שיפור זמן ריצה

לפייתון יש תדמית של שפת תכנות חזקה, גמישה, ורסטילית ופשוטה.
זה מה שהופך אותה לאחת השפות הפעילות ביותר בשוק והשימוש בשפה רק גדל עם הזמן.
אך עם כל כוחה של השפה, העיצוב שלה- השימוש במפרש במקום במהדר, משתנים דינאמיים ולא סטטיים ועוד, גורמים לשפה להיות איטית יותר משפות מכונה מקוריות כמו c/c++ , ולפעמים בצורה משמעותית.
במהלך השנים מתכנתים יצרו כמה כלים לשיפור זמן הריצה של השפה, למשל האפשרות לכתוב קוד שלם ב- c או c++ ולחבר אותו לקוד קיים בפייתון, כך החלק הכבד של הקוד ירוץ במהירות המרבית של שפת מכונה ותוך כדי יהיה ניתן להשתמש בספריות של פייתון, ובאמת כבר יצא לנו לראות כמה ספריות פייתון שמשתמשות ב-c כבסיס לתוכנית , למשל nupy או pandas.
במסמך הבא נסקור כמה כלים שיכולו לעזור לנו לשפר את זמן ריצת התוכנית.

CYTHON -

הוא כלי תכנות לפייתון שמאפשר לנו להפוך את פייתון לסוג של שפה סטטית ובכך הוא משפר את זמן הריצה של התוכנית, ועוזר למנוע שגיאות תכנות נפוצות בשפות דינמיות.
cython הוא שילוב כוחות בין פייתון לשפת c, מה שנותן לנו את האופציה לכתוב קוד פייתון שניתן להעביר לתוכניות ב-c או c++.
התקנה: בשביל להשתמש ב-cython צריך, מלבד שיהיה לנו פייתון במחשב, גם קומפיילר מותקן ל-c. מערכות הפעלה של לינוקס אמורות להגיע כבר עם קומפיילר, ואפשר להתקין תוסף ל-visual studio code למי שמשתמש.
בנוסף צריך להתקין את המודול:

```
pip install cython
```

החלק המסובך ביותר בתהליך הוא להבין כיצד לקמפל קוד פייתון לקוד c ולהפך.
קבצי cython הם קבצים עם הסיומת .pyx'' לאחר שכתבנו את הקוד ניצור כמין קובץ makefile שכתוב בפייתון.
הקובץ setup הזה אמור להכיל :

```
from setuptools import setup
from Cython.Build import cythonize
```

לצורך הדוגמא הקובץ שלנו הוא helloworld.pyx והוא מכיל רק פונקציה אחת והיא הדפסה של "hello world" :

```
print('Hello world')
```

אז בקובץ setup , מתחת לייבוא הספריות, נצטרך להשתמש בפונקציה setup() כדי להמיר את הקוד ל-c או ל-python:

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    ext_modules = cythonize("hello.pyx")
)
```

נשמור את הקובץ, למשל כ-setup.py ונקמפל אותו כך:

```
python setup.py build_ext --inplace
```

עכשיו אמור להיווצר קובץ חדש- בלינוקס הוא אמור להיקרא helloworld.so ובווינדוס helloworld.pyd.
נפעיל את המצב האינטראקטיבי ונראה שנוכל לייבא כעת את הקובץ שייצרנו:

```
>>> import helloworld

Hello world
```

סינטקס:
יש שתי משפחות טיפוסי נתונים ב- cython.
הראשונה היא cdef – והם כל אותם טיפוסי נתונים שמוכרים ב-c כולל מצביעים, מערכים, double ועוד, שאינם קיימים בפייתון :

```
cdef int i, j, k
cdef float f, g[42], *h
```

או אבייקטים ב-c מתקדמים יותר כמו מבנה או union:

