Practical Python Internals

חלק ב' - Bytecode and Objects

מאת אלי קסקי

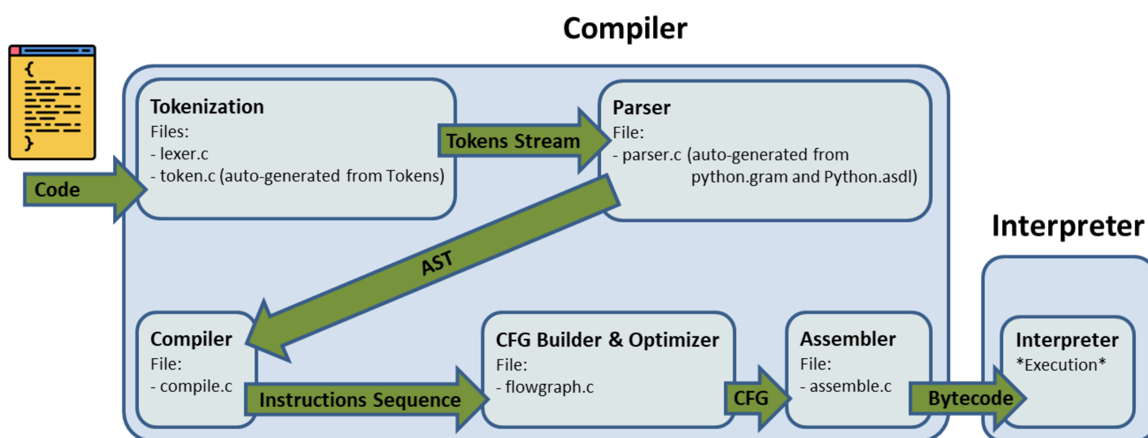
הקדמה

זהו המאמר השני בסדרת מאמרים שבה אני מציג אספקטים שונים במימוש של CPython.

במאמר הראשון למדנו על תהליך הקומפילציה של קוד Python לפקודות Bytecode. הוספנו אופרטור חדש לשפה שמאפשר להריץ את הקוד "++x". בחרנו אילו פקודות יווצרו כשמשתמשים בו, אך לא נגענו בהגדרת הפקודות האלו או באיך הן רצות בפועל על ידי ה-Interpreter.

במאמר זה נבין איך ה-Interpreter מפרש ומריץ את פקודות ה-Bytecode האלו, נוסיף פקודה חדשה משלנו, ונלמד על אובייקטים ב-CPython - איך הם מוגדרים, אופן השימוש בהם, ואיך הם נראים בזיכרון.

ניזכר בתרשים מהמאמר הקודם:



במסגרת המאמר הקודם התמקדנו בחלק של ה-Compiler, שמתואר במלבן השמאלי שבתרשים. במאמר זה נתמקד בחלק של ה-Interpreter, שמתואר במלבן הימני שבתרשים.

איך עובד ה-Interpreter

תפקידו של ה-Interpreter הוא להריץ את פקודות ה-Bytecode שאליהן התקמפל קוד ה-Python. הוא מבצע זאת בפונקציה _PyEval_EvalFrameDefault שבקובץ ceval.c. פונקציה זו מכילה הצהרת -switch case ענקית שמסתכלת על פקודת ה-Bytecode הנוכחית שיש להריץ, וקופצת ל-case המתאים לאותה הפקודה. להלן הקוד הרלוונטי מתוך הפונקציה הזו:

```
773         DISPATCH();
774 
775         {
776         /* Start instructions */
777     #if !USE_COMPUTED_GOTOS
778         dispatch_opcode:
779             switch (opcode)
780     #endif
781             {
782 
783     #include "generated_cases.c.h"
```

בפועל כל ה-case-ים שנמצאים בתוך הצהרת ה-switch-case נמצאים בקובץ generated_cases.c.h. קובץ זה מכיל את המימוש ב-C לכל פקודת Bytecode, והוא מג'ונרט מתוך קובץ בשם bytecodes.c שמכיל את הגדרות כל פקודות ה-Bytecode.

הקובץ bytecodes.c אמנם נראה כמו קובץ C רגיל, אך הוא כתוב בשפת DSL (Language Domain Specific) דמויית C, שנוצרה במיוחד ל-CPython לצורך הגדרות פקודות ה-Bytecode. קובץ זה לא מתקמפל ישירות כשמקמפלים את CPython. אלא, במהלך הקומפילציה של CPython רצים מספר סקריפטים (הכתובים ב-Python!) שמפרסרים את הקובץ ומג'נרטים ממנו קבצי c. ו-h. שונים. הקבצים המג'ונרטים הללו מכילים metadata על הפקודות, ואת קוד ה-Interpreter שרץ בפועל וממשש כל פקודה, והם אלו שעוברים קומפילציה לבסוף. ניתן לקרוא תיעוד על ה-snytax של שפת ה-DSL דמויית C הזו בקובץ interpreter_definition.md שבפרוייקט CPython הרשמי ב[קישור](#).

