# Threading

Threading in python is used to run multiple threads (tasks, function calls) at the same time. Note that this does not mean that they are executed on different CPUs. Python threads will NOT make your program faster if it already uses 100 % CPU time. In that case, you probably want to look into parallel programming.

Python threads are used in cases where the execution of a task involves some waiting. Threading allows python to execute other code while waiting; this is easily simulated with the sleep function.

Threading allows us to run multiple tasks run concurrently, namely running multiple tasks independently, like while task A is running I can run task B and I don’t have to wait for task A to finish; this means running tasks concurrently.

from time import sleep, time

import threading

def calculate\_square(nums):

print('Calculating Square:')

for num in nums:

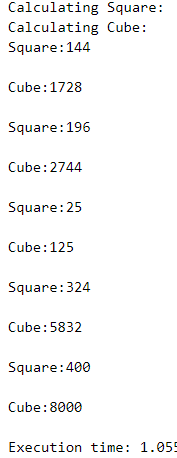
sleep(0.2)

print('Square:' + str(num \* num) + '\n')

def calculate\_cube(nums):

print('Calculating Cube:')

for num in nums:

 sleep(0.2)

print('Cube:' + str(num \* num \* num) + '\n')

t= time()

numbers = [12, 14, 5, 18, 20]

t1 = threading.Thread(target=calculate\_square, args=(numbers,))

t2 = threading.Thread(target=calculate\_cube, args=(numbers,))

t1.start()

t2.start()

t1.join()

t2.join()

print('Execution time:', float(time()-t))

## PyMoondra

## 1stVid

import threading

import time

def sleeper(secs, name):

    print(f'{name} here, sleeping for {secs} second(s)')

    time.sleep(secs)

    print('Done sleeping')

t = threading.Thread(target=sleeper, args=(5, 'Thread 01'))

t.start()

# join() - is called a blocking method, because it blocks the interpretor

# or Python from running main program until the thread finishes its task.

t.join()   # to prevent jumping into main thread

print('Hello world, from main thread')

\_\_\_\_\_\_\_\_|Output|\_\_\_\_\_\_\_\_\_

# Thread 01 here, sleeping for 5 second(s)

# Done sleeping

# Hello world, from main thread

## 2ndVid

import threading

import time

def sleeper(secs, name):

    print(f'{name} here, sleeping for {secs} second(s)')

    time.sleep(secs)

    print(f'{name} is finished now.')

threads\_list = []

t1 = time.time()

for i in range(5):

    t = threading.Thread(target=sleeper, name=f'Thread #{i}', args=(5, f'Thread #{i}'))

    threads\_list.append(t)

    t.start()

for t in threads\_list:

    t.join()

print(f'Execution time: {round(time.time() - t1, 2)}')

print('Hello world, from main thread')

\_\_\_\_\_\_\_\_|Output|\_\_\_\_\_\_\_\_\_

# Thread #0 here, sleeping for 5 second(s).

# Thread #0 has started.

# Thread #1 here, sleeping for 5 second(s).

# Thread #1 has started.

# Thread #2 here, sleeping for 5 second(s).

# Thread #2 has started.

# Thread #3 here, sleeping for 5 second(s).

# Thread #3 has started.

# Thread #4 here, sleeping for 5 second(s).

# Thread #4 has started.

# Thread #2 is finished now.

# Thread #1 is finished now.

# Thread #0 is finished now.

# Thread #3 is finished now.

# Thread #4 is finished now.

# Execution time: 5.0

# Hello world, from main thread

## 3rd Video

Some threads do background tasks, like sending keepalive packets, or performing periodic garbage collection, or whatever. These are only useful when the main program is running, and it's okay to kill them off once the other, non-daemon, threads have exited.

Without daemon threads, you'd have to keep track of them, and tell them to exit, before your program can completely quit. By setting them as daemon threads, you can let them run and forget about them, and when your program quits, any daemon threads are killed automatically.

***Durga Software Solution:*** The thread which is executing in the background is called daemon thread. the main purpose of daemon thread is to provide support for main threads.

for example: garbage collector, signal dispatcher, …

***Quora:*** Daemon threads are like assistants. Non-Daemon i.e. user threads are like front performers. Assistants help performers to complete a job. When the job is completed, no help is needed by performers to perform anymore. As no help is needed the assistants leave the place. So when the jobs of Non-Daemon threads is over, Daemon threads march away.

Daemon threads are "background" threads that provides service to other threads.

***Quora:*** We make threads in Java to do multitasking i.e. one thread doing some processing (maybe listening to UI actions) and some other thread doing background tasks such as fetching data from database.

Suppose, we make a game and there is background music in our game. The music will be played in a separate thread. Now, if the game gets over i.e. the execution of main thread finishes, then we wish to stop the music too. The JRE should not wait for the music thread to finish its execution (i.e. namely play the entire sound clip), so we can make the music thread as a **daemon thread.**

**Daemon Threads** are considered as secondary threads. That means, JRE doesn’t wait for the completion of a daemon thread. If the execution of all non-daemon threads gets over, JRE will terminate the program without bothering about the completion of the daemon thread.

