

Practical Python Performance Optimization

או: איך גורמים לנחש להיות יותר מהיר מצ'יטה

מאת יואב לוי

הקדמה

אם תשאלו כל מהנדס תוכנה מה הבעיה עם שפת פייתון הוא יגיד לכם "איטיות". הסברה המקובלת היא ששפת פייתון נועדה לאפשר כתיבת קוד מהירה ולא ריצת קוד מהירה, ויש בזה הרבה אמת. פייתון נחשבת כשפה הקלה ביותר ללימוד ובשנים האחרונות קצב השימוש בה גובר. כבר כיום פלטפורמות ענק שפועלות בסקאלה תעשייתית עולמית כמו גוגל, יוטיוב, אינסטגרם, נטפליקס, פינטרסט ועוד [מריצות קוד פייתון](#) מאחורי הקלעים.

במאמר זה נלמד על מדדי ביצועים של קוד, מהו פרופיל, מהי אופטימיזציה של קוד, נסקור חלק ממגוון כלי ה-Open Source הזמינים לנו לביצוע אופטימיזציה של קוד פייתון ונראה כיצד ניתן לשפר בעזרתם את הביצועים פי **אלפי אחוזים** בהיבטי זמן ריצה. נשים דגש על הפרקטיות של התהליך בהיבטי המאמץ ומורכבות השימוש בכלים לעומת התועלת שנפיק מהם וגם נבין מתי לא נרצה לבצע אופטימיזציה מסוג זה.

המאמר אינו דורש מומחיות בפייתון ומיועד לכל מפתח שרוצה לשפר את ביצועי הקוד שלו. בסוף המאמר נציג השוואה מסכמת של הכלים שבדקנו. מפתח קצר בזמן שרוצה לנסות את מזלו יוכל לדלג על החלקים התיאורטיים בתחילת המאמר ישר אל לוח התוצאות כדי להחליט באיזה כלי שווה לו לבחור, ונספק גם את קוד המקור לשימוש בכל הכלים שסקרנו.



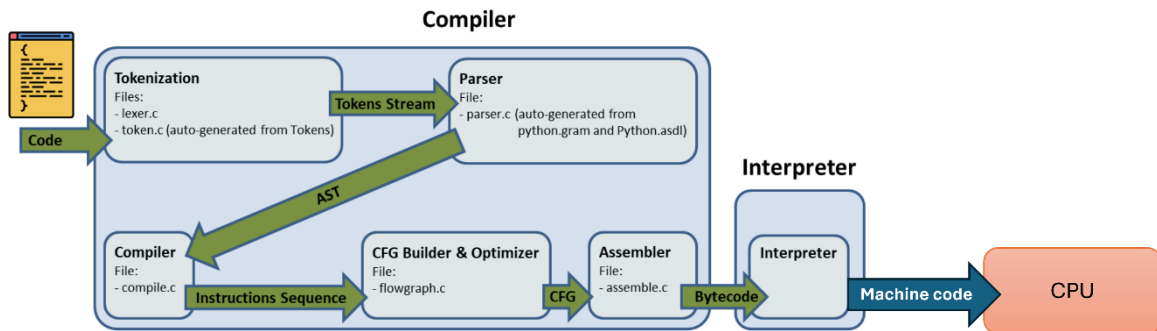
מבוא לביצועים ושפת פייתון

קיימים היבטים רבים ושונים בנוגע לפיתוח בשפת פייתון (תאימות לאחור, תאימות לפני, תאימות למערכות הפעלה ולחומרה ייעודית, אבטחת מידע, הנדסה לאחור, קהילת הקוד הפתוח, שרשרת אספקה, תלויות, design patterns וכו') אך במאמר זה נתמקד בהיבט צר יחסית - **הביצועים של השפה**, ובפרט ביצועים מסוג זמן ריצה.

כמובן שבפיתוח פרויקטי תוכנה גדולים צריך להביא בחשבון את כלל השיקולים, לכן חשוב להבין שמאמר זה לא נועד להיות מדריך לפיתוח בשפת פייתון אלא כלי עזר לשיפור ביצועים של הקוד לאחר שהוא נכתב ונבדק.

נסביר מאוד בקצרה איך עובדת שפת פייתון על מנת שנבין את ההבדלים בינה לבין שפות עיליות אחרות בעלות ביצועים טובים יותר.

על מנת לקבל רקע תיאורטי נתבונן באיור הבא:

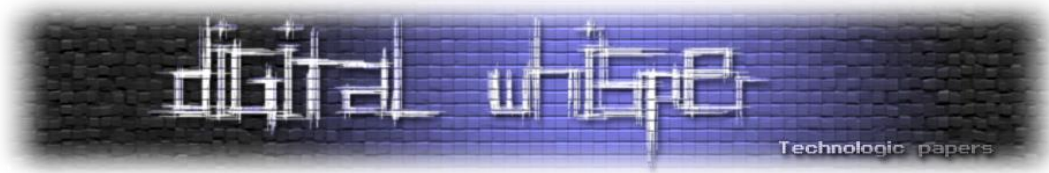


חלקו השמאלי נלקח באדיבות מסדרת המאמרים "[Practical Python Internals](#)" של אלי קסקי בגיליונות ה-159 וה-160. ובאיזה שפה כתובים הקומפיילר וה-interpreter של פייתון? התשובה היא בפייתון ובשפת C (כאשר החלקים הליבתיים כתובים ב-C) ולכן כאשר מדברים על המימושים השונים של שפת פייתון מתייחסים לפרויקט הרשמי בשם CPython.