לצורך הדוגמא נסתכל למשל על הגדרת הפקודה LOAD_CONST מתוך הקובץ bytecodes.c, שבה דוחפים איבר למחסנית:

```
232         pure inst(LOAD_CONST, (-- value)) {
233             value = GETITEM(FRAME_CO_CONSTS, oparg);
234             Py_INCREF(value);
235         }
```

החלק הרלוונטי ב-syntax של שפת ה-DSL בהקשר למאמר זה הוא הסוגריים שבסוף בחתימת הפקודה. מה שמשמאל ל-"--" מייצג את ראש המחסנית לפני הרצת הפקודה, ומה שמימין מייצג את ראש המחסנית אחרי הרצת הפקודה.

בדוגמא זו, לפני הרצת הפקודה - לא משנה מה האיבר שבראש המחסנית ולכן אין התייחסות אליו. לאחר הרצת הפקודה המשתנה value יהיה ראש המחסנית, כלומר פקודה זו דוחפת את המשתנה הזה למחסנית. תוכן הפקודה עצמה מגדיר מה יהיה ערך האיבר value. במקרה זה הוא נלקח מ-tuple שנקרא co_consts (שנמצא באובייקט מסוג code שעליו נלמד במאמר הבא). האינדקס של האיבר ב-tuple הוא הארגומנט (oparg) של פקודת ה-Bytecode הזו. בנוסף לכך הפקודה מגדילה את שדה ה-ob_refcnt באובייקט שמתאים ל-value. נלמד על המשמעות לכך בהמשך מאמר זה.

כאמור זהו לא קוד C תקני אלא קוד DSL שעל בסיסו נוצר קוד C תקני. בקובץ generated_cases.c.h המג'ונרט מ-bytecodes.c נוצר קוד ה-C האמיתי. להלן הקוד שמג'ונרט עבור הפקודה LOAD_CONST:

```
3951            TARGET(LOAD_CONST) {
3952                frame->instr_ptr = next_instr;
3953                next_instr += 1;
3954                INSTRUCTION_STATS(LOAD_CONST);
3955                PyObject *value;
3956                value = GETITEM(FRAME_CO_CONSTS, oparg);
3957                Py_INCREF(value);
3958                stack_pointer[0] = value;
3959                stack_pointer += 1;
3960                DISPATCH();
3961            }
```

ניתן לראות ממש את שתי השורות של תוכן הפקודה שנלקחו מתוך bytecodes.c (שורות 3956-3957). שורות אלה נעטפו בקוד מסוים. החלק הראשון של הקוד העוטף נמצא לפני שורות תוכן הפקודה, ומכיל עדכון של המצביע לפקודה הבאה שיש להריץ, ועדכון סטטיסטיקה שקשורה להרצת הפקודה הנוכחית (מדובר באופטימיזציה ל-Interpreter שאזכיר במאמר הבא). החלק השני של הקוד העוטף נמצא אחרי שורות הפקודה, ומכיל את דחיפת המשתנה value למחסנית בפועל. השורה האחרונה בחלק זה היא המאקרו DISPATCH. מאקרו זה שולף את ה-opcode וה-oparg מתוך המצביע לפקודה הבאה, ולאחר מכן קופץ בחזרה לתחילת ה-switch-case. מבצע בפועל:

```
goto dispatch_opcode
```

כמובן שיש פקודות יותר מעניינות, שמשפיעות על ה-control flow (jump, conditional branch), מבצעות קריאה לפונקציה או זריקת exception, פקודות על אובייקטים ספציפיים (בניית רשימה, סט, מחרוזת, או פעולות אחרות עליהם), פקודות אופטימיזציה, ועוד.

לסיכום חלק זה, מה שצריך לזכור הוא שהקובץ bytecodes.c מגדיר את מימוש הפקודות ואת המחסנית לפני ואחרי הרצת כל פקודה.

הוספת פקודת Bytecode חדשה

במאמר זה נוסיף פקודת Bytecode חדשה, ונכתוב לה מימוש בקוד ה-Interpreter כדי שנוכל גם להריץ אותה. כל השינויים שנראה במאמר זה נמצאים ב-fork שלי ל-CPython בקישור. ישנו תיעוד רשמי שמכיל את רשימת הקבצים שיש לעדכן כשמוסיפים פקודת Bytecode חדשה בקישור.

הפקודה החדשה שנוסיף תבצע חישוב מסוים על האובייקט שנמצא בראש המחסנית. היא תעבוד בדומה לפקודת ה-Bytecode הקיימת UNARY_NEGATIVE שמתאימה לקוד "x-". להלן פקודות ה-Bytecode שאליהן מתקמפל קוד זה:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> dis.dis("-x")
  0           RESUME                   0

  1           LOAD_NAME                0 (x)
              UNARY_NEGATIVE
              RETURN_VALUE
>>>
```

ולהלן הגדרת הפקודה UNARY_NEGATIVE מתוך הקובץ bytecode.c:

