Previous:

* Arrays

Overall

Challenge One

* Have lm1 and l,m2 run in parallel for x time.
* Once complete run mm for a period of time.

Consider the code below:

#!/usr/bin/env python3

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

lm1 = LargeMotor(OUTPUT\_B)

lm2 = LargeMotor(OUTPUT\_C)

mm = MediumMotor()

# run these in parallel

lm1.on\_for\_seconds(speed = 50, seconds=2, brake=True)

lm2.on\_for\_seconds(speed = 50, seconds=4, brake=True)

# run this after the previous have completed

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

*Program1.py*

What happens when you run it? The three motors start one after the other.

**In Parallel**

Which completes first?

#!/usr/bin/env python3

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

lm1 = LargeMotor(OUTPUT\_B)

lm2 = LargeMotor(OUTPUT\_C)

mm = MediumMotor()

# run these in parallel

lm1.on\_for\_seconds(speed = 50, seconds=2, brake=True, block=False)

lm2.on\_for\_seconds(speed = 50, seconds=4, brake=True, block=True)

# run this after the previous have completed

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

*Program2.py*

The block does …. By default, if you do not specify the **block** attribute it will default to the **True**.

We can use the block on the second motor as **we know it will finish last**. But what if we don’t know which will finish first?

But what if we do not know which will complete first.  Imagine a more complex example where we are moving forward to a black line or xxx.

#!/usr/bin/env python3

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

lm1 = LargeMotor(OUTPUT\_B)

lm2 = LargeMotor(OUTPUT\_C)

mm = MediumMotor()

# run these in parallel

lm1.on\_for\_rotations(speed = 30, rotations=4, brake=True, block=False)

lm2.on\_for\_rotations(speed = 40, rotations=3, brake=True, block=True)

# run this after the previous have completed

mm.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 3 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

lm1 = LargeMotor(OUTPUT\_B)

lm2 = LargeMotor(OUTPUT\_C)

mm = MediumMotor()

# run these in parallel

lm1.on\_for\_rotations(speed = 30, rotations=4, brake=True, block=False)

lm2.on\_for\_rotations(speed = 40, rotations=3, brake=True, block=False)

lm1.wait\_until\_not\_moving()

lm2.wait\_until\_not\_moving()

# run this after the previous have completed

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

*Program4.py*

So why don’t we just stick with this simple 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()

lm1 = LargeMotor(OUTPUT\_B)

lm2 = LargeMotor(OUTPUT\_C)

mm = MediumMotor()

# run these in parallel

lm1.on\_for\_rotations(speed = 30, rotations=4, brake=True, block=False)

lm2.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:

lm1.off()

lm2.off()

lm1.wait\_until\_not\_moving()

lm2.wait\_until\_not\_moving()

# run this after the previous have completed

mm.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?

When you run a program, it steps through the various tasks you have specified sequentially until it is complete. If your program wants to execute something else in parallel to the code it is running, it can launch a second process, known as a ‘thread’, to run the additional code. Both the main program and secondary instruction set (running in the ‘thread’) run together on the computer’s processor and share memory and other resources but operate independently.

After spawning one or more threads. the main program can track the progress of these other tasks and either wait until they are complete or progress on to other tasks while they finish up. Threads overcome the issues with tasks ‘blocking’ the processor until they are finished.

If we revisit the program *Program2.py* ..

#!/usr/bin/env python3

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

lm1 = LargeMotor(OUTPUT\_B)

lm2 = LargeMotor(OUTPUT\_C)

mm = MediumMotor()

# run these in parallel

lm1.on\_for\_seconds(speed = 50, seconds=2, brake=True, block=False)

lm2.on\_for\_seconds(speed = 50, seconds=4, brake=True, block=True)

# run this after the previous have completed

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

.. we can restructure it slightly as shown below to create a onForSeconds() function that accepts three parameters - motor, speed and seconds. This change allows us to streamline the rest of the code to use the function to turn on the motors using a common function. The reason for this will become crucial as we start implementing our thread code later.

If you are unfamiliar with the additional changes required to get this code to operate - the main() function and the snippet of code if \_\_name\_\_ == '\_\_main\_\_': main() then I suggest you read the article xx for more information. This is fundamental Python and outside the scope of this article.

#!/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():

lm1 = LargeMotor(OUTPUT\_B)

lm2 = LargeMotor(OUTPUT\_C)

mm = MediumMotor()

# run these in parallel

onForSeconds(motor = lm1, speed = 50, seconds = 2)

onForSeconds(motor = lm2, speed = 40, seconds = 3)

lm1.wait\_until\_not\_moving()

lm2.wait\_until\_not\_moving()

# run this after the previous have completed

onForSeconds(motor = mm, 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():

lm1 = LargeMotor(OUTPUT\_B)

lm2 = LargeMotor(OUTPUT\_C)

mm = MediumMotor()

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

threadPool = []

thread1 = threading.Thread(target = onForSeconds, args = (lm1, 30, 4))

thread2 = threading.Thread(target = onForSeconds, args = (lm2, 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 = (mm, 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():

lm1 = LargeMotor(OUTPUT\_B)

lm2 = LargeMotor(OUTPUT\_C)

mm = MediumMotor()