כפי שניתן לראות באיור, פייתון היא שפת סקריפט שתהליך הריצה שלה כולל 2 שלבים עיקריים:

- **קומפילציה:** בשלב זה מתקבל קוד מקור בשפת פייתון ועובר 5 תתי שלבים שונים של עיבוד כאשר בסוף התהליך מתקבל קובץ פייתון מקומפל בפורמט pyc. ובו פקודות ה-bytecode.
- **אינטרפטציה:** תרגום פקודות ה-bytecode שהתקבלו מהשלב הקודם לשפת המכונה (machine code) שעליה רץ ה-`interpreter`.

קל להבין מדוע יש תקורה רצינית כתוצאה מכך - הקומפיילר וה-`interpreter` גם הם תהליכים שרצים בעצמם על ה-CPU וצורכים זמן ומשאבים כך שכל פקודה אטומית בשפה עילית מצריכה ביצוע מספר רב של פקודות מקדימות מצד הקומפיילר וה-`interpreter` טרם ביצועה במעבד.



זה המקום לציין שאנחנו לא הראשונים (וכנראה גם לא האחרונים) שרוצים לבצע אופטימיזציה לשפת פייתון, וגם קהילת המפתחים של הפרויקט הרשמי מציינת מידי changelog שיפורי ביצועים כאשר חלקם משמעותיים מאוד עד כדי כך ששימוש בגרסת פייתון מתקדמת יותר יכול להקטין את משך הריצה של אותו הקוד בעשרות אחוזים.

מבוא לאופטימיזציה של תוכנה

אופטימיזציה של תוכנה, כמו אופטימיזציה באופן כללי, היא נושא סבוך שדורש בסיס תיאורטי רחב וניסיון קודם. לכן מאמר זה **לא יעסוק** באופטימיזציה של תוכנה באופן כללי אלא רק בהיבט מסוים של אופטימיזציה של קוד פייתון.

רמות האופטימיזציה האפשריות לתוכנה כוללות בין השאר אופטימיזציה ברמת האוטומציה, ברמת החומרה, ברמת האלגוריתם, ברמת קוד המקור, ברמת הקומפיילר (Link-time) וברמת הבינארי (Post-Link). מכיוון שפייתון אינה שפה מקומפלט (במובן הקלאסי) ההיבט של אופטימיזציה שאנו נתמקד בו במאמר זה שוכן רק בין שתי רמות האופטימיזציה האחרונות שהזכרנו.

במאמר זה נתמקד בשיפור ביצועי הריצה של קוד פייתון ללא שינוי הלוגיקה שלו, לאחר שסיימנו לכתוב ולבדוק אותו. נהוג לבצע אופטימיזציה מסוג זה רק לאחר שכל האופטימיזציות האפשריות שקודמות לה בהיררכיה שהזכרנו כבר בוצעו, כי היא דורשת מיומנות לא טריוויאלית וקשיים נוספים שנדבר עליהם.

למה אנחנו רוצים לעשות אופטימיזציה? כי התהליך הזה חוסך לנו זמן וכסף. אמנם כיום נוח מאוד לעולם הפיתוח להתעלם מביצועים גרועים כי החומרה משתפרת מהר יותר מאשר שהקוד נהיה איטי יותר, אבל [שבירת חוק מור](#) בשנים האחרונות מרמז שלתחום האופטימיזציה בתוכנה יהיה מקום משמעותי בעתיד.

בנושא זה אני ממליץ על [הרצאותיו](#) של פרופ' אמרי ברגר מאוניברסיטת מסצ'וסטס, שחוקר ביצועים של תוכנה ואפילו כתב [כלי](#) למדידת [ביצועים של שפת פייתון](#), אשר נזכיר בהמשך.

מדדי ביצועים

לפני שנגדיר מהי אופטימיזציה נגדיר קודם מהו מדד ביצועים (Performance Measure) ולמה הוא כל כך חשוב לתהליך האופטימיזציה. מדד ביצועים יכול להיות כל דבר בר-מדידה שקשור לקוד שכתבנו. לדוגמה:

- זמן הריצה של תוכנית
- כמות הזיכרון המקסימלי הנצרך
- כמות הזיכרון המקסימלי X זמן הריצה
- ההספק החשמלי הנצרך ע"י המעבד במהלך ריצת התוכנית
- הזמן שלוקח למפתח לכתוב את הקוד

ועוד רבים נוספים, כתלות במקרה השימוש ומטרות התוכנית. פונקציית מטרה היא פונקציה של אחד או יותר מדדי ביצועים. תהליך האופטימיזציה נקבע ביחס לפונקציית מטרה מסוימת ומטרתו להביא את פונקציית המטרה **למקסימום או מינימום** בהינתנם של האילוצים הקיימים (אם קיימים).