```
300         inst(UNARY_NEGATIVE, (value -- res)) {
301             res = PyNumber_Negative(value);
302             DECREF_INPUTS();
303             ERROR_IF(res == NULL, error);
304         }
```

פקודה זו שולפת מהמחסנית איבר בשם value ומחשבת עליו חישוב מסוים, במקרה זה כפל המספר במינוס אחד. תוצאת החישוב נשמרת במשתנה res. לאחר מכן ישנו שימוש במאקרו DECREF_INPUTS, שמבצע הקטנה של שדה ה-ob_refcnt של האובייקט value, שכאמור נלמד על כך בהמשך המאמר. ישנה בדיקה שתוצאת החישוב אינה NULL, ולבסוף הפקודה דוחפת למחסנית את res.

הפקודה שלנו תפעל באופן דומה, אך במקום שהחישוב בה יהיה כפל המספר במינוס אחד, היא תחשב את פונקציית הקולץ (Collatz) של המספר:

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod{2}, \\ 3n+1 & \text{if } n \equiv 1 \pmod{2}. \end{cases}$$

בהתאם לכך, נקרא לפקודה החדשה UNARY_COLLATZ.

הוספת אופרטור חדש ל-Python

שלב זה הוא לא ממש חלק מהוספת הפקודה החדשה, אלא נועד רק כדי שנוכל לכתוב קוד Python שיתקמפל לפקודה החדשה שלנו. ראינו במפורט במאמר הקודם איך לעשות זאת ולכן לא ארחיב על כך יותר מדי. האופרטור שבחרתי לשם כך הוא התו $, כי זה בערך התו היחיד שפנוי, ולכן הקוד "$x" הוא זה שיתקמפל לפקודה החדשה שלנו.

בקצרה, יש להוסיף את התו החדש ל-Tokens ולעדכן את הקובץ Python.asdl כך שיכיל ערך חדש ל-Collatz ב-enum שמתאים ל-unaryop. לאחר מכן יש לעדכן את חוק הדקדוק factor בקובץ python.gram כדי שיתייחס לקוד שמכיל את התו החדש, ועבורו ייצור אובייקט expression עם ערך ה-enum החדש. לאחר שינויים אלה יש להריץ:

```
build.bat --regen
```

לבסוף יש לערוך את הפונקציה unaryop שבקובץ compile.c כך שתהיה התייחסות לערך ה-enum החדש, שעבורו פקודת ה-Bytecode שתיווצר תהיה הפקודה החדשה שלנו, UNARY_COLLATZ.

הוספת פקודת Bytecode חדשה

בקובץ bytecodes.c נגדיר פקודת Bytecode חדשה באותו האופן שמוגדרת הפקודה UNARY_NEGATIVE:

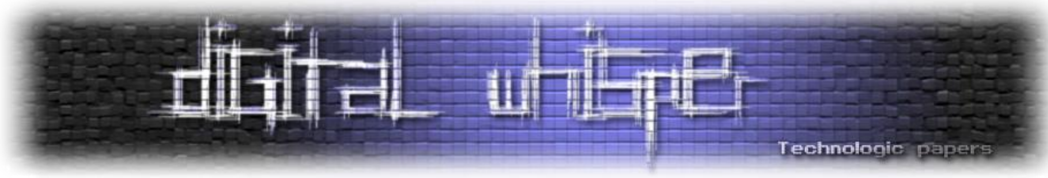
```
06        inst(UNARY_COLLATZ, (value -- res)) {
07            res = PyNumber_Collatz(value);
08            DECREF_INPUTS();
09            ERROR_IF(res == NULL, error);
10        }
```

את הפונקציה PyNumber_Collatz נגדיר בהמשך.

כדי לג'נרט מקובץ ה-bytecodes.c המעודכן את הקבצים השונים יש להריץ את כל הסקריפטים שבשמם יש את המילה generator ונמצאים בתיקיה Tools\cases_generator, ניתן לקרוא על כל אחד מהם בתיעוד בקישור.

בנוסף לכך, עם כל הוספה של פקודת Bytecode יש לעדכן שדה magic_number שנכתב לקבצי pyc. סביר להניח שכבר נתקלתם בקבצים אלה, הם נוצרים כשמריצים קוד שנמצא בתוך קובץ py.. קבצים אלה מכילים פקודות Bytecode של קוד Python מקומפל ונועדו לחסוך את תהליך הקומפילציה במידה והקובץ לא השתנה מאז הפעם האחרונה שהריצו אותו. לפי ההוראות של הוספת פקודת Bytecode חדשה, יש לעדכן שדה זה על ידי הגדלה של המשתנה MAGIC_NUMBER בקובץ _bootstrap_external.py. שינוי זה יגרום לכל קבצי ה-pyc. הקיימים להתקמפל מחדש.

בזאת סיימנו להוסיף את פקודת ה-Bytecode החדשה, אך עדיין לא מימשנו את הלוגיקה שלה - שאמורה להיות בפונקציה חדשה בשם PyNumber_Collatz. בשביל לעשות זאת נצטרך לעשות מעבר חד וללמוד קודם קצת על אובייקטים ב-CPython.



אובייקטים ב-CPython

ישנו המשפט המפורסם:

"Everything in Python is an object"

אבל מה זה אומר בפועל? כל דבר ב-Python הוא אובייקט. בין אם מדובר במספר, מחרוזת, פונקציה, או מודול. אפילו המבנים הפנימיים במימוש של CPython הם אובייקטים. לשם השוואה, בשפת C משתנה שמכיל מספר הוא פשוט איזור בזיכרון (על המחסנית או ב-heap) שמכיל בתים שתוכנם הוא ערך המספר. לעומת זאת ב-Python משתנה שמכיל מספר בפועל מכיל מצביע לאובייקט, והאובייקט מכיל בנוסף לערך המספר גם metadata נוסף עליו. האובייקטים האלו נוצרים ומשתחררים בצורה דינמית ב-heap, למעט מספר קטן של אובייקטים שמוגדרים בצורה סטטית.

הקונבנציה של אובייקטים ב-CPython

ה-struct הבסיסי של אובייקט ב-CPython נקרא PyObject, וכל האובייקטים האחרים ב-CPython יורשים מ-struct זה. להשתמש במונח "ירושה" בהקשר של קוד שכתוב בשפת C זה קצת מוזר. כידוע, C אינה שפת OOP ולכן אין בה באמת ירושה. כדי בכל זאת להשתמש באלמנטים של ירושה, ב-CPython ישנה קונבנציה שבה השדה הראשון בכל סוג אובייקט נקרא ob_base והוא מסוג PyObject, ושאר השדות בו משתנים בהתאם לסוג האובייקט. קונבנציה זו מאפשרת לגשת תמיד לשדה ה-ob_base שמכיל פרטים על האובייקט, בין היתר ה-type שלו, ובהתאם אליו לבצע לאובייקט cast לסוג האובייקט הספציפי המתאים לו.

יש בסך הכל שני שדות שנמצאים ב-PyObject, והם ob_refcnt ו-ob_type.

להלן הגדרתו:

```
typedef __int64 Py_ssize_t;
typedef struct _object PyObject;
struct _object {
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
};
```

שדה ה-ob_refcnt (Reference Count) מכיל מספר - כמות המצביעים לאובייקט זה. כשערך זה משתנה לאפס, זה אומר שאין בתוכנית עוד מצביעים לאובייקט, ואז מגיע ה-Garbage Collector ומשחרר את הזיכרון שהוקצה לאובייקט. יש לציין שקיימים אובייקטים שמכונים immortal, ועבורם שדה זה לא רלוונטי מכיוון שהם מתוכננים לא להשתחרר לעולם. דוגמאות לאובייקטים אלה הם None, True, False, וכל המספרים בטווח שבין 5- ל-256.

שדה ה-ob_type מכיל מצביע לאובייקט מסוג PyTypeObject. זהו אובייקט די חשוב וניתן לקרוא עליו בתיעוד הרשמי בקישור.

להלן תחילת הגדרתו:

```
147  struct _typeobject {
148      PyObject_VAR_HEAD
149      const char *tp_name; /* For printing, in format "<module>.<name>" */
150      Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */
151
152      /* Methods to implement standard operations */
153
154      destructor tp_dealloc;
155      Py_ssize_t tp_vectorcall_offset;
156      getattrfunc tp_getattr;
157      setattrfunc tp_setattr;
158      PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
159                                      or tp_reserved (Python 3) */
160      reprfunc tp_repr;
161
162      /* Method suites for standard classes */
163
164      PyNumberMethods *tp_as_number;
165      PySequenceMethods *tp_as_sequence;
166      PyMappingMethods *tp_as_mapping;
```

באובייקט PyTypeObject יש שימוש במאקרו PyObject_VAR_HEAD שמכיל אובייקט מסוג PyVarObject. משתמשים במאקרו זה באובייקטים בעלי גודל משתנה - למשל list, string ו-bytes. בתחילת אובייקטים מסוג זה, שדה ה-ob_base הוא מסוג PyVarObject (שמכיל בתוכו את PyObject). אובייקטים אלה מכילים בתחילתם בנוסף לשני השדות הסטנדרטיים גם שדה בשם ob_size שערכו הוא מספר הפריטים בחלק המשתנה של האובייקט. ניתן לקרוא עוד על PyObject, PyVarObject, ושדות אלה בתיעוד שבקובץ object.h. להלן הגדרת PyVarObject:

```
#define PyObject_VAR_HEAD      PyVarObject ob_base;

typedef struct {
    PyObject ob_base;
    Py_ssize_t ob_size; /* Number of items in variable part */
} PyVarObject;

