# Overview

This tutorial steps through the process of building a Python-based framework for a First Lego League robot. Ultimately, the tutorial builds up to a framework that support all of the following features however the concepts are built on gradually allowing simpler solutions from earlier steps in the process to be used if they suit your requirements.

* allow multiple actions to be run in parallel, serial or any combination of the two.
* allow the actions of a given run to be specified in an external ‘program’ file. The ‘program’ file will support the parallel and serial constructs mentioned above.
* allow the robot to automatically detect which attachment has been fitted and run the correct program. Attachment recognition will be performed with a dedicated colour sensor which will read different coloured bricks mounted on the attachment itself.
* allow the execution of a program to instantly be interrupted and restarted in the event that the robot is lifted from the mat and returned to base.
* implement a flexible logging system that allows the level of logging in different sections of the code to be controlled granularly.

The framework will allow teams to develop different actions using a standard design that supports them being interrupted and restarted. This tutorial does not attempt to address the various actions that a robot will need to implement for competition – that is left open for teams to develop.

## Pre-requisites.

This tutorial assumes you have been exposed to some basic Python code and have followed the setup and worked through the example code published on the <http://www.ev3python.com> site.

Some of the basic python concepts assumed include:

* Syntax <https://www.w3schools.com/python/python_syntax.asp>
* Variables <https://www.w3schools.com/python/python_variables.asp>
* If .. Else <https://www.w3schools.com/python/python_conditions.asp>
* For Loops <https://www.w3schools.com/python/python_for_loops.asp>
* While Loops <https://www.w3schools.com/python/python_while_loops.asp>
* Sets <https://www.w3schools.com/python/python_sets.asp>
* Dictionaries <https://www.w3schools.com/python/python_dictionaries.asp>
* Arrays <https://www.w3schools.com/python/python_arrays.asp>

# Running Actions in Parallel

In a First Lego League competition, you must complete as many challenges as you can in two and a half minutes. To maximise this time, you will often have to run actions in parallel. Some actions, like driving forward for five seconds, have a known duration whereas others a variable, such as driving to a black line or driving until the ultrasonic sensor signals a stop.

The following examples use a very simple challenge to demonstrate different techniques.

* have the two large motors run in parallel for a period of x and y seconds respectively.
* once both have stopped, then run the medium motor for a period of z seconds.

Consider the following code:

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# run these in parallel

largeMotor\_Left.on\_for\_seconds(speed=50, seconds=2, brake=True)

largeMotor\_Right.on\_for\_seconds(speed=50, seconds=4, brake=True)

# run this after the previous have completed

mediumMotor.on\_for\_seconds(speed = 10, seconds=6)

Program1.py

What happens when you run it? As you will see, the three motors start one after the other. This is due to the default behaviour of the on\_for\_seconds() function which is to pause the program until the action is finished. This behaviour is known as blocking.

The default blocking behaviour can be overridden by using the block attribute as shown below. When starting the left-hand motor, the parameter block=False is added to the call which allows the right hand motor to start at the same time.

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# run these in parallel

largeMotor\_Left.on\_for\_seconds(speed=50, seconds=2, brake=True, block=False)

largeMotor\_Right.on\_for\_seconds(speed=50, seconds=4, brake=True, block=True)

# run this after the previous have completed

mediumMotor.on\_for\_seconds(speed = 10, seconds=6)

Program2.py

Likewise, we can add block=True to the right-hand motor to ensure it is finished before the medium motor starts. We can use the block parameter on the right-hand motor as **we know** it will finish after the left.

But what if we don’t know which will finish first? Imagine a more complex example where the motors are not running for a certain amount of time but are instead completing a different number of rotations at different speeds.

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# run these in parallel

largeMotor\_Left.on\_for\_rotations(speed = 30, rotations=4, brake=True, block=False)

largeMotor\_Right.on\_for\_rotations(speed = 40, rotations=3, brake=True, block=True)

# run this after the previous have completed

mediumMotor.on\_for\_seconds(speed = 10, seconds=6)

Program3.py

Which finishes first? Run the program and see what happens.

As you can see, the left-hand motor actually runs longer than the right-hand motor. However, the code incorrectly specifies the block=True on the right-hand motor. The result is that the once the right-hand motor completes its three revolutions, the left-hand motor and the medium motor are both running together for a period of time.

But can’t I use the wait\_until\_not\_moving() command?

Actually, yes you can! The code below does exactly what we originally asked for – both large motors turn the specified amount before the medium motor is turn on.

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# run these in parallel

largeMotor\_Left.on\_for\_rotations(speed = 30, rotations=4, brake=True, block=False)

largeMotor\_Right.on\_for\_rotations(speed = 40, rotations=3, brake=True, block=False)

largeMotor\_Left.wait\_until\_not\_moving()

largeMotor\_Right.wait\_until\_not\_moving()

# run this after the previous have completed

mediumMotor.on\_for\_seconds(speed = 10, seconds=6)

Program4.py

So why don’t we just stick with this solution? Although it works in this simple example, the solution will not support the idea of stopping the current program as soon as the robot is lifted. The following code demonstrates this.

#!/usr/bin/env python3

from ev3dev2.sensor.lego import TouchSensor

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

ts = TouchSensor()

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# run these in parallel

largeMotor\_Left.on\_for\_rotations(speed = 30, rotations=4, brake=True, block=False)

largeMotor\_Right.on\_for\_rotations(speed = 40, rotations=3, brake=True, block=False)

# stop the rotations if the user lifts the robot (simulate by pressing the button)

if ts.is\_pressed:

largeMotor\_Left.off()

largeMotor\_Right.off()

largeMotor\_Left.wait\_until\_not\_moving()

largeMotor\_Right.wait\_until\_not\_moving()

# run this after the previous have completed

mediumMotor.on\_for\_seconds(speed = 10, seconds=6)

Program5.py

If you run the program and press the touch sensor as soon as the large motors start spinning, you will notice that the wheels **do not** stop. That’s because the touch sensor is tested **exactly once** to see if it has been pressed before the code continues to the wait\_until\_not\_moving() block. You can prove that the code works by re-running the code and holding the touch sensor down **before** the large motors start running. You will hear a click as they start and immediately stop before the program continues on to starting the medium motor.