```
cdef struct Grail:
    int age
    float volume

cdef union Food:
    char *spam
    float *eggs

cdef enum CheeseType:
    cheddar, edam,
    camembert

cdef enum CheeseState:
    hard = 1
    soft = 2
    runny = 3
```

המשפחה השנייה היא משפחת האובייקט פייתון- אותם אובייקטים שלא הגדרנו אותם כמשתני c, אז הקומפיילר מקמפל אותם כאובייקטים מיוחדים של פייתון.
חוץ מהאובייקטים יש גם שלושה סוגי פונקציות: סוג אחד הן פוקציות c שמוגדרות כ-cdef, ניתן להשתמש בהן חופשי בתוך קובצי pyx או לקרוא להן בקובץ c , אבל אי אפשר להשתמש בהן בקובץ פייתון.
סוג נוסף הוא פונקציית פייתון רגילה שמוגדרת כ-def, ויש עוד סוג שמשלב בין השניים הקודמים והוא פונקציית cpdef, שאומנם היא לא מהירה כמו cdef אך היא מהירה יותר מפונקציית def רגילה של פייתון.
מומלץ להתייחס לפונקציות cdef כמו לפונקציות 'פרטיות' של מחלקות, כלומר שנשתמש בהן בתוך המודול כדי לחשב דברים מתוך המודול, אבל פונקציות עבור המשתמש נעדיף פונקציות cpdef ולא def.
לסיום נראה דוגמא לזמן להבדל בין זמן ריצה של תוכנית פייתון רגילה לתוכנית ב- cython.
יצרנו שני קבצים שכל מה שהם עושים זה לחשב סכום אינטרוול :

פייתון:

```
def test(x):
    y=0
    for i in range(x):
        y+=i
    return y
```

:cython

```
cpdef int test(int x):
    cdef int y=0
    cdef int i = 0
    for i in range(x):
        y+=i
    return y
```

והשתמשנו במודול timeit שמאפשר להריץ פונקציה מתוך מודול כמה פעמים, ולמדוד כמה זמן לקח להריץ אותה בסה"כ :

```
import timeit
cy = timeit.timeit('cy_example.test(500)',
                   setup='import cy_example',
                   number = 10000 )
py = timeit.timeit('py_example.test(500)',
                   setup='import py_example',
                   number = 10000 )
print(f'cy = {cy}')
print(f'py = {py}')
print(f'Cython is {py/cy}x faster')
```

```
cy = 0.003027751000000002
py = 0.279153536
Cython is 92.19831353370861x faster
```

יצא ש-cython מהירה פי בערך 92 יותר מתוכנית פייתון רגילה.
זהו cython על קצה המזלג, יש עוד הרבה (מאוד) חומר על הספרייה, ומומלץ להסתכל ב[מדריך הרשמי שלהם](#).

CPPYY -

Cppyy היא ספרייה המקשר בין קודים בשפת c++ לסקריפטים של פייתון, בלי מתווכים או פונקציות מסובכות מידי בין הקודים השונים, ובכך מאפשרת למתכנתים ליהנות משני העולמות- פונקציונליות מהירה של c++ וממשק משתמש של פייתון.
בשביל להשתמש בספרייה אנחנו צריכים להתקין על המחשב איזשהו קומפיילר gcc, clang11 : c++ וכו'.
בנוסף צריך להוריד את המודול:

```
pip install cppyy
```

אז איך זה עובד? יש שלוש שכבות ל-cppyy. בשכבה הראשונה יש את המודל cppyy.gbl שמייצג את המרחב שם הגלובלי (the global namespace), בשכבה השנייה יש מגוון פונקציות עזר, ומתחת יש סט של תת-מודולים שמשרתים מטרה ספציפית.
כשרוצים להגדיר מחלקה חדשה/פונקציה מ-cpp ולהשתמש בה בקוד, ניצור אותה בפונקציה cppdef() שמקבלת docstring עם הפונקציה.