typedef struct _typeobject PyTypeObject;
```

נחזור בחזרה להסתכל על האובייקט PyTypeObject. לאחר השדות הסטנדרטיים שבו, הוא מכיל שדה tp_name, שהוא מצביע למחרוזת שהיא שם סוג האובייקט. החלק העיקרי באובייקט הוא רשימה ארוכה של מצביעים לפונקציות שמממשות את הפונקציונליות של אותו סוג אובייקט. ניתן לקרוא על כל סוגי הפונקציות בתיעוד של Type Objects ב[קישור](). אובייקטי ה-PyTypeObject השונים מוגדרים בצורה סטטית בקוד של CPython. כל אובייקט PyTypeObject מוגדר בקובץ שמתאים לאובייקט שממש אותו. למשל האובייקט ה-PyTypeObject שמתאים למשתנה מסוג bool, מוגדר בקובץ boolobject.c, שבו מוגדר האובייקט שמתאים ל-bool.

בנוסף לפונקציות האלו, ישנם מצביעים לטבלאות פונקציות עם מכנה משותף, למשל הטבלה tp_as_number שמסוג [PyNumberMethods]() (שורה 164 בתמונה) ומכילה מצביעים לפונקציות שעוסקות בפעולות על מספרים.

טבלה זו מכילה 62 מצביעים לפונקציות, להלן חלק מהם:

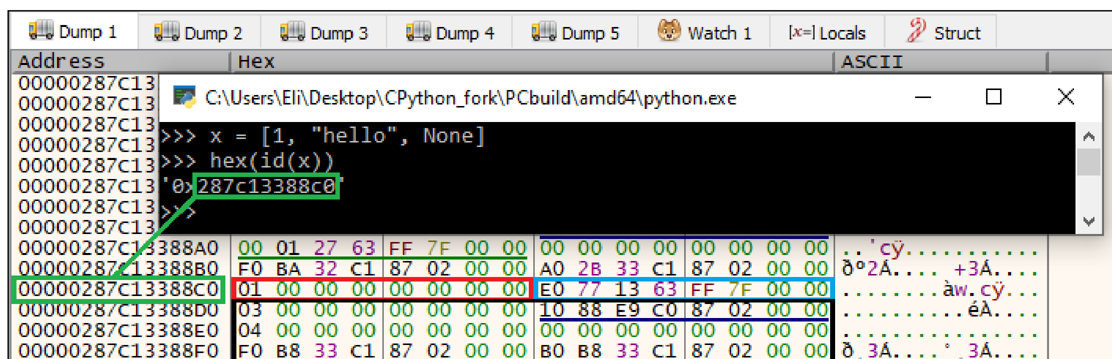
```
61  typedef struct {
62      binaryfunc nb_add;
63      binaryfunc nb_subtract;
64      binaryfunc nb_multiply;
65      binaryfunc nb_remainder;
66      binaryfunc nb_divmod;
67      ternaryfunc nb_power;
68      unaryfunc nb_negative;
69      ...
70  } PyNumberMethods;
```

משמעות הדבר היא שאם בסוג אובייקט מסוים קיימת למשל הפונקציה nb_add, אז ניתן להפעיל על אובייקט מסוג זה את האופרטור הבינארי +.

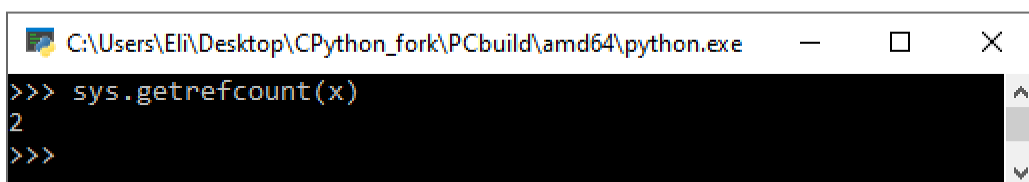
אובייקטים בזיכרון

ראינו איך אובייקטים מוגדרים בקוד המקור של CPython, אבל לדעתי מאוד עוזר להבין כשמסתכלים על איך האובייקטים נראים בזיכרון בפועל. נפתח את python.exe בדיבגר כלשהו, למשל x64dbg, ונסתכל על הזיכרון של התהליך. כדי למצוא בזיכרון בדיוק את האובייקט שאנחנו רוצים, ניתן להשתמש בפונקציה id ב-Python. זוהי פונקציה מובנית בשפה שמחזירה Unique Identifier של אובייקט. ספציפית במימוש של CPython פונקציה זו מחזירה את כתובת האובייקט בזיכרון.

לצורך הדוגמה נסתכל על אובייקט מסוג list:



שני השדות הראשונים הם הסטנדרטיים - באדום זהו ה-ob_refcnt, שבמקרה שלנו הוא 1. אם נמחוק את האובייקט x, או נכניס לתוכו ערך אחר, שדה ה-ob_refcnt באובייקט זה יקטן באחד ויגיע לערך 0, מה שיגרום לשחרור אובייקט זה מהזיכרון. ניתן גם לקרוא את שדה ה-ob_refcnt על ידי שימוש בפונקציה sys.getrefcount:



כפי שכתוב בתיעוד הפונקציה, בדרך כלל ערך החזרה של הפונקציה גדול ב-1 מהערך האמיתי של ob_refcnt, בגלל שמתווסף מצביע לאובייקט זה בקריאה עצמה לפונקציה.

השדה הסטנדרטי השני, בכחול, הוא ה-ob_type, מצביע לאובייקט ה-PyTypeObject שמכיל פרטים על סוג האובייקט. נעקוב אחרי המצביע ob_type כדי לראות פרטים על סוג האובייקט x:

Address	Hex	ASCII
00007FFF631377E0	FF FF FF FF 00 00 00 00 B0 EA 13 63 FF 7F 00 00	ÿÿÿÿ....°ê.cÿ...
00007FFF631377F0	00 00 00 00 00 00 00 00 AC D1 08 63 FF 7F 00 00	........¬Ñ.cÿ...
00007FFF63137800	28 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	(...............
00007FFF63137810	50 10 C9 62 FF 7F 00 00 00 00 00 00 00 00 00 00	P.Ébÿ...........
00007FFF63137820	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00007FFF63137830	00 00 00 00 00 00 00 00 70 12 C9 62 FF 7F 00 00	........p.Ébÿ...
00007FFF63137840	00 00 00 00 00 00 00 00 20 7B 13 63 FF 7F 00 00	.... {.cÿ...
00007FFF63137850	A8 42 13 63 FF 7F 00 00 B0 D5 CA 62 FF 7F 00 00	¨B.cÿ...°ÕÊbÿ...