A better solution: Threads

What are threads?

Lets revisit Program2.py

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# run these in parallel

largeMotor\_Left.on\_for\_seconds(speed = 50, seconds=2, brake=True, block=False)

largeMotor\_Right.on\_for\_seconds(speed = 50, seconds=4, brake=True, block=True)

# run this after the previous have completed

mediumMotor.on\_for\_seconds(speed = 10, seconds=6)

restructure it to look like this:

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

def onForSeconds(motor, speed, seconds):

motor.on\_for\_seconds(speed, seconds, brake = True, block = True)

def main():

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# run these in parallel

onForSeconds(motor = largeMotor\_Left, speed = 50, seconds = 2)

onForSeconds(motor = largeMotor\_Right, speed = 40, seconds = 3)

largeMotor\_Left.wait\_until\_not\_moving()

largeMotor\_Right.wait\_until\_not\_moving()

# run this after the previous have completed

onForSeconds(motor = mediumMotor, speed = 10, seconds = 6)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Program6.py

Add threads:

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

import threading

def onForSeconds(motor, speed, seconds):

motor.on\_for\_seconds(speed, seconds, brake = True, block = True)

def main():

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# create a threadPool array to 'collect' the threads ..

threadPool = []

thread1 = threading.Thread(target = onForSeconds, args = (largeMotor\_Left, 30, 4))

thread2 = threading.Thread(target = onForSeconds, args = (largeMotor\_Right, 40, 3))

threadPool.append(thread1)

threadPool.append(thread2)

# start threads

thread1.start()

thread2.start()

# are any threads still working?

while threadPool:

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

# All threads are complete, so we can run the next step ..

threadPool = []

thread3 = threading.Thread(target = onForSeconds, args = (mediumMotor, 10, 6))

threadPool.append(thread3)

# start the thread

thread3.start()

# are any threads still working?

while threadPool:

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Python7.py

We can smarten the code up a little ..

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

import threading

def waitUntilAllThreadsComplete(threadPool):

while threadPool:

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

def onForSeconds(motor, speed, seconds):

motor.on\_for\_seconds(speed, seconds, brake = True, block = True)

def main():

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# create a threadPool array to 'collect' the threads ..

threadPool = []

thread1 = threading.Thread(target = onForSeconds, args = (largeMotor\_Left, 30, 4))

thread2 = threading.Thread(target = onForSeconds, args = (largeMotor\_Right, 40, 3))

threadPool.append(thread1)

threadPool.append(thread2)

# start threads

thread1.start()

thread2.start()

# are any threads still working?

waitUntilAllThreadsComplete(threadPool)

# All threads are complete, so we can run the next step ..

threadPool = []

thread3 = threading.Thread(target = onForSeconds, args = (mediumMotor, 10, 6))

threadPool.append(thread3)

# start the thread

thread3.start()

# are any threads still working?

waitUntilAllThreadsComplete(threadPool)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Python8.py

So .. what happens when you have a really big program?  How large does this get???

Can we build a run list?

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

import threading

import types

def onForSeconds(motor, speed, seconds):

motor.on\_for\_seconds(speed, seconds, brake = True, block = True)

def createAction(name, motor, speed, seconds):

action = types.SimpleNamespace()

action.name = name

action.motor = motor

action.speed = speed

action.seconds = seconds

return action

def main():

actions = []

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

action1 = createAction("onForSeconds", largeMotor\_Left, 20, 4)

action2 = createAction("onForSeconds", largeMotor\_Right, 40, 3)

action3 = createAction("onForSeconds", mediumMotor, 10, 8)

actions.append(action1)

actions.append(action2)

actions.append(action3)

for action in actions:

if action.name == "onForSeconds":

onForSeconds(action.motor, action.speed, action.seconds)

if \_\_name\_\_ == '\_\_main\_\_':

main()  
 Python9.py

But wait!  All of the actions are executed one after the other.  We need to be able to specify those that run in parallel.

Arrays of arrays ..

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

import types

def onForSeconds(motor, speed, seconds):

motor.on\_for\_seconds(speed, seconds, brake = True, block = True)

def createAction(name, motor, speed, seconds):

action = types.SimpleNamespace()

action.name = name

action.motor = motor

action.speed = speed

action.seconds = seconds

return action

def main():

actions = []

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

action1 = createAction("onForSeconds", largeMotor\_Left, 20, 4)

action2 = createAction("onForSeconds", largeMotor\_Right, 40, 3)

action3 = createAction("onForSeconds", mediumMotor, 10, 8)

actionParallel = []

actionParallel.append(action1)

actionParallel.append(action2)

actions.append(actionParallel)

actions.append(action3)

for action in actions:

# are their multiple actions to execute in parallel?

if isinstance(action, list):

for subAction in action:

if subAction.name == "onForSeconds":

onForSeconds(subAction.motor, subAction.speed, subAction.seconds)

# is there a single action to execute?

else:

if action.name == "onForSeconds":

onForSeconds(action.motor, action.speed, action.seconds)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Python10.py

Now let’s put that thread stuff back in.

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

import types

import threading

def waitUntilAllThreadsComplete(threadPool):

while threadPool:

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

def onForSeconds(motor, speed, seconds):

motor.on\_for\_seconds(speed, seconds, brake = True, block = True)

def createAction(name, motor, speed, seconds):

action = types.SimpleNamespace()

action.name = name

action.motor = motor

action.speed = speed

action.seconds = seconds

return action

def main():

threadPool = []

actions = []

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

action1 = createAction("onForSeconds", largeMotor\_Left, 20, 4)

action2 = createAction("onForSeconds", largeMotor\_Right, 40, 3)

action3 = createAction("onForSeconds", mediumMotor, 10, 8)

actionParallel = []

actionParallel.append(action1)

actionParallel.append(action2)

actions.append(actionParallel)

actions.append(action3)

for action in actions:

# are their multiple actions to execute in parallel?

if isinstance(action, list):

for subAction in action:

if subAction.name == "onForSeconds":

thread = threading.Thread(target = onForSeconds, args = (subAction.motor, subAction.speed, subAction.seconds))

threadPool.append(thread)

thread.start()

# is there a single action to execute?

else:

if action.name == "onForSeconds":

thread = threading.Thread(target = onForSeconds, args = (action.motor, action.speed, action.seconds))

threadPool.append(thread)

thread.start()

waitUntilAllThreadsComplete(threadPool)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Python11.py

This approach allows us to add extra functions in easy.

