This is also commonly written as deg/s^2.

For example, you can adjust the rotational acceleration setting of a `Motor` to change how smoothly or how quickly it reaches the constant speed set point.

8.9 percentage: %

Some signals do not have specific units but instead range from a minimum (0%) to a maximum (100%). A specific type of percentages is *relative distances*.

For example, the sound *volume* ranges from 0% to 100%.

8.10 frequency: Hz

Sound frequencies are expressed in Hertz (Hz).

For example, you can choose the frequency of a *beep* to change the pitch.

8.11 voltage: mV

Voltages are expressed in millivolt (mV).

For example, you can check the voltage of the *battery*.

8.12 current: mA

Electrical currents are expressed in milliampere (mA).

For example, you can check the current that is supplied by the *battery*.

CHAPTER
NINE

ROBOT EDUCATOR

This example makes the Robot Educator (Figure 9.1) drive until it ‘sees’ an obstacle. Then it backs up, turns around and starts driving again.

You can find building instructions for the Robot Educator on the LEGO® Education website.

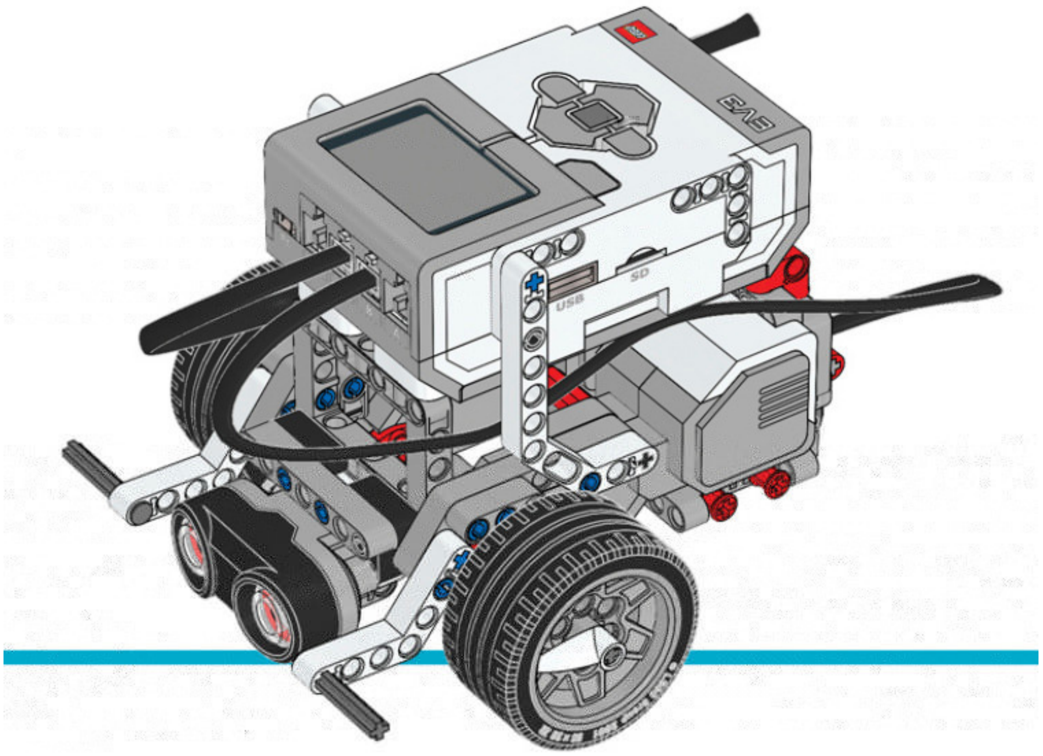


Figure 9.1: Robot Educator with the Ultrasonic Sensor.

```
#!/usr/bin/env pybricks-micropython

from pybricks import ev3brick as brick
from pybricks.ev3devices import Motor, TouchSensor, ColorSensor
from pybricks.parameters import Port, Button, Color, ImageFile, SoundFile
from pybricks.tools import wait
from pybricks.robotics import DriveBase

# Play a sound.
brick.sound.beep()

# Initialize the Ultrasonic Sensor. It is used to detect
# obstacles as the robot drives around.
obstacle_sensor = UltrasonicSensor(Port.S4)

# Initialize two motors with default settings on Port B and Port C.
# These will be the left and right motors of the drive base.
left_motor = Motor(Port.B)
right_motor = Motor(Port.C)

# The wheel diameter of the Robot Educator is 56 millimeters.
wheel_diameter = 56

# The axle track is the distance between the centers of each of the wheels.
# For the Robot Educator this is 114 millimeters.
axle_track = 114

# The DriveBase is composed of two motors, with a wheel on each motor.
# The wheel_diameter and axle_track values are used to make the motors
# move at the correct speed when you give a motor command.
robot = DriveBase(left_motor, right_motor, wheel_diameter, axle_track)

# The following loop makes the robot drive forward until it detects an
# obstacle. Then it backs up and turns around. It keeps on doing this
# until you stop the program.
while True:
    # Begin driving forward at 200 millimeters per second.
    robot.drive(200, 0)

    # Wait until an obstacle is detected. This is done by repeatedly
    # doing nothing(waiting for 10 milliseconds) while the measured
    # distance is still greater than 300 mm.
    while obstacle_sensor.distance() > 300:
        wait(10)

    # Drive backward at 100 millimeters per second. Keep going for 2 seconds.
    robot.drive_time(-100, 0, 2000)

    # Turn around at 60 degrees per second, around the midpoint between
    # the wheels. Keep going for 2 seconds.
    robot.drive_time(0, 60, 2000)
```

CHAPTER

TEN

COLOUR SORTER

This example program for the colour sorter (Figure 10.1) allows you to use the Colour Sensor to scan coloured Technic beams.

Scan the coloured beams one by one and add them to the tray. A beep confirms that the colour has been registered. When the tray is full or when you press the Centre Button, the robot will start distributing the Technic bricks by colour.

You can find building instructions for the colour sorter on the LEGO® Education website.