```
import cppyy
cppyy.cppdef("""
class Integer1 {
public:
    Integer1(int i) : m_data(i) {}
    int m_data;
};""")
```

כרגע המחלקה שיצרנו עדיין נמצאת בצד של קוד ה-++c, ומוגבלת תחת החוקים של השפה.
בשביל להגיע למחלקה שיצרנו נצטרך להשתמש במודל הגלובלי, כלומר cppyy.gbl.integer1:

```
print(cppyy.gbl.Integer1)
```

```
<class cppyy.gbl.Integer1 at 0x000000000C2BD6F0>
```

אובייקטים מקושרים מ-++c לפייתון נחשבים כאובייקטים של פייתון לכל דבר, וכאלה אנחנו יכולים לאתחל אותם, להשתמש בכלי עזר של פייתון עליהם, הם זורקים שגיאה של פיתון במקרה של חריגה, הזיכרון שהם תופסים מנוהל לפי ה-garbage collector של פייתון וכו'.
יותר מזה אנחנו יכולים לשלב אותם עם אובייקטים אחרים של ++c.

```
# for convenience, bring Integer1 into __main__
from cppyy.gbl import Integer1

# create a C++ Integer1 object
i = Integer1(42)

# use Python inspection
print("Variable has an 'm_data' data member?", hasattr(i, 'm_data') and 'Yes!' or 'No!')
print("Variable is an instance of int?", isinstance(i, int) and 'Yes!' or 'No!')
print("Variable is an instance of Integer1?", isinstance(i, Integer1) and 'Yes!' or 'No!')
```

```
Variable has an 'm_data' data member? Yes!
Variable is an instance of int? No!
Variable is an instance of Integer1? Yes!
```

הספרייה הסטנדרטית של ++c כבר שמורה במשתנה gbl וניתן לגשת אליה ישירות:

```
# pull in the STL vector class
from cppyy.gbl.std import vector

# create a vector of Integer1 objects; note how [] instantiates the template and () instantiates the
class
v = vector[Integer1]()

# populate it
v += [Integer1(j) for j in range(10)]

# display our vector
print(v)
```

```
<cppyy.gbl.std.vector<Integer1> object at 0x336ddb0>
```

הוקטור לא נראה כל כך יפה, אבל עכשיו שהוא אובייקט של פייתון אנחנו יכולים לשפץ אותו כמו כל אובייקט אחר של פייתון ולהוסיף לו מתודת __str__ וכו'.

דרך נוספת לייבא קוד היא ע"י שימוש בקובצי header שהגדרנו מראש, נוכל לקרוא להם עם הפונקציה include() שמקבלת במחרוזת את שם הקובץ כארגומנט.
נראה דוגמא, נניח יש לנו את הקובץ hello.hpp 'header' הבא:

```
#include <iostream>

void say_hello() {
    std::cout << "Hello Python!" << std::endl;
}
```

נוכל לייבא אותו לסקריפט פייתון והוא ישמר אוטומטית במרחב שם הגלובלי:

```
import cppyy
cppyy.include("hello.hpp")
cppyy.gbl.say_hello()
print("Hello C++!")
```