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

from time import sleep

import threading

import types

def waitUntilAllThreadsComplete(threadPool):

while threadPool:

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

def onForSeconds(motor, speed, seconds):

motor.on\_for\_seconds(speed, seconds, brake = True, block = True)

def delayForSeconds(seconds):

sleep(seconds)

def createAction(name, motor, speed, seconds):

action = types.SimpleNamespace()

action.name = name

action.motor = motor

action.speed = speed

action.seconds = seconds

return action

def launchStep(action):

if action.name == "onForSeconds":

thread = threading.Thread(target = onForSeconds, args = (action.motor, action.speed, action.seconds))

thread.start()

return thread

if action.name == "delayForSeconds":

thread = threading.Thread(target = delayForSeconds, args = (action.seconds, ))

thread.start()

return thread

def main():

threadPool = []

actions = []

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

action1 = createAction("onForSeconds", largeMotor\_Left, 20, 4)

action2 = createAction("onForSeconds", largeMotor\_Right, 40, 3)

action3 = createAction("delayForSeconds", None, None, 2)

action4 = createAction("onForSeconds", mediumMotor, 10, 8)

actionParallel = []

actionParallel.append(action1)

actionParallel.append(action2)

actions.append(actionParallel)

actions.append(action3)

actions.append(action4)

for action in actions:

# are their multiple actions to execute in parallel?

if isinstance(action, list):

for subAction in action:

thread = launchStep(subAction)

threadPool.append(thread)

# is there a single action to execute?

else:

thread = launchStep(action)

threadPool.append(thread)

waitUntilAllThreadsComplete(threadPool)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Python12.py

# Stopping the Threads

Now that our framework can support running actions in parallel using threads, we can move on to implementing code to stop the actions when the touch sensor is pressed. Later we will revisit this and swap the touch sensor for a colour sensor that is monitoring the table surface.

As mentioned earlier, threads are designed to run on their own allowing the main program to continue its own work. It doesn’t matter that a command like on\_for\_seconds() or sleep() block execution as they **only block on their own thread**.

## Stopping the Threads (Eventually)

Below is a slightly updated version of Python12.py which includes additional code to test whether the touch sensor has been pressed. If it has, a new variable named stopProcessing is set to True and this is used control whether to continue processing or to exit both the while and for loops.

def main():

threadPool = []

actions = []

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

action1 = createAction("onForSeconds", largeMotor\_Left, 20, 4)

action2 = createAction("onForSeconds", largeMotor\_Right, 40, 3)

action3 = createAction("delayForSeconds", None, None, 2)

action4 = createAction("onForSeconds", mediumMotor, 10, 8)

actionParallel = []

actionParallel.append(action1)

actionParallel.append(action2)

actions.append(actionParallel)

actions.append(action3)

actions.append(action4)

for action in actions:

# are their multiple actions to execute in parallel?

if isinstance(action, list):

for subAction in action:

thread = launchStep(subAction)

threadPool.append(thread)

# is there a single action to execute?

else:

thread = launchStep(action)

threadPool.append(thread)

while not stopProcessing:

# remove any completed threads from the pool

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

# if there are no threads running, exist the 'while' loop

# and start the next action from the list

if not threadPool:

break

# if the touch sensor is pressed then complete everything

if ts.is\_pressed:

stopProcessing = True

sleep(0.25)

# if the 'stopProcessing' flag has been set then break out of altogether

if stopProcessing:

break

if \_\_name\_\_ == '\_\_main\_\_':

main()

Python13.py

Run the program above. When the two large motors start, hold down the touch sensor. The program will not stop immediately but will stop after the first actions are complete. The issue is that motor.on\_for\_seconds() continues to run for the specified time regardless of whether the touch sensor has been pressed or not. It is a single, uninterruptible command within the EV3 environment.

## Stopping the Threads (Immediately)

adadasd

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

from ev3dev2.sensor.lego import TouchSensor

from time import sleep

import threading

import time

import types

def onForSeconds(stop, motor, speed, seconds):

start\_time = time.time()

motor.on(speed, brake = True, block = False)

while time.time() < start\_time + seconds:

# if we are stopping prematurely break out of loop

if stop():

break

motor.off()

def delayForSeconds(stop, seconds):

start\_time = time.time()

while time.time() < start\_time + seconds:

if stop():

break

def createAction(name, motor, speed, seconds):

action = types.SimpleNamespace()

action.name = name

action.motor = motor

action.speed = speed

action.seconds = seconds

return action

def launchStep(stop, action):

if action.name == "onForSeconds":

thread = threading.Thread(target = onForSeconds, args = (stop, action.motor, action.speed, action.seconds))

thread.start()

return thread

if action.name == "delayForSeconds":

thread = threading.Thread(target = delayForSeconds, args = (stop, action.seconds))

thread.start()

return thread

def main():

threadPool = []

actions = []

stopProcessing = False

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

ts = TouchSensor()

action1 = createAction("onForSeconds", largeMotor\_Left, 20, 4)

action2 = createAction("onForSeconds", largeMotor\_Right, 40, 3)

action3 = createAction("delayForSeconds", None, None, 2)

action4 = createAction("onForSeconds", mediumMotor, 10, 8)

actionParallel = []

actionParallel.append(action1)

actionParallel.append(action2)

actions.append(actionParallel)

actions.append(action3)

actions.append(action4)

for action in actions:

# are their multiple actions to execute in parallel?

if isinstance(action, list):

for subAction in action:

thread = launchStep(lambda:stopProcessing, subAction)

threadPool.append(thread)

# is there a single action to execute?

else:

thread = launchStep(lambda:stopProcessing, action)

threadPool.append(thread)

while not stopProcessing:

# remove any completed threads from the pool

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

# if there are no threads running, exist the 'while' loop

# and start the next action from the list

if not threadPool:

break

# if the touch sensor is pressed then complete everything

if ts.is\_pressed:

stopProcessing = True

sleep(0.25)

# if the 'stopProcessing' flag has been set then break out of the program altogether

if stopProcessing:

break

if \_\_name\_\_ == '\_\_main\_\_':

main()

Python14.py

The solution to this is to change our various ‘action’ functions to perform their work within a continuous loop thus allowing them to check whether or not they should continue on each iteration. This is illustrated simply in the change made to the function delayForSeconds().

Below is the original code. Like the motor.on\_for\_seconds() function, the sleep() command is also non-interruptible.

def delayForSeconds(seconds):

sleep(seconds)

Restructuring the code, as shown below, allows the code to be interrupted. When this thread is started, the initial time is retrieved from the operating system. The process then simply loops while the current time is less than the original start time plus the delay (passed in as the parameter seconds) is reached.

An additional parameter, stop, is used to signify that the process should be interrupted. If stop evaluates to True, the program breaks out of the while loop and the thread terminates at that point.

def delayForSeconds(stop, seconds):

start\_time = time.time()

while time.time() < start\_time + seconds:

if stop():

break

The onForSeconds() function has also been rewritten to allow it to be interrupted. As with the delayForSeconds() function, it records the start time before turning on the motor. This time, the motor is turned on without any conditions and is only turned off if the time limit is exceeded or the stop parameter is set to True thus forcing the loop to end prematurely.

def onForSeconds(stop, motor, speed, seconds):

start\_time = time.time()

motor.on(speed, brake = True, block = False)

while time.time() < start\_time + seconds:

# if we are stopping prematurely break out of loop

if stop():

break

motor.off()

Hang on, you might be thinking, the idea of the thread is that we start it and it runs along by itself. We may pass it some parameters when we kick it off but those don’t change as the thread runs so how does the program stop the processing using the stop parameter?

This is the trick to this whole process – the stop parameter used in each function is not a traditional parameter but is actually a function which is evaluated every time it is referenced. As such, it can change at any time and can be used to break out of the while loop in the delayForSeconds() or onForSeconds() function.

If you look in the main() function, you will see the following lines (separated by other code). The variable stopProcessing is a simple Boolean value that is set to True when the user presses the touch sensor.

You will also notice that when we launch the thread, we are using the keyword lambda: to indicate that this should be passed as a function rather than a value so that it can be evaluated and re-evaluated in the ‘action’ threads. You don’t really need to understand exactly how this works other than to know that a change in the stopProcessing variable in the main() function can now be evaluated in the ‘action’ threads at any time.

stopProcessing = False

…

while True:

thread = launchStep(lambda:stopProcessing, action)

if ts.is\_pressed:

stopProcessing = True

break

Finally, if you look again at the onForSeconds() function you will notice that when evaluating the stop variable that it is referenced with parenthesis – further hinting that the variable is actually a reference to a function and can be re-evaluated.

if stop():

break

More information regarding lambda expressions can be found here > https://docs.python.org/3.3/tutorial/controlflow.html?highlight=lambda#lambda-expressions

# Specifying our Robot Actions in a File

So now that we have the robot performing actions in parallel and stopping as soon as it is signalled to do so, it’s time to move our focus to reading files. This section starts with reading actions from a simple text file which is adequate for specifying actions that will be completed in sequence but is not flexible enough to specify parallel actions.

To address this, we will then switch out focus to XML which is a mark-up language used to specify more complex data in an easily readable way. It can handle all the complexities of our requirements plus more!

## Reading from Text Files

So far, our code forces us to define the various tasks we want the robot to perform using the Action named-tuple and to put them into the array for processing. Although this is not too difficult to do, wouldn’t it be better if we could simply read the instructions from a file?

As Program15.py below shows, reading from a text file in Python is really simple. The open() command takes two parameters – the file name to be opened and the mode in which to access it. An ‘r’ indicates that we only want to read the file’s contents but other values such as ‘a’ (append to the bottom of the file), ‘w’ (write to the file) and ‘x’ (create a new file) are also valid.

The open() command returns a file ‘handle’ that we can then use to read lines and to ultimately close the file.

#!/usr/bin/env python3

from time import sleep

f = open("Program15\_data.txt", "r")

for aLineOfText in f:

print(aLineOfText)

f.close()

sleep(5)

Python15.py

Run the program on the EV3. You will see the EV3’s screen clear and print the contents of the text file on the screen in really small text. I have included a sleep() statement at the end of the program to prevent the program from completing for 5 seconds allowing you enough time to read the screen before the EV3 menu is shown again.

In Program16.py, the program has been expanded to read a line from the configuration file shown below and split the text into four separate values which we will later map to our Action named-tuple.

onForSeconds,largeMotor\_Left,20,4

onForSeconds,largeMotor\_Right,40,3

delayForSeconds,None,0,2

onForSeconds,mediumMotor,10,8

After reading the line of text from the tile, the program splits the text into individual tokens using the split(",") function. The result is an array of individual strings that we could iterate through or we can refer to using their index.