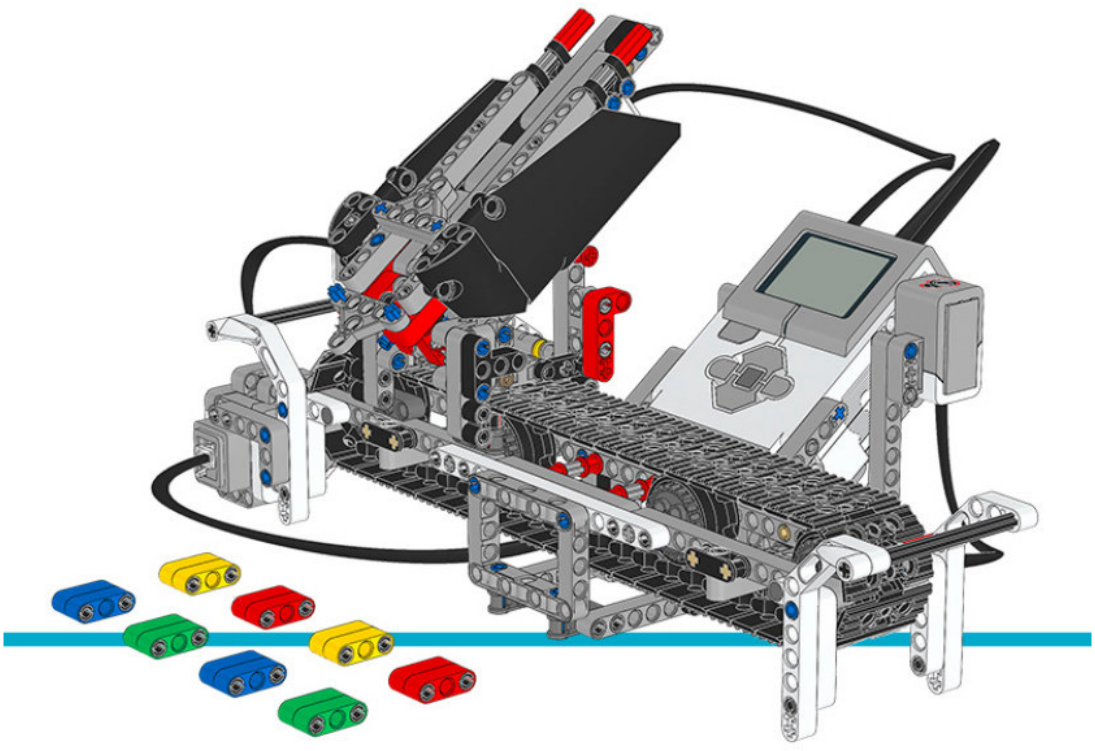


Figure 10.1: Colour Sorter

```
#!/usr/bin/env pybricks-micropython

from pybricks import ev3brick as brick
from pybricks.ev3devices import Motor, TouchSensor, ColorSensor
from pybricks.parameters import Port, Button, Color, ImageFile, SoundFile
from pybricks.tools import wait

# The colored objects are either red, green, blue, or yellow.
POSSIBLE_COLORS =(Color.RED, Color.GREEN, Color.BLUE, Color.YELLOW)

# Initialize the motors that drive the conveyor belt and eject the objects.
belt_motor = Motor(Port.D)
feed_motor = Motor(Port.A)

# Initialize the Touch Sensor. It is used to detect when the belt motor
# has moved the sorter module all the way to the left.
touch_sensor = TouchSensor(Port.S1)

# Initialize the Color Sensor. It is used to detect the color of the objects.
color_sensor = ColorSensor(Port.S3)

# This is the main loop. It waits for you to scan and insert 8 colored objects.
# Then it sorts them by color. Then the process starts over and you can scan
# and insert the next set of colored objects.
while True:
    # Get the feed motor in the correct starting position.
    # This is done by running the motor forward until it stalls. This
    # means that it cannot move any further. From this end point, the motor
    # rotates backward by 180 degrees. Then it is in the starting position.
    feed_motor.run_until_stalled(120)
    feed_motor.run_angle(450, -180)

    # Get the conveyor belt motor in the correct starting position.
    # This is done by first running the belt motor backward until the
    # touch sensor becomes pressed. Then the motor stops, and the the angle is
    # reset to zero. This means that when it rotates backward to zero later
    # on, it returns to this starting position.
    belt_motor.run(-500)
    while not touch_sensor.pressed():
        pass
    belt_motor.stop()
    wait(1000)
    belt_motor.reset_angle(0)

    # Clear all the contents from the display.
    brick.display.clear()

    # When we scan the objects, we store all the color numbers in a list.
    # We start with an empty list. It will grow as we add colors to it.
    color_list = []

    # This loop scans the colors of the objects. It repeats until 8 objects
    # are scanned and placed in the chute. This is done by repeating the loop
    # while the length of the list is still less than 8.
    while len(color_list) < 8:
        # Show an arrow that points to the color sensor.
        brick.display.image(ImageFile.RIGHT)
```

(continues on the next page)

(continued from previous page)

```
    # Show how many colored objects we have already scanned.
    brick.display.text(len(color_list))