פה המקום לציין שפונקציות מטרה יכולות להיות מסובכות מידי לאנליזה מתמטית כך שלא ניתן יהיה לדעת מהם המקסימום/מינימום שלהן, ולכן נרחיב את ההגדרה ונגיד שמטרת תהליך האופטימיזציה הוא להקטין או להגדיל את פונקציית המטרה ככל שידינו משגת.

חשוב להגדיר את פונקציית המטרה טרם תחילת תהליך המיטוב כי **ניסיון לביצוע אופטימיזציה ללא פונקציית מטרה ברורה היא כמו ירייה בחשיכה** - רק לפעמים נפגע, וגם אם נפגע לא נדע שפגענו. כדי להציג את התהליך בצורה ברורה ככל האפשר במסגרת מאמר זה נבחר בפונקציית מטרה ששווה בדיוק למדד זמן הריצה של תוכנית מסוימת שמקבלת קלט קבוע מסוים ונגדיר שאנחנו מעוניינים להביא את פונקציית המטרה למינימום.

איך מודדים? מה הוא פרופיל?

כדי לדעת איך נכון לבצע אופטימיזציה לתוכנה, חשוב לדעת מה החלקים בה אשר מהווים את המעמסה הגדולה ביותר על הביצועים (פונקציית המטרה) שהגדרנו. **פרופיל** הוא פילוח של הביצועים על פי פונקציות או שורות בקוד. לדוגמה, אם הגדרנו שפונקציית המטרה שלנו היא זמן הריצה של הקוד, פרופיל מפורט של הקוד יכלול הצגה של זמן הריצה המצטבר הכולל של כל שורה בקוד. המידע הזה יעזור לנו להבין מאיפה נכון להתחיל את תהליך האופטימיזציה כך שנטפל קודם בשורות הקוד עם הביצועים הגרועים ביותר.

לשפת פייתון קיימים עשרות פרופילרים שונים אך המקיף, המדויק והמהיר ביותר מביניהם הוא [Scalene](https://www.scalene.com/), אשר נכתב ע"י פרופ' אמרי ברגר ומפתחים נוספים בקוד פתוח.

מי שקרא את המבוא זוכר שבמאמר זה לא נלמד איך לבצע אופטימיזציה ברמת קוד המקור, אז למה הזכרתי בכלל מה זה פרופיל ואיך מומלץ לבצע אותו?

א. כדי להביא למודעות הקוראים שיש אפשרות לבצע אופטימיזציות קוד מקור, ושזה חלק בלתי נפרד מתהליך האופטימיזציה המלא.

ב. כדי לחדד את ההבדל בין רמה זאת של אופטימיזציה לבין מה שנלמד במאמר. מכיוון שמאמר זה נבצע רק אופטימיזציה ברמות הקומפילר והבינארי, אין לנו צורך בפרופילר אלא רק בדרך נוחה למדוד את זמן הריצה הכולל. אמנם טרם למדנו איך לבצע אופטימיזציה אבל בידינו היכולת לקבוע האם גרסת תוכנה מסוימת היא לא אופטימלית - אם מדדנו שגרסה אחרת בעלת פונקציונליות זהה מביאה את זמן הריצה לערך נמוך יותר. קיימים כלים רבים למדידת זמן ריצה של קוד פייתון, כאשר רובם הם בעצמם ספריות פייתון (לדוגמה `timeit` המוכר והפופולרי). אנחנו נבחר דווקא בכלי מדידה שאינו כתוב בשפת פייתון כדי לנתק את התלות במימוש הרשמי של השפה או בספרייה מסוימת בו.

הכלי שנבחר הוא [Hyperfine](#) אשר כתוב בשפת Rust, ומאפשר לנו למדוד באופן אמין ובלתי תלוי כל פקודת הרצה שנגדיר לו. הכלי מודד את זמן הריצה הממוצע של מספר איטרציות עוקבות ונותן כפלט את הממוצע והשוונות שלהן:

```
▶ hyperfine --warmup 3 'fd -e jpg -uu' 'find -iname "*.jpg"'
Benchmark #1: fd -e jpg -uu
Time (mean ± σ): 329.5 ms ± 1.9 ms [User: 1.019 s, System: 1.433 s]
Range (min ... max): 326.6 ms ... 333.6 ms 10 runs

Benchmark #2: find -iname "*.jpg"
Time (mean ± σ): 1.253 s ± 0.016 s [User: 461.2 ms, System: 777.0 ms]
Range (min ... max): 1.233 s ... 1.278 s 10 runs

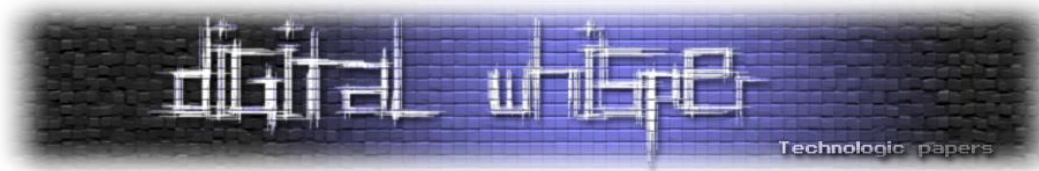
Summary
'fd -e jpg -uu' ran
3.80 ± 0.05 times faster than 'find -iname "*.jpg"'
▶
```

[דוגמה לשימוש בכלי Hyperfine, מתוך התיעוד הרשמי]

ממוצע זמן הריצה שהכלי יספק לנו בפלט מהווה את פונקציית המטרה אשר הגדרנו.