#!/usr/bin/env python3

from time import sleep

f = open("Program16\_data.txt", "r")

for aLineOfText in f:

tokens = aLineOfText .split(",")

# read the string values into local variables - make

# the speed and seconds floating point numbers

name = tokens[0]

motor = tokens[1]

speed = float(tokens[2])

seconds = float(tokens[3])

print( "name = {}, motor = {}, speed = {}, seconds = {}".format(name, motor, speed, seconds) )

f.close()

sleep(5)

Python16.py

The output from the program above is shown below:

name = onForSeconds, motor = largeMotor\_Left, speed = 20, seconds = 4

name = onForSeconds, motor = largeMotor\_Right, speed = 40, seconds = 3

name = delayForSeconds, motor = None, speed = 0, seconds = 2

name = onForSeconds, motor = mediumMotor, speed = 10, seconds = 8

Now that we can read and parse our commands from a text file, we can incorporate this into our original code. The affected sections of code are shown below.

def createAction(name, motor, speed, seconds):

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

action = types.SimpleNamespace()

action.name = name

action.speed = speed

action.seconds = seconds

if (motor == "largeMotor\_Left"):

action.motor = largeMotor\_Left

if (motor == "largeMotor\_Right"):

action.motor = largeMotor\_Right

if (motor == "mediumMotor"):

action.motor = mediumMotor

return action

def main():

threadPool = []

actions = []

stopProcessing = False

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

ts = TouchSensor()

f = open("Program16\_data.txt", "r")

for aLineOfText in f:

tokens = aLineOfText .split(",")

# read the string values into local variables - make

# the speed and seconds floating point numbers

name = tokens[0]

motor = tokens[1]

speed = float(tokens[2])

seconds = float(tokens[3])

action = createAction(name, motor, speed, seconds)

# launch the action

thread = launchStep(lambda:stopProcessing, action)

threadPool.append(thread)

while not stopProcessing:

# remove any completed threads from the pool

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

# if there are no threads running, exist the 'while' loop

# and start the next action from the list

if not threadPool:

break

# if the touch sensor is pressed then complete everything

if ts.is\_pressed:

stopProcessing = True

sleep(0.25)

# if the 'stopProcessing' flag has been set then break out of the program

if stopProcessing:

break

if \_\_name\_\_ == '\_\_main\_\_':

main()

Python17.py

It appears we have taken one step forward and two steps back. Although we are reading from a text file, we have lost the ability to specify that some actions be executed in parallel. We could look at changing our file structure to include an additional field to indicate which actions should be performed but there must be a better way!

## Recursion

But before we look at a solution to allow us to specify our actions a little better, we need to look at a concept called recursion. The program below demonstrates this perfectly:

#!/usr/bin/env python3

from time import sleep

def printNumber(n):

print("{}, ".format(n), end="")

if n < 10:

printNumber(n + 1)

print("{}, ".format(n), end="")

def main():

printNumber(0)

print("")

sleep(5)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Python18.py

When the program is run, the output below is seen on the EV3 screen.

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0

As you can see, the program has counted from 0 to 10 and then back down again - but how did it do it?

When the program first starts, it calls the printNumber() function with a starting value of zero. The printNumber() function prints the value to the screen and then **calls itself** **again** with the number it was passed plus one. The next call to printNumber() prints out the new value of one before calling itself again .. the process continues until the passed number value equals ten.

Once the recursion has completed, the function ends and prints out the value of ten and completes. Program controls passes back to the previous instance of printNumber() where it prints the parameter it was passed (a nine) before completing. Control passes back to the previous instance of printNumber()where it prints the parameter it was passed (an eight) before completing and so on until the all calls are completed and the program completes.

The nested calls to printNumber() can be visualised as shown below:

Main starts()

Call printNumber(0)

print(0)

Call printNumber(0 + 1)

print(1)

Call printNumber(1 + 1)

print(2)

Call printNumber(2 + 1)

print(3)

.. and so on until the parameter equals 10.

print(3)

end function

print(2)

end function

print(1)

end function

print(0)

end function

end program

As you will see in the next section, recursive calls can be a very useful tool.

## XML

As we discovered earlier, text files are easy to work with but are not good for specifying the relationships between lines. Other data formats, such as JSON and XML, do a much better job of this and are just as easy to work with.

Below is a sample XML file that models the same actions we have been using through this tutorial. As you can see, each step is described in <step> tags with the values of the action name, motor, speed and seconds listed as name=value pairs. Tags always come in pairs – in the sample below you will see opening and closing tags, such as <step> and </step>. Tags can contain other tags and this is how we have modelled the parallel actions – the steps to perform in parallel are wrapped in an outer tag. Visually, I have indented the parallel actions in the sample below to reinforce this relationship but in reality, XML processing ignores the whitespace between tags.

In XML parlance, the tags are called elements and the name value pairs are called attributes.

<steps>

<step action="launchInParallel">

<step action="onForSeconds" motor="largeMotor\_Left" speed="20" seconds="4"></step>

<step action="onForSeconds" motor="largeMotor\_Right" speed="40" seconds="3"></step>

</step>

<step action="delayForSeconds" motor="None" speed="0" seconds="2"></step>

<step action="onForSeconds" motor="mediumMotor" speed="10" seconds="8"></step>

</steps>

One little trick with XML that you will see everywhere is that an open and close tag with nothing between them can be simplified as shown below.

<steps>

<step action="launchInParallel">

<step action="onForSeconds" motor="largeMotor\_Left" speed="20" seconds="4" />

<step action="onForSeconds" motor="largeMotor\_Right" speed="40" seconds="3" />

</step>

<step action="delayForSeconds" motor="None" speed="0" seconds="2" />

<step action="onForSeconds" motor="mediumMotor" speed="10" seconds="8" />

</steps>

Python handles XML and JSON easily due to the rich libraries of functions that are available however most Python tutorials on the web show how to open and parse JSON rather than XML. I have chosen XML for this tutorial as it offers a number of features not found in JSON, including:

* Support for embedded comments. As your robot runs grow in complexity, you will probably need to add comments to the file to remind you what each step does. Surprisingly for a new standard, JSON does not offer this.
* XML can be validated to ensure that it is structurally and logically correct. I have not described the validation process in this tutorial but a quick search for XSD schema validation will reveal tutorials and even online validators.
* In my opinion, XML is easier to read and author than JSON.

Some sample code for reading an XML file is shown below. Note that we open the file and parse the contents into an object called xmlDocument and from there we retrieve the top-level node, called the root node, as our starting point for processing.