```
Hello Python!
Hello C++!
```

זהו רק קצה הקרחון של הספרייה. לעוד אינפורמציה עליה ניתן למצוא בדוקומנטציה שלה

PYPY AND NUMBA -

בנתיים ראינו דרכים לשפר את זמן ריצת התוכנית ע"י יבוא של מודולים חיצוניים או קודים משפות אחרות, אבל ישנה דרך נוספת לשפר את זמן הריצה של התוכנית –שימוש במפרש אחר.
במסמך הראשון בנושא פייתון ראינו כי פייתון משתמש במפרש שמתרגם את הקוד לשפה תחתונה שנקראת cpython (ולא להתבלבל עם cython) וכך הוא מריץ את הקוד במהירות.
pypy הוא סוג אחר של מפרש שמתרגם את הקוד בצורת jit(just in time) ומריץ את הקוד בממוצע פי ארבע יותר מהר.
חשוב לציין כי pypy לא תמיד יהיה הפתרון המהיר מבין השניים , היות והוא לא מזהה את כל סוגי המודולים הקיימים, למשל הוא לא תומך ב-numpy ובנוסף הוא מצריך להוריד מפרש דבר שהוא לא הכי סימפטי, ולמרות שאלה חסרונות רציניים , חשוב להכיר אותו, ואפילו גידו ואן רוסו(מייסד פייתון) אמר עליו:

" If you want your code to run faster, you should probably just use pypy".

לעוד פרטים בנוגע ל- pypy אפשר למצוא כאן.

ספרייה נוספת שמתרגמת את הקוד בצורת jit היא numba.
מבלי שנצטרך להשתמש בקומפיילר חיצוני או להוריד מפרש חדש, נוכל לקמפל את התוכנית דרך numba ובכך להריץ את הקוד בצורה משמעותית יותר מהירה.
numba עובדת עם decorators ובכך מאפשרת לשפר את הקוד מבלי כמעט לשנות אותו.
בשלב הראשון צריך להריץ את הפונקציה שעליה מפעילים את הדקורטור פעם אחת. בפעם הראשונה הפונקציה תרוץ לאט יותר כי המפרש צריך להבין אילו טיפוסי נתונים פועלים בפונקציה, אבל בפעם השנייה שנריץ את הפונקציה השינוי יהיה ניכר בהחלט.
numba עוצבה לעבוד עם מערכים של numpy ופונקציות, והיא עובדת מעולה גם עם ג'ופיטר.
numba טובה בעיקר עבור תוכניות חישוביות שעובדות עם הרבה מאוד לולאות.
התקנה:

```
pip install numba
```

כפי שאמרנו קודם numba מחשבת את האובייקטים שנכנסים למערכים ובכך היא מאיצה את מהירות הריצה של התוכנית, לכן חשוב לדאוג שהמערכים יהיו הומוגנים ככל האפשר, כמו כן עדיף לא להשתמש בפונקציות או אובייקטים של פייתון כמו enumerate (פונקציות generator) אלא בפונקציות כמו len() שנותנות ערך פעם אחת, כמשתמשים בקשטנים של הספרייה.
כמה מהקשטנים הנפוצים ביותר של הספרייה:

- @jit – jit הוא הקשטן הבסיסי והוא פועל בשני מצבי קומפילציה: nonpython ו-object .

ה-nonpython הוא מצב שמשמש כדי לקמפל את הפונקציה עם הקשטן ללא התערבות של המפרש של פייתון.
הוא הדרך היעילה לשימוש בnumba, אבל יש לפעמים אובייקטים שהקומפיילר של numba לא מכיר, למשל

אובייקטים של pandas, במקרים כאלה כדאי להשתמש במצב object שמקמפל רק שורות שהוא מזהה שניתן להגדיר אותן ללא המפרש של פייתון, אך סתם ככה תמיד נעדיף את המצב הראשון:

```
from numba import jit
import numpy as np
import time

x = np.arange(1000000).reshape(1000, 1000)

@jit(nopython=True)
def go_fast(a): # Function is compiled and runs in machine code
    trace = 0.0
    for i in range(a.shape[0]):
        trace += np.tanh(a[i, i])
    return a + trace

# DO NOT REPORT THIS... COMPILATION TIME IS INCLUDED IN THE EXECUTION TIME!
start = time.time()
go_fast(x)
end = time.time()
print("Elapsed (with compilation) = %s" % (end - start))