    # Wait for the center button to be pressed or a color to be scanned.
    while True:
        # Store True if the center button is pressed or False if not.
        pressed = Button.CENTER in brick.buttons()
        # Store the color measured by the Color Sensor.
        color = color_sensor.color()
        # If the center button is pressed or a color is detected,
        # break out of the loop.
        if pressed or color in POSSIBLE_COLORS:
            break

    if pressed:
        # If the button was pressed, end the loop early.
        # We will no longer wait for any remaining objects
        # to be scanned and added to the chute.
        break
    else:
        # Otherwise, a color was scanned.
        # So we add(append) it to the list.
        brick.sound.beep(1000, 100, 100)
        color_list.append(color)

        # We don’t want to register the same color once more if we’re
        # still looking at the same object. So before we continue, we
        # wait until the sensor no longer sees the object.
        while color_sensor.color() in POSSIBLE_COLORS:
            pass
        brick.sound.beep(2000, 100, 100)

        # Show an arrow pointing to the center button,
        # to ask if we are done.
        brick.display.image(ImageFile.BACKWARD)
        wait(2000)

# Play a sound and show an image to indicate that we are done scanning.
brick.sound.file(SoundFile.READY)
brick.display.image(ImageFile.EV3)

# Now sort the bricks according the list of colors that we stored.
# We do this by going over each color in the list in a loop.
for color in color_list:

    # Wait for one second between each sorting action.
    wait(1000)

    # Run the conveyor belt motor to the right position based on the color.
    if color == Color.BLUE:
        brick.sound.file(SoundFile.BLUE)
        belt_motor.run_target(500, 10)
    elif color == Color.GREEN:
    brick.sound.file(SoundFile.GREEN)
        belt_motor.run_target(500, 132)
```

(continues on the next page)

(continued from previous page)

```
        elif color == Color.YELLOW:
             brick.sound.file(SoundFile.YELLOW)
             belt_motor.run_target(500, 360)
        elif color == Color.RED:
             brick.sound.file(SoundFile.RED)
             belt_motor.run_target(500, 530)

        # Now that the conveyor belt is in the correct position,
        # eject the colored object.
        feed_motor.run_angle(1500, 90)
        feed_motor.run_angle(1500, -90)
```

CHAPTER
ELEVEN

ROBOT ARM H25

This example program makes the robot arm (Figure 11.1) move the black wheel hub stacks around forever. The robot arm will first initialise and then start moving the hubs around.