The individual elements are presented to the program as an array which can be looped through using the common for statement that we have used for other arrays.

#!/usr/bin/env python3

import xml.etree.ElementTree as ET

xmlDocument = ET.parse('Program18\_data.xml')

steps = xmlDocument.getroot()

for step in steps:

action = step.get('action')

print("action = {}".format(action))

sleep(5)

Python19.py

The output of the above program is shown below:

action = launchInParallel

action = delayForSeconds

action = onForSeconds

But why are there only three elements when our XML has more?

The answer is that when looping through the child elements of a parent element, only the first level children are returned. If you want to look at a child node’s children, you have to recursively process the document. I told you recursion would come in handy.

The program below will process all of the elements in the sample file. A simple test in the loopThroughXML() function determines if the current element is a launchInParallel tag and, if so, simply calls the loopThroughXML()function recursively to process the children.

#!/usr/bin/env python3

import xml.etree.ElementTree as ET

from time import sleep

def printAction(step):

action = step.get('action')

motor = step.get('motor')

speed = float(step.get('speed'))

seconds = float(step.get('seconds'))

print("action = {}".format(action), end="" )

print(", motor = {}".format(motor), end="" )

print(", speed = {}".format(speed), end="" )

print(", seconds = {}".format(seconds) )

def loopThroughXML(steps):

for step in steps:

action = step.get('action')

if action == 'launchInParallel':

loopThroughXML(step)

else:

printAction(step)

def main():

xmlDocument = ET.parse('Program18\_data.xml')

steps = xmlDocument.getroot()

loopThroughXML(steps)

sleep(5)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Python20.py

As you would expect, the output lists all elements in the input XML file.

action = onForSeconds, motor = largeMotor\_Left, speed = 20.0, seconds = 4.0

action = onForSeconds, motor = largeMotor\_Right, speed = 40.0, seconds = 3.0

action = delayForSeconds, motor = None, speed = 0.0, seconds = 2.0

action = onForSeconds, motor = mediumMotor, speed = 10.0, seconds = 8.0

Updating our code to use an XML file for input results in the code shown below. To reduce the size of the listing, I have removed the unchanged code.

def launchStep(stop, action):

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

name = action.get('action')

motor = action.get('motor')

speed = float(action.get('speed'))

seconds = float(action.get('seconds'))

if name == "onForSeconds":

if (motor == "largeMotor\_Left"):

motorToUse = largeMotor\_Left

if (motor == "largeMotor\_Right"):

motorToUse = largeMotor\_Right

if (motor == "mediumMotor"):

motorToUse = mediumMotor

thread = threading.Thread(target = onForSeconds, args = (stop, motorToUse, speed, seconds))

thread.start()

return thread

if name == "delayForSeconds":

thread = threading.Thread(target = delayForSeconds, args = (stop, seconds))

thread.start()

return thread

def main():

threadPool = []

actions = []

stopProcessing = False

ts = TouchSensor()

xmlDocument = ET.parse('Program21\_data.xml')

steps = xmlDocument.getroot()

for step in steps:

action = step.get('action')

# are their multiple actions to execute in parallel?

if action == 'launchInParallel':

for subSteps in step:

thread = launchStep(lambda:stopProcessing, subSteps)

threadPool.append(thread)

# is there a single action to execute?

else:

thread = launchStep(lambda:stopProcessing, step)

threadPool.append(thread)

while not stopProcessing:

# remove any completed threads from the pool

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

# if there are no threads running, exist the 'while' loop

# and start the next action from the list

if not threadPool:

break

# if the touch sensor is pressed then complete everything

if ts.is\_pressed:

stopProcessing = True

break

sleep(0.25)

# if the 'stopProcessing' flag has been set then break out altogether

if stopProcessing:

break

if \_\_name\_\_ == '\_\_main\_\_':

main()

Python21.py

The launchStep() function now accepts two parameters, our original stop flag to kill operation and a single XML element called action. The action name, motor, speed and seconds variables are populated with values retrieved from the XML element before launching a thread with the relevant action.

You will notice that the speed and seconds variable are converted into floating point numbers as they are extracted. XML files do not have a concept of a data type and all values read from them are string values. The conversion is necessary as attempting to assign a string value to the motor speed property, for example, will result in an error.

def launchStep(stop, action):

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

name = action.get('action')

motor = action.get('motor')

speed = float(action.get('speed'))

seconds = float(action.get('seconds'))

if name == "onForSeconds":

if (motor == "largeMotor\_Left"):

motorToUse = largeMotor\_Left

if (motor == "largeMotor\_Right"):

motorToUse = largeMotor\_Right

if (motor == "mm"):

motorToUse = mm

thread = threading.Thread(target = onForSeconds, args = (stop, motorToUse, speed, seconds))

thread.start()

return thread

if name == "delayForSeconds":

thread = threading.Thread(target = delayForSeconds, args = (stop, seconds))

thread.start()

return thread

The processing of the XML file is straight forward. Once opened and parsed, the program simply steps through the top-level elements and launches the appropriate actions using the launchStep() function. As before, each step is launched in its own thread and these are monitored for completion before the

If the current element is determined to be a launchInParallel step, then the program loops through its children and launches these simultaneously.

xmlDocument = ET.parse('Program21\_data.xml')

steps = xmlDocument.getroot()

for step in steps:

action = step.get('action')

# are their multiple actions to execute in parallel?

if action == 'launchInParallel':

for subSteps in step:

thread = launchStep(lambda:stopProcessing, subSteps)

threadPool.append(thread)

# is there a single action to execute?

else:

thread = launchStep(lambda:stopProcessing, step)

threadPool.append(thread)

while not stopProcessing:

...

Run the code and prove that the parallel steps do actually run together and that pressing the touch sensor will stop the process completely.