בחירת אמת המידה להשוואה (Benchmarking)

עכשיו כשאנחנו יודעים איך למדוד, עולה השאלה - מה נמדוד? קוד שמשמש להשוואת ביצועים נקרא `benchmark`, ואנחנו נבחר באחד מאתגר חישובית על מנת להדגיש את ההבדלים בין הכלים השונים ולהפוך את זמן ריצת הקוד שעוטף את פונקציית ה-`benchmark` לזניח (כמו לדוגמה טעינת ספריות באמצעות הפקודה `import`). לצורך ההשוואה נריץ את ה-`benchmarks` על מכונה וירטואלית בהפצת לינוקס Ubuntu 22.04 עם משאבים סטנדרטיים (2 ליבות ו-4GB RAM).



ניקח בעיה מתמטית קשה חישובית: בדיקת ראשוניות של מספר. נדאג שה-benchmark יהיה על בסיס [מימוש יעיל שלה](#) כדי לייטר את שלב האופטימיזציה של קוד המקור (שכאמור לא נעסוק בו במאמר זה) ובשפת פייתון טהורה (ללא שימוש בספריות חיצוניות) כדי שהקוד יהיה נתמך בכל הכלים שנבדוק.

```
def is_prime(n: int) -> bool:
    if n <= 3:
        return n > 1
    if (not (n % 2)) or (not (n % 3)):
        return False
    for i in range(5, int(n**0.5)+1, 6):
        if (not (n % i)) or (not (n % (i+2))):
            return False
    return True
```

נשאר לנו להגדיר מה יהיה הקלט שנזין לפונקציה, מן הסתם כדי להקשות על התוכנית נרצה קלט שהוא מספר ראשוני גדול מאוד כדי שהתוכנית תשלים את כל האיטרציות בלולאה ולא תסיים לרוץ מיד.

המספר הראשוני הלא טריוויאלי שנבחר יהיה:

8,225,092,069,056,351,469

נריץ את ה-benchmark עם CPython 3.10 ונמדוד באמצעות Hyperfine:

```
dylan@dylan-pc:~/Desktop/isPrime$ hyperfine ./bench_python
Benchmark 1: ./bench_python
Time (mean ± σ): 92.603 s ± 0.597 s [User: 92.237 s, System: 0.057 s]
Range (min ... max): 91.892 s ... 93.830 s 10 runs
```

זמן ההרצה הממוצע הינו כדקה וחצי וזה לא מעט בכלל אבל הגיוני בהינתן שהקוד יצטרך לבצע

$$\left\lfloor \frac{\sqrt{8225092069056351469}}{6} \right\rfloor = 477,990,355$$

איטרציות ובכל אחת מהן לבצע עד 2 חישובי שארית.

כלי (Just-in-Time) JIT נפוצים

קומפילציה מסוג JIT מתבצעת בזמן הריצה ומהותה הוא שמירת תוצרי הקומפילר וה-interpreter בזיכרון כך שכל קוד יקומפל ויתורגם רק פעם אחת ובכך ישופרו ביצועי זמן ריצה. חשוב להבין שכלים מסוג זה לא יתנו לנו יתרון בקוד שכל שורה בו רצה בקירוב רק פעם אחת. נשתמש בסוג זה של אופטימיזציה בקוד שמכיל לולאות או קריאות חוזרות לפונקציות במהלך הריצה.

Numba

.Numpy הכלי המוכר ביותר בהקשרי JIT בעל תמיכה מלאה בפיתון טהור ובספריית החישוב המדעי Numpy. השימוש בכלי מתבצע בתוך קוד המקור באמצעות ייבוא הספרייה והוספת decorator לפני הפונקציות אותן נרצה להאיץ. קיימים מספר פונקציות וארגומנטי קונפיגורציה לצורך כיוונון הביצועים ואנחנו נתמקד בפונקציה אחת וב-3 ארגומנטים שחשובים לפונקציית המטרה שהגדרנו:

```
import numba

@numba.njit(nopython=True, cache=True, nogil=True)
def is_prime(n: int) -> bool:
    if n <= 3:
        return n > 1
    if (not (n % 2)) or (not (n % 3)):
        return False
    for i in range(5, int(n**0.5)+1, 6):
        if (not (n % i)) or (not (n % (i+2))):
            return False
    return True

if __name__ == "__main__":
    is_prime(8225092069056351469)
```