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

threadPool = []

thread1 = threading.Thread(target = onForSeconds, args = (lm1, 30, 4))

thread2 = threading.Thread(target = onForSeconds, args = (lm2, 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 = (mm, 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

from collections import namedtuple

import threading

def onForSeconds(motor, speed, seconds):

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

def main():

actions = []

Action = namedtuple('Action', 'name, motor, speed, seconds')

lm1 = LargeMotor(OUTPUT\_B)

lm2 = LargeMotor(OUTPUT\_C)

mm = MediumMotor()

action1 = Action("onForSeconds", lm1, 20, 4)

action2 = Action("onForSeconds", lm2, 40, 3)

action3 = Action("onForSeconds", mm, 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

from collections import namedtuple

def onForSeconds(motor, speed, seconds):

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

def main():

actions = []

Action = namedtuple('Action', 'name, motor, speed, seconds')

lm1 = LargeMotor(OUTPUT\_B)

lm2 = LargeMotor(OUTPUT\_C)

mm = MediumMotor()

action1 = Action("onForSeconds", lm1, 20, 4)

action2 = Action("onForSeconds", lm2, 40, 3)

action3 = Action("onForSeconds", mm, 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

from collections import namedtuple

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 = False)

def main():

threadPool = []

actions = []

Action = namedtuple('Action', 'name, motor, speed, seconds')

lm1 = LargeMotor(OUTPUT\_B)

lm2 = LargeMotor(OUTPUT\_C)

mm = MediumMotor()

action1 = Action("onForSeconds", lm1, 20, 4)

action2 = Action("onForSeconds", lm2, 40, 3)

action3 = Action("onForSeconds", mm, 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 collections import namedtuple

from time import sleep

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 = False)

def delayForSeconds(seconds):

sleep(seconds)

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 = []

Action = namedtuple('Action', 'name, motor, speed, seconds')

lm1 = LargeMotor(OUTPUT\_B)

lm2 = LargeMotor(OUTPUT\_C)

mm = MediumMotor()

action1 = Action("onForSeconds", lm1, 20, 4)

action2 = Action("onForSeconds", lm2, 40, 3)

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

action4 = Action("onForSeconds", mm, 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

 So can we kill it yet?

 Not quite but we can stop it between steps.

#!/usr/bin/env python3

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

from ev3dev2.sensor.lego import TouchSensor

from collections import namedtuple

from time import sleep

import threading

def onForSeconds(motor, speed, seconds):

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

def delayForSeconds(seconds):

sleep(seconds)

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 = []

stopProcessing = False

Action = namedtuple('Action', 'name, motor, speed, seconds')

lm1 = LargeMotor(OUTPUT\_B)

lm2 = LargeMotor(OUTPUT\_C)

mm = MediumMotor()

ts = TouchSensor()

action1 = Action("onForSeconds", lm1, 20, 4)

action2 = Action("onForSeconds", lm2, 40, 3)

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

action4 = Action("onForSeconds", mm, 10, 8)

actionParallel = []

actionParallel.append(action1)

actionParallel.append(action2)

actions.append(actionParallel)

actions.append(action3)

actions.append(action4)

for action in actions:

while True:

# 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)

# 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, complete everything

if ts.is\_pressed:

stopProcessing = True

break

sleep(0.25)

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

if stopProcessing:

break

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

main()

*Python13.py*

Upload and test the Python13.py program shown 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.

But why did it not stop immediately?

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.

So how do we stop the code as soon as the robot is lifted from the table (or in our case, when the touch sensor) is pressed?

# Dead, once and for all.

How do we stop the stop the program as soon as the robot is lifted?

The code below shows one solution which uses ‘non-blocking’ threads to do its work. If you compare the code below to that of program *Python13.py*, you will see significant changes - especially in the ‘action’ definitions – that are described in detail after the program listing.

#!/usr/bin/env python3

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

from ev3dev2.sensor.lego import TouchSensor

from collections import namedtuple

from time import sleep

import threading

import time

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 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

Action = namedtuple('Action', 'name, motor, speed, seconds')

lm1 = LargeMotor(OUTPUT\_B)

lm2 = LargeMotor(OUTPUT\_C)

mm = MediumMotor()

ts = TouchSensor()

action1 = Action("onForSeconds", lm1, 20, 4)

action2 = Action("onForSeconds", lm2, 40, 3)

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

action4 = Action("onForSeconds", mm, 10, 8)

actionParallel = []

actionParallel.append(action1)

actionParallel.append(action2)

actions.append(actionParallel)

actions.append(action3)

actions.append(action4)

for action in actions:

while True:

# 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)

# 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, complete everything

if ts.is\_pressed:

stopProcessing = True

break

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 simply 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

Recursion.

def printLowerNumber(n):

  print(“{}, ”.format(n);

  if n>1:

    printLowerNumber(n-1)

# Main program

printLowerNumber(10)

Output should be:

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

DEBUG