# What Next?

Although this version of the program is starting to look finished are still a number of things to be done to make it complete. Additional features could include:

* A launchInSeries element that would allow multiple actins to be launched one after the other. This could be useful when running multiple actions in parallel and one stream needs to perform a number of tasks in series. Consider the sample XML below:

<steps>

<step action="launchInParallel">

<step action="onForSeconds" motor="largeMotor\_Left" speed="20" seconds="4" />

<step action="launchInSeries">

<step action="delayForSeconds" motor="" speed="0” seconds="2" />

<step action="onForSeconds" motor="largeMotor\_Right" speed="40" seconds="3" />

</step>

</step>

<step action="onForSeconds" motor="mediumMotor" speed="10" seconds="8" />

</steps>

The launchInParallel step contains two actions, one of which is a launchInSeries step which, in turn, contains two child actions. When executing, the left-hand motor will start followed by the second motor one second later. The program will then wait until both of these processes re finished before activating the medium motor.

* Recursive processing of the XML file. As it stands, the current implementation will not support a launchInParallel step within another launchInParallel step (or launchInSeries steps either for that matter).

My final framework – named main.py – incorporates these features and more. Please review these to see how they were implemented.

# Launching Multiple Programs

One of our design goals was to allow the robot to have multiple programs available for each run we make on the table. To reduce the risk of the robot operators selecting the wrong program to run, the robot should use the colour sensor to detect a coloured patch on the fitted attachment and run the correct program for that attachment. Furthermore, if the robot is lifted during that run it should return to the start of the same program and wait for the operator to press a button to launch the program again.

Using XML again, we can define a list of programs that our robot can run as shown below. For each program, we can list the program name, the filename of the XML program itself and the red / green / blue components that the colour sensor must match before starting the program.

<programs>

<program name="Run1" fileName="Program22\_program\_1.xml" r="100" g="80" b="58" />

<program name="Run2" fileName="Program22\_program\_2.xml" r="200" g="80" b="58" />

</programs>

Program22\_programs.xml

The program below shows how this configuration file is used. After opening the file, the program enters an endless loop. At the start of the loop, the value of the colour sensor is read into a variable named rgb. The value is presented as an array of three numbers representing the red, green and blue components of the detected colour.

The program then loops through the <program> elements of the XML configuration file. For each program specified, the program retrieves the red, green and blue components and compares these to the values retrieved from the colour sensor. If the colours match, the program is started.

#!/usr/bin/env python3

import xml.etree.ElementTree as ET

from ev3dev2.sensor.lego import ColorSensor

from sys import stderr

def main():

cl = ColorSensor()

# Load programs ..

programsXML = ET.parse('Program22\_programs.xml')

programs = programsXML.getroot()

while True:

rgb = cl.raw

for program in programs:

programName = program.get('name')

rProgram = int(program.get('r'))

gProgram = int(program.get('g'))

bProgram = int(program.get('b'))

rColourSensor = rgb[0]

gColourSensor = rgb[1]

bColourSensor = rgb[2]

print('Colour sensor {} compared to {} ({}, {}, {}) result ({}, {}, {})'.format(rgb, programName, rProgram, gProgram, bProgram, rColourSensor - rProgram, gColourSensor - gProgram, bColourSensor - bProgram), file = stderr)

if abs(rColourSensor - rProgram) < 20 and abs(gColourSensor - gProgram) < 20 and abs(bColourSensor - bProgram) < 20:

print('Run program {}'.format(program.get('fileName')), file=stderr)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Program22.py

The line of code repeated below probably deserves an explanation. When reading a value from the colour sensor, the result may be affected by the surrounding light and reflections. To overcome this, the comparison for each component needs to be ‘close’ but not exact.

if abs(rColourSensor - rProgram) < 20 and abs(gColourSensor - gProgram) < 20 and abs(bColourSensor - bProgram) < 20:

The abs() function returns the absolute (or positive) value of a number. The code print(abs(-3)) will result in the number 3 (positive) being printed. The code above essentially says ‘if the red colour sensor value and red value retrieved from the program definition in the XML file are + / - 20 of each other then they match’. How does this work?

Let’s assume the red value retrieved from the colour sensor is 104 and the XML program definition defines a red value of 100. Doing the maths:

if abs(rColourSensor - rProgram) < 20 ... becomes

if abs(104 – 100) < 20 ... becomes

if abs(4) < 20 ... becomes

if 4 < 20 ... is true so the two values ‘close’

As you have probably guessed, the abs() function takes care of comparisons where the first value is smaller than the second value. For example, assume the red value retrieved from the colour sensor is 98 and the XML program definition defines a red value of 100.

if abs(rColourSensor - rProgram) < 20 ... becomes

if abs(98 – 100) < 20 ... becomes

if abs(-2) < 20 ... the abs(-2) equals 2, so

if 2 < 20 ... is true so the two values ‘close’

But wouldn’t the logic have worked anyway without the abs() command? After all, -2 is less than 20. No. What if the red values read from the colour sensor was 72? The equation (72 - 100) equals -28 which is definitely less than 20 so the result would be true. Logically though, 72 is not within 100 + / - 20.

Running the program delivers the following results:

Colour sensor (210, 72, 57) compared to Run1 (100, 80, 58) = (110, -8, -1)

Colour sensor (207, 73, 57) compared to Run2 (200, 80, 58) = (7, -7, -1)

Colour sensor (210, 72, 57) compared to Run1 (100, 80, 58) = (110, -8, -1)

Colour sensor (207, 73, 57) compared to Run2 (200, 80, 58) = (7, -7, -1)

...

If you are then to place a red brick over the colour sensor, the program will detect the colours and (hopefully) run the Run2 program. Of course, the values I detected for a red brick may not work in your environment so you may need to tweak the values in the XML configuration program.

But won’t this be a problem when you are competing – how do you know what values to enter? To avoid any environmental variations, the design of your robot and accessories should be such that the red brick is placed directly in front of the sensor and almost touching it. This will prevent stray light affecting the reading.