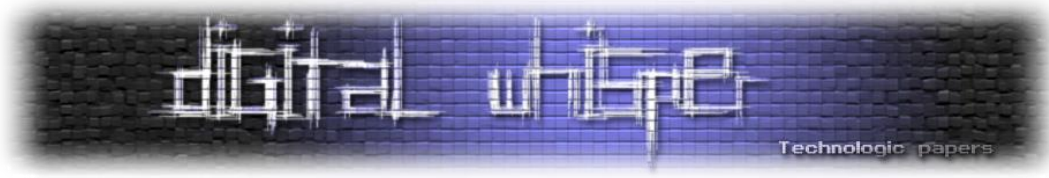
השתמשנו בפונקציית decorator בשם njit כך שמשמעות הארגומנט הראשון היא שאין צורך להתייחס למשתנים שבקוד בתור אובייקטים של פיתון (וכך נחסוך תקורה של הקצאות זיכרון מיותרות ו-Type checking בזמן הריצה), משמעות הארגומנט השני היא שימוש ב-caching על הדיסק על מנת לחסוך את שלב הקומפילציה (בפעמים הבאות שהקוד ירוץ) ומשמעות הארגומנט השלישי היא שאין צורך בשימוש ב-GIL (Global Interpreter Lock) של פיתון בזמן ריצת הקוד. שימו לב שבחירת ארגומנטים זו מועילה לפונקציית המטרה שהגדרנו אבל עבור פונקציות מטרה שונות יכול להיות שנרצה להשתמש בקונפיגורציות אחרות.

PyPy

פרויקט שהוא מימוש חליפי ל-CPython ומטרתו המוצהרת היא שיפור ביצועים. פרויקט זה הינו המוכר והפופולרי ביותר מבין קטגוריית ה-JIT. זוהי אינה ספרייה אלא קובץ הרצה שונה של Interpreter של פיתון והוא כולל בתוכו גם קומפיילר מסוג JIT. כל קוד פיתון טהור עד גרסה 3.10 כולל נתמך והתמיכה בספריות חיצוניות קיימת אך מידת שיפור הביצועים עבור קוד שמשמש בהן מוגבלת. ההרצה מתבצעת כמו שימוש רגיל ב-interpreter של פיתון.

Pyston

מימוש חליפי ל-CPython 3.8 בעל יכולות JIT כאשר המפתחים טוענים שניתן בעזרתו להשיג שיפור ביצועים בשיעור של 30% ללא צורך בשינוי הקוד הקיים.



כלי AoT (Ahead-of-Time) נפוצים

בקטגוריה זאת קיימים כלי אופטימיזציה שמטרתם היא להפוך את קוד המקור לבינארי שמקומפל במובן הקלאסי (ישירות לשפת מכונה). איך זה עובד? ממירים את קוד המקור בפיתון לקוד מקור בשפה מקומפלט (לרוב C או C++) ואז מקמפלים את הקוד המתקבל לשפת מכונה בעזרת קומפיילר מוכר (לרוב gcc) בתהליך שנקרא Cross Compilation. היתרון הוא ברור: לאחר הקימפול לשפת מכונה אין צורך יותר בקומפיילר וב-interpreter של פיתון, ויש לנו בינארי שיכול לרוץ ישירות על המעבד.

קטגוריה זו נחשבת למוצלחת יותר בהיבטי שיפור ביצועים אבל גם למוגבלת יותר בהיבטי תמיכה בספריות חיצוניות וקשה יותר לביצוע בהיבטי המומחיות הנדרשת לשינוי הקוד באופן שיתמוך בקימפול.

Cython

ה-Cross Compiler הנפוץ והמתוחזק ביותר לשפת פיתון, בעל תמיכה רחבה בספריות. בניגוד לכלים אחרים שראינו עד כה, על מנת לבצע Cross Compilation באמצעות Cython יש לכתוב קוד המקור מחדש כדי להכווין את הקומפיילר כיצד לתרגם את הקוד לקובץ הרצה יעיל יותר אבל פונקציונלי באותה מידה. בעבר הקומפיילר של Cython תמך רק בשפת Cython (שאינה מהווה קוד פיתון תקין) אך כיום ניתן להשתמש ב-Cython גם עבור קוד שכתוב ב-Pure Python עם שינויים קלים.

השינויים כוללים בעיקר type annotations של משתנים לוקליים. מכיוון שהשימוש ב-Pure Python קל יותר מאשר השימוש בשפת Cython, זאת הדוגמה אשר נראה במאמר (ובכל מקרה אין שינוי בביצועים בין שתי הגישות).

פונקציית ה-benchmark שלנו תראה כך לאחר הוספת ה-type annotations:

```
import cython

def is_prime(n: cython.ulonglong) -> bool:
    if n <= 3:
        return n > 1
    if (not (n % 2)) or (not (n % 3)):
        return False
    i: cython.ulonglong = 0
    for i in range(5, int(n**0.5)+1, 6):
        if (not (n % i)) or (not (n % (i+2))):
            return False
    return True
```

בעצם הוספנו רק הגדרת טיפוס לכל המשתנים הלוקליים מתוך הטיפוסים הנתמכים בספריית Cython. במקרה זה בגלל שהקלט לפונקציה צפוי להיות מספר גדול אז הגדרנו עבורו טיפוס שמייצג את הטיפוס long long unsigned int בשפת C.