You can find building instructions for the robot arm on the LEGO® Education website.

Tip: When you're building the robot, reverse the orientation of the EV3 Brick so that the micro SD card is easily accessible.

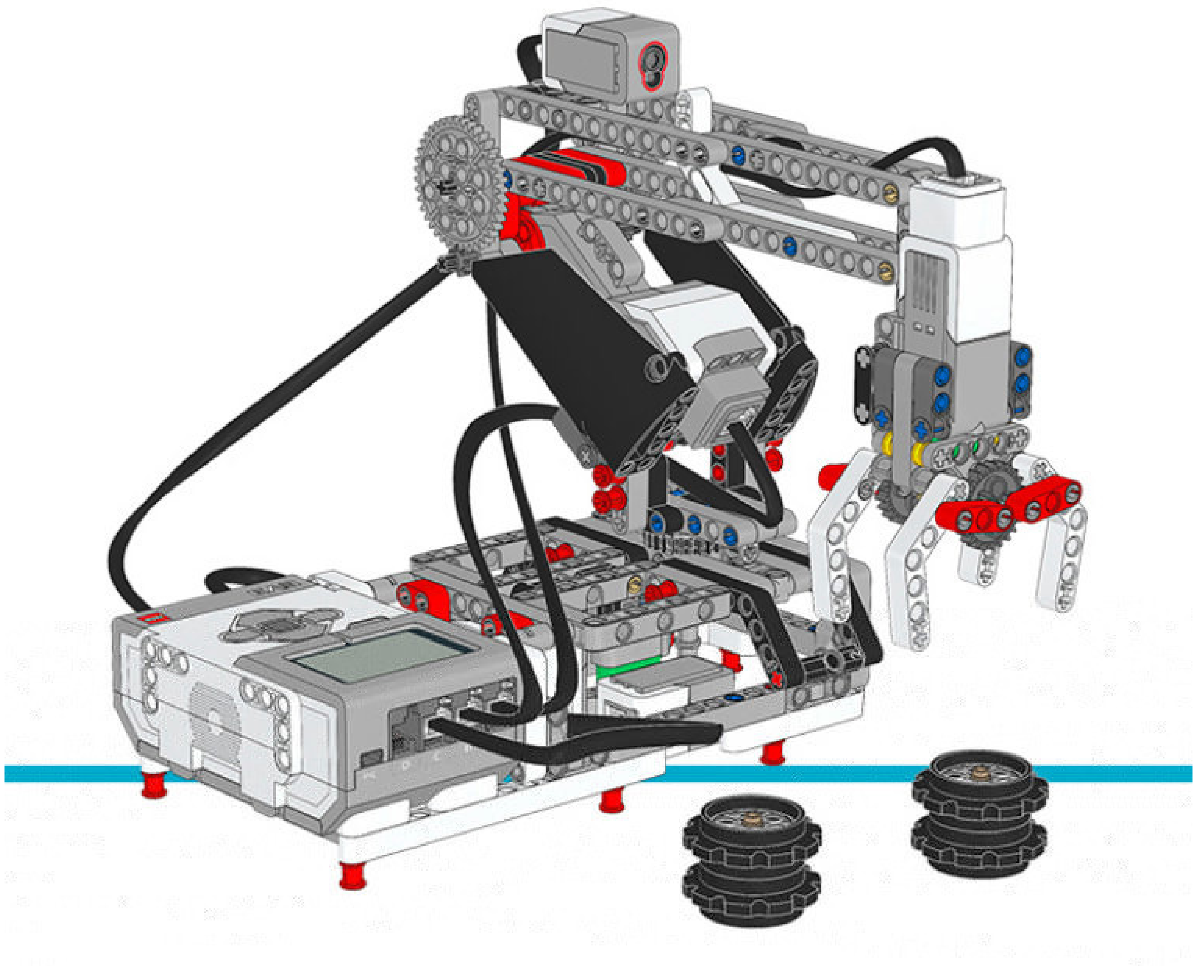


Figure 11.1: Robot Arm H25

(continued from previous page)

```
#!/usr/bin/env pybricks-micropython

from pybricks import ev3brick as brick
from pybricks.ev3devices import Motor, TouchSensor, ColorSensor
from pybricks.parameters import Port, Stop, Direction
from pybricks.tools import wait

# Configure the gripper motor on Port A with default settings.
gripper_motor = Motor(Port.A)

# Configure the elbow motor. It has an 8-teeth and a 40-teeth gear
# connected to it. We would like positive speed values to make the
# arm go upward. This corresponds to counterclockwise rotation
# of the motor.
elbow_motor = Motor(Port.B, Direction.COUNTERCLOCKWISE, [8, 40])

# Configure the motor that rotates the base. It has a 12-teeth and a
# 36-teeth gear connected to it. We would like positive speed values
# to make the arm go away from the Touch Sensor. This corresponds
# to counterclockwise rotation of the motor.
base_motor = Motor(Port.C, Direction.COUNTERCLOCKWISE, [12, 36])

# Limit the elbow and base accelerations. This results in
# very smooth motion. Like an industrial robot.
elbow_motor.set_run_settings(60, 120)
base_motor.set_run_settings(60, 120)

# Set up the Touch Sensor. It acts as an end-switch in the base
# of the robot arm. It defines the starting point of the base.
base_switch = TouchSensor(Port.S1)

# Set up the Color Sensor. This sensor detects when the elbow
# is in the starting position. This is when the sensor sees the
# white beam up close.
elbow_sensor = ColorSensor(Port.S3)

# Initialize the elbow. First make it go down for one second.
# Then make it go upwards slowly(15 degrees per second) until
# the Color Sensor detects the white beam. Then reset the motor
# angle to make this the zero point. Finally, hold the motor
# in place so it does not move.
elbow_motor.run_time(-30, 1000)
elbow_motor.run(15)
while elbow_sensor.reflection() < 32:
   wait(10)
elbow_motor.reset_angle(0)
elbow_motor.stop(Stop.HOLD)

# Initialize the base. First rotate it until the Touch Sensor
# in the base is pressed. Reset the motor angle to make this
# the zero point. Then hold the motor in place so it does not move.
base_motor.run(-60)
while not base_switch.pressed():
   wait(10)
base_motor.reset_angle(0)
base_motor.stop(Stop.HOLD)
```

(continues on the next page)

