תהליכונים

עד עכשיו מרבית התוכניות אם לא כולן התנהלו בקצב סינכרוני- כל פקודה התבצעה ברגע שהגענו אליה.
אבל לפעמים תהליך סינכרוני כזה הוא דווקא לרעתנו , למשל הכפלה של שתי מטריצות גדולות מאוד, אם נעשה זאת בתהליך סינכרוני נסיים את ההכפלה רק אחרי שעברנו כל שורה ועמודה של המטריצות, אבל אם היינו במקום מכפילים כמה שורות בעמודות במקביל התהליך הכללי היה לוקח הרבה פחות זמן. בדיוק בשביל זה יש תהליכונים.
תהליכונים או תרדים (threads) הם תת תהליך שמתבצע במקביל לתהליך המרכזי, ומאפשרים לבצע חישובים א-סינכרוניים.
לפייתון יש בספרייה הסטנדרטית כמה מודולים שמתאים מראש לשימוש בתהליכונים, למשל המודול threading שיש לה מחלקה בשם Thread(). המחלקה מקבלת לתוכה כפרמטר פונקציית מטרה ,שאותה היא אמורה להפעיל, והפרמטרים שלה, וכדי להפעיל אותה נצטרך להשתמש במתודה start() של האובייקט.
למשל נסתכל על הקוד הבא:

```
import time

start = time.perf_counter()

def do_something(time_to_run = 1):
    print(f'Sleeping {time_to_run} sec...')
    time.sleep(time_to_run)
    print('Done sleeping')

do_something(1)
do_something(1)

finish = time.perf_counter()
print(f'Finished in {round(finish-start,2)} second(s)')
```

```
Sleeping 1 sec...
Done sleeping
Sleeping 1 sec...
Done sleeping
Finished in 2.0 second(s)
```

הפונקציה בסה"כ קוראת לפונקציה do_something() פעמיים, והפונקציה כל פעם 'ישנה' ,כלומר עוצרת את התוכנית למשך שנייה ובסוף הקריאות מחשבת כמה יצא כל התהליך. סה"כ הוא רץ שתי שניות.
עכשיו בא נראה איך היינו ממשים את אותה התוכנית רק עם תרדים:

```
import time
import threading

start = time.perf_counter()

def do_something(time_to_run = 1):
    print(f'Sleeping {time_to_run} sec...')
    time.sleep(time_to_run)
    print('Done sleeping')
```

```
t1 = threading.Thread(target = do_something ,args = [1.5] )
t2 = threading.Thread(target = do_something ,args = [1.5] )
t1.start()
t2.start()

finish = time.perf_counter()
print(f'Finished in {round(finish-start,2)} second(s)')
```

```
Sleeping 1.5 sec...
Sleeping 1.5 sec...Finished in 0.04 second(s)
```

הנה משהו מעניין, הרצנו את התוכנית שקראה לתהליכונים שרצו במקביל לתהליך הראשי, אבל מה שקרה זה שהתוכנית נגמרה לפני שהתהליכונים סיימו להתבצע (לא היו הדפסות done..) .
זה משום שהתהליכונים תלויים בתהליך הראשי, ואם הוא הסתיים לפניהם הם יגמרו איתו.
אם נרצה שהתוכנית הראשית תחכה לסיום התהליכונים כדי להסתיים נשתמש במתודה join() של התהליכונים כדי להראות לתהליך הראשי לחכות להם:

```
import time
import threading

start = time.perf_counter()

def do_something(time_to_run = 1):
    print(f'Sleeping {time_to_run} sec...')
    time.sleep(time_to_run)
    print('Done sleeping')

t1 = threading.Thread(target = do_something ,args = [1.5] )
t2 = threading.Thread(target = do_something ,args = [1.5] )
t1.start()
t2.start()
t1.join()
t2.join()

finish = time.perf_counter()
print(f'Finished in {round(finish-start,2)} second(s)')
```

```
Sleeping 1.5 sec...
Sleeping 1.5 sec...
Done sleeping
Done sleeping
Finished in 1.52 second(s)
```

עכשיו התהליך הראשי חיכה לשני התהליכונים הקטנים, ואפילו ניתן לראות כמה המהירות של התוכנית השתפרה כתוצאה משימוש בתהליכונים, ובמקום שכל התוכנית תיקח 3 שניות (כי הפעלנו את הפונקציה עם הפרמטר 1.5) היא לקחה 1.52 שניות ממש טיפה יותר מחצי מהזמן.