Shed Skin

פרויקט ניסיוני למימוש קרוס-קומפיילר אוטומטי מפייתון לשפת ++C. בתיקיית הפרויקט ב-GitHub ישנן עשרות דוגמאות לתוכניות פייתון לדוגמה שהביצועים שלהן משתפרים פלאים בעזרת הכלי הזה גם מעיד שיש לכלי תאימות טובה עם ספריות פייתון פופולריות והוא מתוחזק באופן תדיר. בדומה ל-Cython ניתן ליצור בעזרת הכלי (.so) extension module שניתן לייבא ממנו פונקציות מתוך קוד פייתון רגיל.

נקמפל את קובץ ה-benchmark המקורי שלנו (utils.py) בעזרת הפקודות הבאות (שימו לב שעל מנת לתמוך במספרים גדולים מאוד נוסף את הדגל --long בעת הקומפילציה):

```
shedskin build --Long -e utils.py
```

ונקבל בתוך תיקיית build שנוצרה קובץ ספרייה תואם בשם utils.so אותו נייבא לצורך ה-benchmark בדומה לשימוש שהראנו ב-Cython.

Pythran

הכלי הכי מוכר בשיטת AOT אחרי Cython, ובניגוד אליו כמעט לא דורש התאמה ושינוי של הקוד טרם הקומפילציה. עבור כלי זה על הסקריפט שמתקבל כקלט להכיל פונקציות בלבד ויש צורך להדריך את הכלי אילו פונקציות לייצא באמצעות הוספת [שבנג](#) בראש הקובץ:

```
#pythran export is_prime(int)
```

הקמפול מתבצע ע"י הרצת הכלי עם ארגומנט יחיד שהוא הסקריפט שלנו:

```
~<pythran path>/pythran utils.py
```

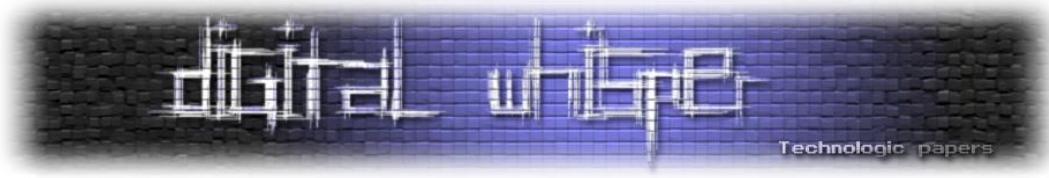
לאחר הפעלת הכלי יוצר extension module תואם לגרסת פייתון המותקנת על המחשב:



לוח תוצאות

נציג את תוצאות זמן הריצה שמדדנו באמצעות הכלי Hyperfine בטבלה:

is_prime(8225092069056351469)							
AOT			JIT			Vanilla	שיטה
Pythran	Shed Skin	Cython	Pyston	PyPy	Numba	CPython 3.10	כלי
7.12	7.7	5.728	73.45	9.49	8.6	92.6	זמן (שניות)
13.0	12.0	16.2	1.3	9.8	10.8	1	פקטור שיפור



כמה אנחנו רחוקים מהאופטימום האפשרי?

כל הפתרונות שהצגנו מערבים מן הסתם שימוש בשפת פייתון, אבל אם נרצה לבחון אובייקטיבית את ביצועי השפה (לאחר שימוש בכלי האופטימיזציה) ביחס לשפות תכנות אחרות?

הסטנדרט הלא רשמי לקוד שאמור לרוץ מהר הוא תוכנית שעברה קומפילציה בשפת C. ניקח מימוש זהה פונקציונלית לבדיקת הראשוניות בשפת C:

```
int IsPrime(unsigned long long int n)
{
    if (n == 2 || n == 3)
        return 1;
    if (n <= 1 || n % 2 == 0 || n % 3 == 0)
        return 0;
    for (unsigned long long int i = 5; i * i <= n; i += 6)
    {
        if (n % i == 0 || n % (i + 2) == 0)
            return 0;
    }
    return 1;
}

int main()
{
    IsPrime(8225092069056351469);
}
```

נקמפל באמצעות הקומפילר gcc ונמדוד:

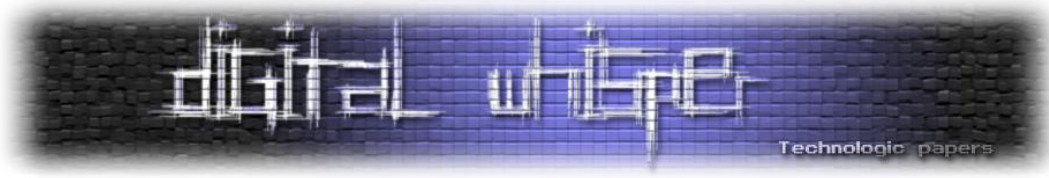
```
dylan@dylan-pc:~/Desktop/C_bench$ gcc is_prime.c -o is_prime
dylan@dylan-pc:~/Desktop/C_bench$ hyperfine ./is_prime
Benchmark 1: ./is_prime
Time (mean ± σ):      5.823 s ± 0.028 s    [User: 5.801 s, System: 0.005 s]
Range (min ... max):  5.767 s ... 5.859 s    10 runs
```

התוכנית אפילו מעט איטית יותר מהשימוש בכלי Cython. וזה מרמז שאנחנו בכלל לא רחוקים מהאופטימום האפשרי במקרה הזה.