# NOW THE FUNCTION IS COMPILED, RE-TIME IT EXECUTING FROM CACHE
start = time.time()
go_fast(x)
end = time.time()
print("Elapsed (after compilation) = %s" % (end - start))
```

- @njit- כמו jit רק שהמצב הדיפולטיבי שלו הוא nonpython , ולכן לא צריך להגדיר פרמטר nonpython.
- @vectorize – קשטן המתלבש על פונקציה סקלרית כלשהי, והופך אותה לפונקציה וקטורית ('פונקציה אוניברסלית'). יש להגדיר את סוגי הנתונים המועברים בקלט והמוחזרים ממנה, מהטיפוסים הקיימים במאגר של numba. למשל:

```
from numba import vectorize, float64

@vectorize([float64(float64, float64)])
def f(x, y):
    return x + y
```

כאן אנחנו מגדירים פונקציה המקבלת שני וקטורים של משתנים ממשיים עם 64 סיביות (float64), ומחזירה וקטור של float64.

אפשר להעביר גם כמה סוגי משתנים אופציונליים עבור אותה פונקציה. יש רק לשים לב שככל שהמשתנה יותר משמעותי (יותר 'חלש') כך צריך לשים אותו קודם, לכן אם למשל נגדיר float64 לפני int64 הפונקציה תמיד תקבל float64, לא משנה מה הארגומנטים שנשלחו:

```
@vectorize([int32(int32, int32),
            int64(int64, int64),
            float32(float32, float32),
```

```
            float64(float64, float64)])
def f2(x, y):
    return x + y
```

שמרנו את שתי הפונקציות בקובץ vectorized_numba.py, ועכשיו אם נריץ את הפונקציה נראה שהיא רצה גם על מערכים ולא רק על מספרים בודדים (המערכים היעילים של numpy):

```
>>> import vectorize_numba
>>> vectorize_numba.f2([ 0,  2,  4,  6,  8, 10],[ 0,  2,  4,  6,  8, 10])
array([ 0,  4,  8, 12, 16, 20])
```

מתי numba יותר מהירה?
כפי שאמרנו קודם, בהרצה הראשונה של numba על פונקציות קטנות הפונקציה דווקא תיקח יותר זמן מהרצה דרך המפרש, כי היא מצריכה את numba להכיר את טיפוסי הנתונים בפונקציה, אבל בפעם השנייה יורגש ההבדל.

בנוסף בפונקציות עם לולאות ארוכות וטיפוסים מוגדרים היטב יורגש ממש הפער לטובת numba.
לצורך הדוגמא השתמשנו בקוד הבא:

```
@my_timer
@njit
def  f(x_range,y_range,z_range):
    lst = []
    for x in range(x_range):
        for y in range(y_range):
            for z in range(z_range):
                if (z + x + y)/10 == x:
                    lst.append(x)
    return lst
```

הרצנו את הקוד בשלושה סיבובים , פעם בלי הקשטן ופעם איתו.
בסיבוב הראשון הרצנו עם הארגומנטים 10,10,10, בסבוב השני עם הארגומנטים 10,1000,1000 ובשלישי עם הארגומנטים 10,1000,10000 ובדקנו מה יהיו התוצאות כאשר נריץ את הפונקציה עם הדקורטור njit ובלעדיו.
לצורך ההשוואה njit_f היא הפונקציה עם הדקורטור:

```
x = 10 , y = 10 , z= 10
f ran in: 0.0 sec
njit_f ran in: 0.38302159309387207 sec

x = 10 , y = 1000 , z= 1000
f ran in: 1.6180927753448486 sec
njit_f ran in: 0.021001100540161133 sec

x = 10 , y = 1000 , z= 10000
f ran in: 16.720956563949585 sec
njit_f ran in: 0.1910109519958496 sec
```

לוקח פי ~ 87.53... יותר זמן למפרש להריץ את הקוד משלוקח ל-numpa כאשר הוא רץ בלולאה של מאה מיליון איטרציות.
יש עוד הרבה מה להרחיב בנוגע לnumba לכן נמליץ להסתכל בדוקומנטציה של הספרייה באתר.