- MULTI THREADING

אם תהליכונים זה דבר כל-כך הרי שלא נסתפק רק באחד, נרצה להפעיל כמה שניתן (באופטימליות מרבית).
הדרך האינטואיטיבית אומרת שכדי להפעיל כמה תרדים נבנה רשימה של תרדים ונריץ אותם בלולאה:

```
start = time.perf_counter()
threads = []
for _ in range(10):
    t = threading.Thread(target = do_something ,args = [1] )
    t.start()
    threads.append(t)
for thread in threads:
    thread.join()
finish = time.perf_counter()
print(f'Finished in {round(finish-start,2)} second(s)')
```

```
Sleeping 1 sec...
Sleeping 1 sec...
Sleeping 1 sec... Sleeping 1 sec...
Sleeping 1 sec...
Sleeping 1 sec...
Sleeping 1 sec...
Sleeping 1 sec...
Sleeping 1 sec... Sleeping 1 sec...
Done sleeping Done sleepingDone sleeping
Done sleeping Done sleeping
Done sleepingDone sleeping
Done sleeping Done sleeping
Done sleeping
Finished in 1.12 second(s)
```

התוכנית נגמרה לאחר שנייה וקצת במקום תוך עשר שניות, אכן ניכר השיפור.
מעניין גם לראות שחלק מההדפסות נכנסו אחת בשנייה.
השיטה הזאת היא לגיטימית לחלוטין, ואפילו היתה מקובלת עד לא מזמן, אבל החל מפייתון גירסה 3.2 נכנס מודול חדש לספרייה הסטנדרטית שיכול לטפל במה שנקרא thread-pool , או מאגר תרדים.
המודול החדש הוא concurrent ויש לו מרחב שם שנקרא futures , בתוך futuers יש מחלקה שנקראת ThreadPoolExecutor() ונהוג להפעיל אותה בcontext manager. המחלקה מפעילה אוטומטית תרדים ומצרפת אותם לתהליך הראשי בתוך ה-context manager דרך מתודה שנקראת submit() , בשביל הדוגמא נשנה קצת את הפונקציה do_something כך שתחזיר את הערך במקום להדפיס אותו:

```
def new_do_something(time_to_run):
    print(f'Sleeping {time_to_run} sec...')
    time.sleep(time_to_run)
    return f'Done sleeping for {time_to_run}'
```

```
import concurrent.futures
import time
```

```
start = time.perf_counter()

with concurrent.futures.ThreadPoolExecutor() as exect:
    f1 = exect.submit(new_do_something , 1)
    print(f1.result())

finish = time.perf_counter()
print(f'Finished in {round(finish-start,2)} second(s)')
```

```
Sleeping 1 sec...
Done sleeping for 1
Finished in 1.03 second(s)
```

שימו לב שהמשתנה f1 הוא אובייקט שנוצר מהמחלקה ThreadPoolExecutor והוא שומר בתוכו את ערך הקבלה, וכדי לקבל אותו היינו צריכים להשתמש בפונקציה result(), כמו כן בכלל לא התשתמשנו בפונקציה join() כדי לצרף את התרד לתוכנית הראשית.
אם נרצה להפעיל כמה תרדים בבאת נוכל לבצע זאת בתוך רשימה כפי שעשינו קודם, וכדי לראות את התוצאות נוכל להשתמש בפונקציה as_completed() שמקבלת את הרשימה של התרדים ושומרת בתוכה את התרדים שסיימו את ריצתם:

```
start = time.perf_counter()

with concurrent.futures.ThreadPoolExecutor() as exect:
    secs = [1,2,3,4,5]
    results = [exect.submit(new_do_something , i) for i in secs]
    for res in concurrent.futures.as_completed(results):
        print(res.result())

finish = time.perf_counter()
print(f'Finished in {round(finish-start,2)} second(s)')
```

```
Sleeping 1 sec...
Sleeping 2 sec...
Sleeping 3 sec...Sleeping 4 sec...
Sleeping 5 sec...
Done sleeping for 1
Done sleeping for 2
Done sleeping for 3
Done sleeping for 4
Done sleeping for 5
Finished in 5.08 second(s)
```

הערה בקשר ל-ThreadPoolExecutor : המחלקה עלולה לגרום לכמה שגיאות מבלבלות.
למשל אם העברנו לפונקצית מטרה ללא פרמטרים איזשהו ארגומנט התרד אמור לזרוק חריגה, אבל לצערנו המחלקה ThreadPoolExecutor תסתיר את החריגה הזו, והתוכנית תיגמר ללא פלט. זה עלול להיות די מבלבל לדאבג את זה בהתחלה.