מתי נשתמש בזה?

למרות שהפונקציה לבדיקת ראשוניות שבחרנו יחסית אופטימלית ברמת קוד המקור ומכילה אך ורק חישובים לוגיים ומתמטיים, עדיין הצלחנו לשפר את ביצועי זמן הריצה שלה פי 16 והראנו ש**נחש יכול להיות יותר מהיר מציטה** (אם יודעים באילו כלים להשתמש).

חלק מהכלים מצריכים ידע מקדים ושינויים בקוד וחלקם לא מצריכים מאמץ בכלל ועדיין מביאים לתוצאות מרשימות. עבור benchmarks מסובכים יותר שמערבים לא רק חישובים מתמטיים אלא גם שימוש במבני נתונים גדולים אנחנו עשויים לראות אפילו שיפור גדול יותר אבל היחס בין ביצועי הכלים השונים ייטה להישמר.



עם זאת צריך להיות מודעים לעובדה שבמקרים חריגים אנחנו עלולים לחוות ירידה בביצועים לאחר שימוש בחלק מהכלים. לכן לפני שאתם רצים למטב את הקוד שלכם חשוב שנלמד גם על המגבלות של הכלים שהצגנו.

מתי לא נשתמש בזה?

למרות שפייתון נוטה להיות איטית, יש מקרים בהם היא מאוד מהירה. מקרים כאלה בד"כ יכללו שימוש בעיקר בפונקציות פרימיטיביות של השפה אשר ממומשות מראש בשפת C למטרות מיטוב ביצועים.

נביא דוגמה נגדית לאלגוריתם בפייתון שקשה להשיג עבורו ביצועים טובים יותר בשפות אחרות, וקשה מאוד לשפר אותו אפילו לאחר שימוש בכלי האופטימיזציה שסקרנו. האלגוריתם הפעם יקבל כקלט מספר גדול כלשהו N ויחזיר בפלט קבוצה (set) של כל המספרים הראשוניים שקטנים או שווים ל-N.

קל להבין שמדובר בבעיה קשה יותר מאשר בדיקת ראשוניות של מספר יחיד: נדרש להחזיר קבוצה (בגודל שאינו ידוע מראש) של ראשוניים (שאינם ידועים מראש) וגם להקצות כמות זיכרון לא זניחה לשם כך. גם כאן, נבחר מימוש אידיאלי מבחינת זמן ריצה בשיטת [הנפה של ארטוסתנס](#):

```
def sieve_of_eratosthenes(n):
    sieve = set(range(2, n + 1))
    primes = []
    while sieve:
        prime = sieve.pop()
        primes.append(prime)
        sieve -= set(range(prime, n + 1, prime))
    return primes

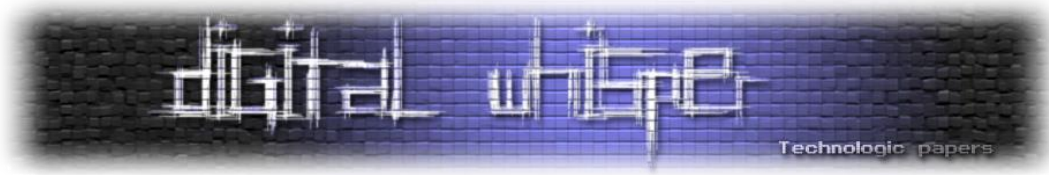
if __name__ == "__main__":
    N = 1000000
    prime_numbers = sieve_of_eratosthenes(N)
```

ואכן בהרצת הקוד באמצעות הפרויקט הרשמי (CPython) נקבל תוצאות לא רעות:

```
dylan@dylan-pc:~/Desktop/Primes$ hyperfine ./bench_python
Benchmark 1: ./bench_python
Time (mean ± σ): 704.9 ms ± 12.0 ms [User: 587.7 ms, System: 116.0 ms]
Range (min ... max): 691.1 ms ... 736.7 ms 10 runs
```

כעת נשתמש בכלי האופטימיזציה כמו שלמדנו ונשווה את התוצאות:

sieve_of_eratosthenes(1000000)							שיטה
AOT			JIT			Vanilla	
Pythran	Shed Skin	Cython	Pyston	PyPy	Numba	CPython 3.10	כלי
484.31	420.24	0.692	0.689	1.34	1.017	0.704	זמן (שניות)
0.0	0.0	1.0	1.0	0.5	0.7	1	פקטור שיפור



כמעט כל הכלים הביאו לביצועים גרועים (!) יותר מאשר המימוש הרשמי וחלקם אפילו היו בערך פי 600 יותר איטיים.