(continued from previous page)

```
# Initialize the gripper. First rotate the motor until it stalls.
# Stalling means that it cannot move any further. This position
# corresponds to the closed position. Then rotate the motor
# by 90 degrees such that the gripper is open.
gripper_motor.run_until_stalled(200, Stop.COAST, 50)
gripper_motor.reset_angle(0)
gripper_motor.run_target(200, -90)

def robot_pick(position):
    # This function makes the robot base rotate to the indicated
    # position. There it lowers the elbow, closes the gripper, and
    # raises the elbow to pick up the object.

    # Rotate to the pick-up position.
    base_motor.run_target(60, position, Stop.HOLD)
    # Lower the arm.
    elbow_motor.run_target(60, -40)
    # Close the gripper to grab the wheel stack.
    gripper_motor.run_until_stalled(200, Stop.HOLD, 50)
    # Raise the arm to lift the wheel stack.
    elbow_motor.run_target(60, 0, Stop.HOLD)

def robot_release(position):
    # This function makes the robot base rotate to the indicated
    # position. There it lowers the elbow, opens the gripper to
    # release the object. Then it raises its arm again.

    # Rotate to the drop-off position.
    base_motor.run_target(60, position, Stop.HOLD)
    # Lower the arm to put the wheel stack on the ground.
    elbow_motor.run_target(60, -40)
    # Open the gripper to release the wheel stack.
    gripper_motor.run_target(200, -90)
    # Raise the arm.
    elbow_motor.run_target(60, 0, Stop.HOLD)

# Play three beeps to indicate that the initialization is complete.
brick.sound.beeps(3)

# Define the three destinations for picking up and moving the wheel stacks.
LEFT = 160
MIDDLE = 100
RIGHT = 40

# This is the main part of the program. It is a loop that repeats endlessly.
#
# First, the robot moves the object on the left towards the middle.
# Second, the robot moves the object on the right towards the left.
# Finally, the robot moves the object that is now in the middle, to the right.
#
# Now we have a wheel stack on the left and on the right as before, but they
# have switched places. Then the loop repeats to do this over and over.
while True:
```

(continues on the next page)

(continued from previous page)

```
    # Move a wheel stack from the left to the middle.
    robot_pick(LEFT)
    robot_release(MIDDLE)

    # Move a wheel stack from the right to the left.
    robot_pick(RIGHT)
    robot_release(LEFT)

    # Move a wheel stack from the middle to the right.
    robot_pick(MIDDLE)
    robot_release(RIGHT)
```

PYTHON MODULE INDEX

INDEX