מרוץ תהליכונים-

מרוץ תהליכונים הוא מצב בו כמה תהליכון מתחרים ביניהם על אותו משאב. משום שהגישה למשאב מוגבלת למספר מסויים של תהליכונים (לרוב אחד), אנחנו מקבלים מצב בו מתקיימת תחרות בין התרדים בה כל הקודם

זוכה.
במקרה כזה נרצה לסנכרן חלקים מהתוכנית, כך שלא כל התרדים יוכלו להיכנס לכל הפונקציות בבאת אחת.
הדוגמא הבאה תמחיש לנו את זה הכי טוב:

```
format = "%(asctime)s: %(message)s"
logging.basicConfig(format=format, level=logging.INFO,datefmt="%H:%M:%S")

class FakeDatabase:
    def __init__(self):
        self.value = 0

    def update(self, name):
        logging.info("Thread %s: starting update", name)
        local_copy = self.value
        local_copy += 1
        time.sleep(1)
        self.value = local_copy
        logging.info("Thread %s: finishing update", name)

database = FakeDatabase()
logging.info("Testing update. Starting value is %d.", database.value)
with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
    for index in range(2):
        executor.submit(database.update, index)
logging.info("Testing update. Ending value is %d.", database.value)
```

הפונקציה basicConfig מבצעת לוגיניג לקונסול, כלומר היא מדפיסה למסך המקום לקובץ לוג.
מה שקורה בקוד הוא שהגדרנו מחלקה שקוראים לה FakeDatabase ויש לה משתנה שקוראים לו value,
למחלקה יש מתודה בשם update() והיא אמורה לעדכן את ערכו של המשתנה value (לעלות אותו פלוס אחד).
בתוכנית הראשית יצרנו שני תהליכונים וכל תהליכון מפעיל את המתודה update(), לכאורה היינו מצפים שבסוף התהליך הערך של value יהיה 2, כי שני תהליכונים מבצעים אותה, אבל מה שקורה בפועל הוא שהערך הוא 1 עדיין.
וזה משום ששני התהליכונים נכנסו למתודה במקביל. התהליכון הראשון קיבל את הערך של value (שבהתחלה הוא 0) שמר אותו במשתנה, הוסיף למשתנה אחד והלך לישון לשנייה, בנתיים התהליכון השני נכנס למתודה קיבל את הערך של value שעדיין לא התשנה, שמר אותו במשתנה נפרד והלך לישון, ואז(או שבמקביל) התהליכון הראשון מתעורר ומעדכן את value להיות שווה 1, ואח"כ גם התהליכון השני מתעורר ומעדכן את value להיות שווה 1.
כדי לפתור את זה נצטרך להשתמש בסינכרון בסיסי, במודול threading יש מחלקה שקוראים לה Lock והיא דואגת שקטע קוד מסויים יהיה סגור ע"י שתי מתודות- aquire() שמגדירה שהקט קוד הבא הוא סגור למספר מצומצם של תהליכונים, ו- realse() שמשחרר את הסגירה על הקטע קוד. המחלקה תומכת ב-context manager וכדי לא להתבלבל מומלץ להשתמש ב-with ולא בשתי המתודות.
בואו נתקן את המחלקה ,נוסיף משתנה עצם של המחלקה מסוג מטיפוס Lock וניצור פונקציה חדשה שבה אנחנו משתמשים בנעילה :

```
class FakeDatabase:
    def __init__(self):
        self.value = 0
        self._lock = threading.Lock()
```

```
    def locked_update(self, name):
        logging.info("Thread %s: starting update", name)
        logging.debug("Thread %s about to lock", name)
        with self._lock:
            logging.debug("Thread %s has lock", name)
            local_copy = self.value
            local_copy += 1
            time.sleep(0.1)
            self.value = local_copy
            logging.debug("Thread %s about to release lock", name)
        logging.debug("Thread %s after release", name)
        logging.info("Thread %s: finishing update", name)
```

נבצע את אותו קוד שהשתמשנו בו קודם, רק שבמקום הפונקציה update() נשתמש בפונקציה locked_updete :

```
database = FakeDatabase()
logging.info("Testing update. Starting value is %d.", database.value)
with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
    for index in range(2):
        executor.submit(database.locked_update, index)
logging.info("Testing update. Ending value is %d.", database.value)
```

```
17:36:32: Testing update. Starting value is 0.
17:36:32: Thread 0: starting update
17:36:32: Thread 1: starting update
17:36:32: Thread 0: finishing update
17:36:32: Thread 1: finishing update
17:36:32: Testing update. Ending value is 2.
```

לסיום, הנה עוד סיבה למה להשתמש ב-context manager- ניתן להשתמש בפונקציה aquire() של Lock יותר מפעם אחת אחרי השני, מה שעלול לגרום לבעיה שנקראת deadlock שבה נוצר פקק בקבוק של תהליכונים שמחכים להשתחרר, לרוב זה קורא כי מישהו לא שיחרר aquire() או כי יש יותר מידי תהליכונים שמעמיסים על המעבד לחלק את המשאבים.