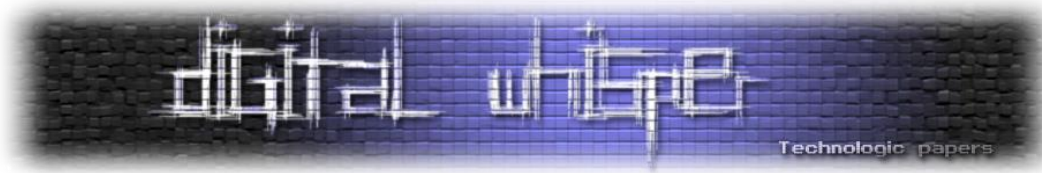
מה קורה פה? ממה נובע ההבדל? שימו לב שהפונקציה שמייצרת את הראשוניים כמעט ולא מכילה ביטויים אריתמטיים אלא בעיקר עבודה עם מבני נתונים כמו [טווחים](#) ו**קבוצות**, אשר ממומשים באמצעות פונקציות פרימיטיביות של שפת פייתון. [פונקציות פרימיטיביות](#) הן פונקציות ליבתיות בשפה (כמו [appendpop](#)) אשר אינן צריכות לעבור תרגום וממומשות בפרויקט הרשמי בצורה מיטבית. כלומר הן נכתבות מראש בשפת C, עוברות תהליך אופטימיזציה ידני (בשפת C) ע"י המפתחים ולבסוף מקומפלות לשפת מכונה. שימוש בכלי AOT שאינם מונחי טיפוסים (כמו Pythran ו-Shed Skin) במקרה כזה "יכריח" לתרגם אותן מחדש (לשפת C), ובאופן בלתי נמנע התרגום עצמו לא יהיה מיטבי כמו המקור (שכבר עבר אופטימיזציה).

אבחנה מעניינת היא שהכלי Pyston שהציג את השיפור הכי פחות משמעותי עבור ה-benchmark שבחרנו לבדיקת ראשוניות הפעם דווקא הציג את הביצועים הטובים ביותר (בהפרש קטן) מבין כל הכלים והצליח לשפר מעט את זמן הריצה. זה הגיוני אם נזכרים ש-Pyston נמנה על הכלים הפועלים בשיטת JIT אשר משתמשים באופן יעיל יותר ב-`interpreter` של פייתון מבלי לתרגם את הקוד בפועל לשפות אחרות.

עם זאת, הדוגמה הזו מלמדת שכאשר הכלים שלמדנו כמעט ולא מצליחים לשפר בפועל את זמן הריצה מדובר ברמז לכך שהקוד עצמו כבר במצב לא רע מבחינת ביצועים.

אחרית דבר

לאורך כתיבת המאמר גיליתי שלמרות שהמאמר נועד להיות מקיף ושלם, בפועל הוא רק מייצג מבוא לעולם של Link-Time/Post-Link Optimization בשפת פייתון ויש עוד הרבה ידע שלצערי לא הצלחתי להעביר במאמר במסגרת המאמצים לשמור אותו תמציתי, פרקטי ובעל ערך מיידי לקורא. אם קראתם את המאמר, ביצעתם את השלבים המתוארים בו ובכל זאת יש לכם צורך בשיפור נוסף בביצועים אני ממליץ לעיין ברשימת הקישורים השימושיים להעמקה בסוף המאמר.



סיכום

התחלנו את המאמר עם מטרה אמורפית של שיפור ביצועי קוד פייתון והעמקנו בהגדרות כדי לזקק את ההבנה של מה המשמעות האמיתית של אופטימיזציה וכיצד נגדיר אותה באופן פורמלי. למדנו מהם מדדי ביצועים ופונקציות מטרה וגם ראינו כיצד ניתן למדוד חלק מהם באופן מדויק ויעיל. סקרנו את 6 הכלים המובילים כיום לביצוע Link-Time Optimization על קוד פייתון, למדנו כיצד להשתמש בהם ומהם היתרונות והחסרונות של כל אחד.

בעזרת השימוש בכלים הצלחנו לשפר את ביצועי זמן הריצה של פונקציה לבדיקת ראשוניות של מספר גדול מאוד פי יותר מ-1,600% וראינו כיצד בא לידי ביטוי הקשר בין מעורבות המתכנת בתהליך האופטימיזציה לבין השגת גבול היכולת הביצועית של הקוד. מעבר לכך, גילינו שהקוד המתקבל לאחר אופטימיזציה מצליח אפילו להשיג תוצאות מעט טובות יותר מאשר שפות low-level כמו C. לבסוף ראינו באילו מקרים לא נרצה לבצע אופטימיזציה באמצעות הכלים שלמדנו והבנו גם מהי הסיבה לכך.

על המחבר

יואב לוי בעל תואר ראשון בהנדסת מחשבים ותוכנה, תואר שני בהנדסת חשמל, ועוסק כיום במחקר סייבר אבטחתי בתחומי ה-Malware Analysis ו-Reverse Engineering. מתעניין בעולמות האופטימיזציה, סייבר, מודיעין וגיאו-פוליטיקה.

קישורים שימושיים

- [פריקט Cython](#)
- [פריקט Shed Skin](#)
- [פריקט Pythran](#)
- [פריקט Numba](#)
- [פריקט PyPy](#)
- [פריקט Pyston](#)
- [הרצאה עם פתרון מודרך לאופטימיזציה באמצעות Numba ו-Cython](#)
- [הרצאתו של פרופ' אמרי ברגר על ביצועי שפת פייתון ודרכי המדידה](#)
- [רטון הדרכה על שימוש מתקדם ב-Cython 3.0](#)

קוד מקור

<https://github.com/YoavLevi/Practical-Python-Performance-Optimization-Source-Code.git>