00007FFF63137860	00 00 00 00 00 00 00 00 F0 A5 CD 62 FF 7F 00 00	........ð¥Íbÿ...

הגענו לאובייקט ה-PyTypeObject שמתאים ל-list, סוג האובייקט x. השדה שנסתכל עליו הוא tp_name, בצבע אדום, שמכיל מצביע למחרוזת שמתארת את סוג האובייקט:

Address	Hex	ASCII
00007FFF6308D1AC	6C 69 73 74 00 00 00 00 00 00 00 00 68 65 61 70	list........heap
00007FFF6308D1BC	70 6F 70 00 68 65 61 70 72 65 70 6C 61 63 65 00	pop.heapreplace.
00007FFF6308D1CC	00 00 00 00 68 65 61 70 70 75 73 68 70 6F 70 00	....heappushpop.
00007FFF6308D1DC	00 00 00 00 68 65 61 70 69 66 79 00 5F 68 65 61	....heapify._hea
00007FFF6308D1EC	70 70 6F 70 5F 6D 61 78 00 00 00 00 5F 68 65 61	ppop_max...._hea
00007FFF6308D1FC	70 72 65 70 6C 61 63 65 5F 6D 61 78 00 00 00 00	preplace_max....

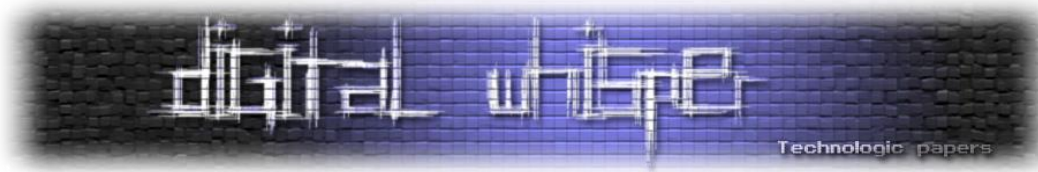
בנוסף למצביע למחרוזת זו ישנם בהמשך כל המצביעים לפונקציות השונות שהאובייקט x תומך בהן, ולטבלאות הפונקציות הנוספות ששייכות לסוג האובייקט.

מכיון שאובייקט ה-PyTypeObject שמתאים ל-list הוא אובייקט בפני עצמו, גם לו יש שדות סטנדרטיים של ob_refcnt ו-ob_type. אם נעקוב אחרי ה-ob_type שלו, בצבע סגול, נגיע ל-PyTypeObject שמתאים ל-type:

Address	Hex	ASCII
00007FFF6313EAB0	FF FF FF FF 00 00 00 00 B0 EA 13 63 FF 7F 00 00	ÿÿÿÿ....°ê.cÿ...
00007FFF6313EAC0	00 00 00 00 00 00 00 00 B4 C3 0A 63 FF 7F 00 00	........´Ã.cÿ...
00007FFF6313EAD0	A0 03 00 00 00 00 00 00 28 00 00 00 00 00 00 00	.......(.......
00007FFF6313EAE0	A0 9A CD 62 FF 7F 00 00 90 01 00 00 00 00 00 00	.Íbÿ...........
00007FFF6313EAF0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00007FFF6313EB00	00 00 00 00 00 00 00 00 40 3F CD 62 FF 7F 00 00	........@?Íbÿ...
00007FFF6313EB10	30 E9 13 63 FF 7F 00 00 00 00 00 00 00 00 00 00	0é.cÿ...........
00007FFF6313EB20	00 00 00 00 00 00 00 00 80 BE CD 62 FF 7F 00 00	.........¾Íbÿ...

אפשר לראות זאת בעזרת ההבחנה ששדה ה-ob_type מכיל מצביע בחזרה לאובייקט עצמו, וגם עם מעקב אחרי שדה ה-tp_name שלו, באדום:

Address	Hex	ASCII
00007FFF630AC3B4	74 79 70 65 00 00 00 00 00 00 00 00 77 65 61 6B	type........weak
00007FFF630AC3C4	6C 69 73 74 20 6F 66 66 73 65 74 20 25 64 20 69	list offset %d i
00007FFF630AC3D4	73 20 6F 75 74 20 6F 66 20 62 6F 75 6E 64 73 20	s out of bounds
00007FFF630AC3E4	66 6F 72 20 74 79 70 65 20 27 25 73 27 20 28 74	for type '%s' (t
00007FFF630AC3F4	70 5F 62 61 73 69 63 73 69 7A 65 20 3D 20 25 64	p_basicsize = %d
00007FFF630AC404	29 00 00 00 00 00 00 00 00 00 00 00 74 70 5F 62	)...........tp_b
00007FFF630AC414	61 73 69 63 73 69 7A 65 20 66 6F 72 20 74 79 70	asicsize for typ
00007FFF630AC424	65 20 27 25 73 27 20 28 25 64 29 20 69 73 20 74	e '%s' (%d) is t



אם נחזור חזרה לייצוג בזיכרון של האובייקט x עצמו, נראה שבשחור ישנם השדות הספציפיים שקיימים באובייקט מסוג list. מבלי לקרוא את קוד המקור של מימוש אובייקט הרשימה, ניתן לנחש שקיימים בו שדות של:

- מערך מצביעים לאובייקטים שנמצאים ברשימה.
- מספר המייצג את גודל המערך המוקצה לרשימה.
- מספר שמייצג את כמות האיברים בפועל ברשימה.

אנחנו אכן רואים את המספר 3, מצביע כלשהו, ואת המספר 4, סביר להניח שאלה השדות שניחשנו שקיימים. אם נעקוב אחרי המצביע נראה שהוא מכיל מערך של מצביעים:

```
Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1 | [x=] Locals | Struct
Address           Hex                                              ASCII
00000287C0E98810  80 FB 26 63 FF 7F 00 00 D0 B8 E7 C0 87 02 00 00  .û&cÿ...Ð¸çÀ....
00000287C0E98820  00 9C 13 63 FF 7F 00 00 90 EF 17 01 1F 0A 00 00  ...cÿ....ï......
00000287C0E98830  01 00 00 00 00 00 00 00 10 80 13 63 FF 7F 00 00  ...........cÿ...
00000287C0E98840  08 00 00 00 00 00 00 00 00 00 10 00 87 02 00 00  ................
00000287C0E98850  01 00 00 00 00 00 00 00 10 80 13 63 FF 7F 00 00  ...........cÿ...
00000287C0E98860  10 00 00 00 00 00 00 00 C0 9F 11 01 1F 0A 00 00  ........À.......
00000287C0E98870  02 00 00 00 00 00 00 00 10 80 13 63 FF 7F 00 00  ...........cÿ...
```

אם נעקוב אחרי המצביע השני במערך, באדום, נגיע לאובייקט שיש בו את המחרוזת "hello":

```
Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1 | [x=] Locals | Struct
Address           Hex                                              ASCII
00000287C0E7B8D0  FF FF FF FF 00 00 00 00 90 0F 14 63 FF 7F 00 00  ÿÿÿÿ.......cÿ...
00000287C0E7B8E0  05 00 00 00 00 00 00 00 C7 FF F1 E6 22 3C 08 76  ........Çÿñæ"<.v
00000287C0E7B8F0  66 64 00 20 2D 34 32 0A 68 65 6C 6C 6F 00 00 00  fd. -42.hello...
00000287C0E7B900  01 00 00 00 00 00 00 00 C0 71 12 63 FF 7F 00 00  ........Àq.cÿ...
00000287C0E7B910  01 00 00 00 00 00 00 00 FF FF FF FF FF FF FF FF  ........ÿÿÿÿÿÿÿÿ
00000287C0E7B920  20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

זהו אכן האובייקט השני ברשימה שלנו, וניתן לוודא זאת גם כן בעזרת id:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> hex(id(x[1]))
'0x287c0e7b8d0'
>>>
```

זו אותה הכתובת שהגענו אליה בעצמנו.

פונקציה שימושית נוספת היא הפונקציה [sys.getsizeof](https://docs.python.org/3/library/sys.html#sys.getsizeof) שמחזירה את הגודל של אובייקט. ליתר דיוק, את הגודל של אובייקט לא כולל הגודל של האובייקטים שהוא מצביע אליהם. כשמריצים את הפונקציה על הרשימה x שלנו מקבלים את הערך 88:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> sys.getsizeof(x)
88
>>>
```

אם נוסיף לרשימה איבר נוסף ונחשב את גודל הרשימה החדש, נקבל עדיין את הערך 88:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> x.append("goodbye")
>>> sys.getsizeof(x)
88
>>>
```

זה קורה בגלל שכמו שמצאנו, מאחורי הקלעים הרשימה ממומשת בעזרת מערך שגדל וקטן באופן דינמי. ראינו קודם ששני הערכים המעניינים שנמצאים בזיכרון של האובייקט הם 3 ו-4, ואם נסתכל שוב נראה שעכשיו הם 4 ו-4:

Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1 | Locals | Struct

Address	Hex	ASCII
00000287C13388C0	01 00 00 00 00 00 00 00 E0 77 13 63 FF 7F 00 00	........àw.cÿ...
00000287C13388D0	04 00 00 00 00 00 00 00 10 88 E9 C0 87 02 00 00	..........éÀ....
00000287C13388E0	04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00000287C13388F0	F0 B8 33 C1 87 02 00 00 B0 B8 33 C1 87 02 00 00	ð¸3Á....°¸3Á....

ולכן סביר להניח שאם נוסיף איבר נוסף עכשיו, התהליך יהיה חייב להקצות מערך חדש גדול יותר, להעתיק אליו את המצביעים מהמערך הקודם, ולהוסיף אליו את האיבר החדש. ניתן לוודא זאת עם sys.getsizeof גם כן:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> x.append("hello again")
>>> sys.getsizeof(x)
120
>>>
```

וניתן לראות בפועל בזיכרון שהערכים השתנו ל-5 ו-8, וגם שהמצביע למערך השתנה:

Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1 | Locals | Struct

Address	Hex	ASCII
00000287C13388C0	01 00 00 00 00 00 00 00 E0 77 13 63 FF 7F 00 00	........àw.cÿ...
00000287C13388D0	05 00 00 00 00 00 00 00 F0 82 34 C1 87 02 00 00	........ð.4Á....
00000287C13388E0	08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00000287C13388F0	F0 B8 33 C1 87 02 00 00 B0 B7 32 C1 87 02 00 00	ð¸3Á....°·2Á....

כמו ששיערנו התהליך הקצה מערך גדול יותר, במקרה זה בגודל כפול מהקודם - 8 איברים במקום 4, והוסיף אליו את האיבר החדש. הניסוי שעשינו גם נתן לנו קצת מושג על אלגוריתם ה-reallocation של רשימות. ראינו שמהסתכלות בזיכרון ניתן להסיק כל מיני מסקנות על איך אובייקטים מוגדרים ב-CPython, ואופן הפעולות שהם מבצעים מאחורי הקלעים. כמובן שניתן להסתכל על הקוד והתיעוד כדי להבין הכל יותר לעומק, אבל גם דרך זו עוזרת ללמידה.