By default, all threads are non-daemon, we can set any thread as daemon before starting it.

import threading

import time

total = 4

def create\_items():

    global total

    for i in range(10):

        time.sleep(2)

        print('added item\n')

        total += 1

    print('creation is done\n')

def create\_items\_2():

    global total

    for i in range(7):

        time.sleep(1)

        print('added item\n')

        total += 1

    print('creation is done\n')

def limits\_items():

    global total

    while True:

        if total > 5:

            print ('overload\n')

            total -= 3

            print('subtracted 3\n, current value is', total)

        else:

            time.sleep(1)

            print('waiting\n')

creator1 = threading.Thread(target  = create\_items)

creator2 = threading.Thread(target = create\_items\_2)

limitor = threading.Thread(target = limits\_items, daemon = True)

print(limitor.isDaemon())

creator1.start()

creator2.start()

limitor.start()

creator1.join()

creator2.join()

print('our ending value of total is' , total)

## 4th Video - Sub-classing 01

# according to the dots it is advisable to change \_\_init\_\_() method and run() methods.

**# overriding and modifying the run() method in this example**

import time

import threading

class MyThread(threading.Thread):

    def run(self):

        print('{} has started!'.format(self.getName()))

        try:

            if self.\_target:

                self.\_target(\*self.\_args, \*\*self.\_kwargs)

        finally:

            del self.\_target, self.\_args, self.\_kwargs

        print('{} has finished!'.format(self.getName()))

def sleeper (n, name) :

    print('Hi, I am {}. Going to sleep for 5 seconds\

          \n'.format(name))

    time.sleep(n)

    print('{} has woken up from sleep \n'.format(name))

for i in range(4):

    t = MyThread(target = sleeper,

                 name = 'thread {}'.format(i+1),

                 args = (3, 'thread {}'.format(i+1)))

    t.start()

**# overriding and modifying the \_\_init\_\_() method in this example**

# overriding \_\_init\_\_

# def \_\_init\_\_(self, group=None, target=None, name=None,

#  args=(), kwargs=None, \*, daemon=None):

import time

import threading

class CustomThread(threading.Thread):

    def \_\_init\_\_(self, number, func, args):

        threading.Thread.\_\_init\_\_(self)

        self.number = number

        self.func = func

        self.args = args

    def run(self):

        print('thread {} has started'.format(self.number))

        self.func(\*self.args)

        print('thread {} has finished'.format(self.number))

def double(number, cycles):

    for i in range(cycles):

        number += number

    print(number)

threads\_list = []

for i in range(5):

    t = CustomThread(number = i + 1, func = double, args = [i, 3])

    threads\_list.append(t)

    t.start()

for t in threads\_list:

    t.join()

## 5th Video - Sub-classing 02

import threading

import time

class MyThread(threading.Thread):

    def \_\_init\_\_(self, number, style, \*args, \*\*kwargs):

# this is kinda pulling out the original source code and extracting it here

        super(MyThread, self).\_\_init\_\_(\*args, \*\*kwargs)

        self.number = number

        self.style = style

    def run(self, \*args, \*\*kwargs):

        print('thread starting')

# this is kinda pulling out the original source code and extracting it here

        super(MyThread, self).run(\*args, \*\*kwargs)

        print('thread has ended')

def sleeper (num, style):

    print('we are sleeping for {} seconds as {}'.format(num, style))

    time.sleep(num)

# def \_\_init\_\_(self, group=None, target=None, name=None,

#  args=(), kwargs=None, \*, daemon=None):

t = MyThread(number =3, style = 'yellow', target = sleeper, args = [3, 'yellow'])

t.start()

# 06 - Locks - Threading

locks are usually used when you have many threads tryin to access same variable. Usually a global variable.

import threading

lock = threading.Lock()

x = 0

count = 1000

def add\_3():

    global x

    with lock:

        for i in range(count):

            x += 3

def add\_4():

    global x

    with lock:

        for i in range(count):

            x += 4

def subtract\_2():

    global x

    with lock:

        for \_ in range(count):

            x -= 2

def subtract\_5():

    global x

    lock.acquire()

    for \_ in range(count):

        x -= 5

    lock.release()

t1 = threading.Thread(target=add\_3)

t2 = threading.Thread(target=subtract\_2)

t3 = threading.Thread(target=subtract\_5)

t4 = threading.Thread(target=add\_4)

t1.start()

t2.start()

t3.start()

t4.start()

t1.join()

t2.join()

t3.join()

t4.join()

print(x)

## 5th Video - Queue

Queue - are container of data items/ comments that are waiting to be retrieved. Say you are receiving data from the website then you are manipulating the data and then writing the data to a file, so, there’re

There’re three types of Queues:

1. FIFO (first in first out)
2. LIFO (last in first out)
3. Priority

import queue

# this is a simple FIFO queue

q = queue.Queue()

# to put something in this queue

q.put()

# to get an item from queue

print(q.get())

# to check if queue is empty

print(q.empty())

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# FIFO

import queue

q = queue.Queue()

for i in range(5):

    q.put(i)

while not q.empty():

    print(q.get(), end = '   ')

# 0   1   2   3   4

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# LIFO

import queue

q = queue.LifoQueue()

for i in range(5):

    q.put(i)

while not q.empty():

    print(q.get(), end = '   ')

# 4   3   2   1   0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

import queue

q = queue.PriorityQueue()

q.put(30)

q.put(5)

q.put(1)

q.put(2)

for i in range(q.qsize()):

    print(q.get(), end='  ')

# 1  2  5  30

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

import queue

q = queue.PriorityQueue()

q.put((1, 'Priority 1'))

q.put((3, 'Prioirty 3'))

q.put((4 ,'Priority 4'))

q.put((2 ,'Priority 2'))

for i in range(q.qsize()):

    print(q.get()[1], )

# Priority 1 Priority 2 Prioirty 3 Priority 4