Colour sensor (210, 72, 57) compared to Run1 (100, 80, 58) = (110, -8, -1)

Colour sensor (207, 73, 57) compared to Run2 (200, 80, 58) = (7, -7, -1)

Colour sensor (210, 72, 57) compared to Run1 (100, 80, 58) = (110, -8, -1)

Colour sensor (207, 73, 57) compared to Run2 (200, 80, 58) = (7, -7, -1)

Run program Program22\_program\_2.xml

Now that we can detect the accessory by colour, all we need to do is load the corresponding program. This is shown in the program below:

def main():

cl = ColorSensor()

# Load programs ..

programsXML = ET.parse('Program23\_programs.xml')

programs = programsXML.getroot()

while True:

rgb = cl.raw

for program in programs:

programName = program.get('name')

rProgram = int(program.get('r'))

gProgram = int(program.get('g'))

bProgram = int(program.get('b'))

rColourSensor = rgb[0]

gColourSensor = rgb[1]

bColourSensor = rgb[2]

if abs(rColourSensor - rProgram) < 20 and abs(gColourSensor - gProgram) < 20 and abs(bColourSensor - bProgram) < 20:

fileName = program.get('fileName')

# Load program into memory ..

dataXML = ET.parse(fileName)

steps = dataXML.getroot()

for step in steps:

print(step.get('name'))

if \_\_name\_\_ == '\_\_main\_\_':

main()

Python23.py

The program above extends on Python22.py by opening the XML program after the colour is detected. It uses the same XML logic as that for reading the program definitions and then simply prints out the elements to the screen.

Run the program and place a red brick in front of the colour sensor. The result should be:

runInParallel

delayForSeconds

onForSeconds

Incorporating the XML logic to read both the list of programs and the selected program from files back into our main robot program results in the following program. To reduce space, I only the updated main() function is shown. The original logic to launch a step and the individual step logic has identical to Program21.py.

def main():

threadPool = []

actions = []

stopProcessing = False

ts = TouchSensor()

cl = ColorSensor()

# Load programs ..

programsXML = ET.parse('Program24\_programs.xml')

programs = programsXML.getroot()

while True:

rgb = cl.raw

for program in programs:

programName = program.get('name')

rProgram = int(program.get('r'))

gProgram = int(program.get('g'))

bProgram = int(program.get('b'))

rColourSensor = rgb[0]

gColourSensor = rgb[1]

bColourSensor = rgb[2]

if abs(rColourSensor - rProgram) < 20 and abs(gColourSensor - gProgram) < 20 and abs(bColourSensor - bProgram) < 20:

fileName = program.get('fileName')

# Load program into memory ..

dataXML = ET.parse(fileName)

steps = dataXML.getroot()

for step in steps:

action = step.get('action')

# are their multiple actions to execute in parallel?

if action == 'launchInParallel':

for subSteps in step:

thread = launchStep(lambda:stopProcessing, subSteps)

threadPool.append(thread)

# is there a single action to execute?

else:

thread = launchStep(lambda:stopProcessing, step)

threadPool.append(thread)

while not stopProcessing:

# remove any completed threads from the pool

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

# if there are no threads running, exist the 'while' loop

# and start the next action from the list

if not threadPool:

break

# if the touch sensor is pressed then complete everything

if ts.is\_pressed:

stopProcessing = True

break

sleep(0.25)

# if the 'stopProcessing' flag has been set then finish the step loop

if stopProcessing:

break

if \_\_name\_\_ == '\_\_main\_\_':

main()

So far, the main robot program fulfils the following of our original objectives:

* allow activities to run in parallel.
* allow activities to be interrupted immediately.
* allow the robot to automatically select the program to run based on a colour tile placed in front of a dedicated colour sensor.
* allow the programs to be specified in an external file rather than in code.

The only objective to be completed is for the robot to detect when the robot has been lifted off the table and returned to the start of the current run.

# Stopping the Robot when Lifted

So far, we have used the touch sensor to stop operation of the program. If you are lucky enough to have multiple light sensors, you can dedicate one for detecting the attachment using the approach detailed in the previous section. An additional pair can be used to detect lines on the mat and be used to detect when the robot has been lifted.

from ev3dev2.sensor.lego import TouchSensor, ColorSensor

from ev3dev2.sensor.lego import INPUT\_1, INPUT\_2, INPUT\_3, INPUT\_4

COLOUR\_SENSOR\_MAT = INPUT\_3

COLOUR\_SENSOR\_ATTACHMENTS = INPUT\_2

def isRobotLifted():

cl = ColorSensor(COLOUR\_SENSOR\_MAT)

return cl.raw[0] < 5 and cl.raw[1] < 5 and cl.raw[2] < 5

Python25.py

So far in this tutorial, we have relied on the EV3 to work out which input ports our sensors have been plugged in to. Adding a second (or even third) colour sensor to our robot means that we will need to specify which colour sensor is performing which task.

In the above code, I have created two separate variables called COLOUR\_SENSOR\_MAT and COLOUR\_SENSOR\_ATTACHMENTS to track what input port that sensor is using. When creating a reference to the colour sensor, I then use the appropriate variable as shown below.

cl = ColorSensor(COLOUR\_SENSOR\_MAT)

Of course, I could have specified INPUT\_3 directly when creating the colour sensor reference. However, creating the constants at the top of the program allows me to change the value in one spot and have the change update all references automatically. As your program grows, you will likely have actions for driving straight until a line is reached, following a line or squaring up using a line. All of these will refer to the colour sensor and each will need the input specified.

If you only have one colour sensor, you can assign both variables to the same input port for testing.

When the robot is lifted from the mat, the light sensor reads close to zero on all of the red, green and blue channels as very little of the light the unit emits is reflected back to the sensor. I have set an arbitrary threshold of 5 and assume that if the red, green and blue light readings are below this then the robot has been lifted. You may need to play with these values depending on your light conditions.

The code in the main() that used to detect a press of the touch sensor has been changed to the code below. As previously, once it has been detected that the robot has been lifted then the stopProcessing flag is set and the code jumps out of the processing loop. If you recall, this variable was passed to the individual action functions using the lambda keyword and it, in turn, stops their processing.

# if the robot has been lifted then complete everything

if isRobotLifted():

stopProcessing = True

break