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?

Lets revisit 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)

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():

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

Dead once and for all.

#!/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()

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