PyNumberMethods

כאמור אובייקטים מסוגים שונים, לא רק מספרים, יכולים לממש חלק מהפונקציות שבטבלת ה-PyNumberMethods. דוגמאות לכך הן למשל אובייקט מסוג bytes שתומך באופרטור % לפעולת Formatting:

```
2547  static PyObject *
2548  bytes_mod(PyObject *self, PyObject *arg)
2549  {
2550      if (!PyBytes_Check(self)) {
2551          Py_RETURN_NOTIMPLEMENTED;
2552      }
2553      return _PyBytes_FormatEx(PyBytes_AS_STRING(self), PyBytes_GET_SIZE(self),
2554                               arg, 0);
2555  }
2556
2557  static PyNumberMethods bytes_as_number = {
2558      0,              /*nb_add*/
2559      0,              /*nb_subtract*/
2560      0,              /*nb_multiply*/
2561      bytes_mod,      /*nb_remainder*/
2562  };
```

[מתוך הקובץ bytesobject.c]

מה שמאפשר להריץ את הקוד:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> b'dec=%d hex=%X str=%s' % (10, 10, b'Ten')
b'dec=10 hex=A str=Ten'
```

או למשל אובייקט מסוג set שתומך באופרטורים | ,^ ,& ,-:

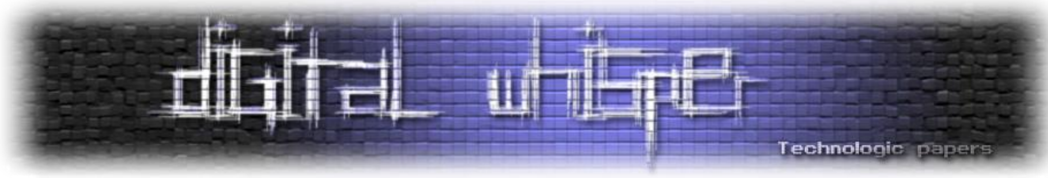
```
2086  static PyNumberMethods set_as_number = {
2087      0,                              /*nb_add*/
2088      (binaryfunc)set_sub,            /*nb_subtract*/
2089      0,                              /*nb_multiply*/
2090      ...
2091      0,                              /*nb_rshift*/
2092      (binaryfunc)set_and,            /*nb_and*/
2093      (binaryfunc)set_xor,            /*nb_xor*/
2094      (binaryfunc)set_or,             /*nb_or*/
2095      0,                              /*nb_int*/
2096      ...
2097  };
```

[מתוך הקובץ setobject.c]

מה שמאפשר להריץ את הקוד:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> {1,2,3} - {2,4}
{1, 3}
>>> {1,2,3} ^ {2,4}
{1, 3, 4}
>>> {1,2,3} & {2,4}
{2}
```

ועוד.



אנחנו נרצה להגדיר מצביע חדש לפונקציה בטבלה הפונקציות הזו, שיתאים לאופרטור $ החדש, ונממש את הפעולה המתאימה לו באובייקט שמתאים למספרים שלמים.

האובייקט PyLongObject

ב-CPython מספרים שלמים מיוצגים באובייקט ששמו PyLongObject. כאמור הלוגיקה של כל סוג אובייקט ממומשת בקובץ בשם nameobject.c, ובפרט עבור סוג אובייקט זה היא ממומשת בקובץ longobject.c. גם אובייקט ה-PyTypeObject המתאים לו מוגדר בקובץ זה. הגדרת האובייקט PyLongObject ותיעוד שלו נמצאים בקובץ longintrepr.h. להלן הגדרתו:

```
typedef struct _longobject PyLongObject;
struct _longobject {
    PyObject_HEAD
    _PyLongValue long_value;
};

typedef struct _PyLongValue {
    uintptr_t lv_tag; /* Number of digits, sign and flags */
    digit ob_digit[1];
} _PyLongValue;

typedef uint32_t digit;
```

כמו כל האובייקטים, הוא מכיל בתחילתו את ה-PyObject הסטנדרטי, עם שדות ה-ob_refcnt וה-ob_type. לאחר מכן הוא מכיל שדה בשם lv_tag (long value tag bits) שמכיל מידע על המספר המיוצג - הסימן שלו, וכמות הספרות שבו:

```
/* Long value tag bits:
 * 0-1: Sign bits value = (1-sign), ie. negative=2, positive=0, zero=1.
 * 2: Reserved for immortality bit
 * 3+ Unsigned digit count
 */
#define SIGN_MASK 3
#define SIGN_ZERO 1
#define SIGN_NEGATIVE 2
#define NON_SIZE_BITS 3
```

שני הביטים הנמוכים מייצגים את סימן המספר - אפס \ חיובי \ שלילי. הביט השלישי הוא ביט שמור, ושאר 29 הביטים מייצגים את כמו הספרות שבמספר המיוצג.

השדה האחרון ב-PyLongObject הוא מערך של ספרות המספר המיוצג, וגודלו הוא כמספר הספרות כפי שמקודד בשדה lv_tag. בהקשר של האובייקט זה, כל "ספרה" נמצאת באיבר בגודל 32 ביט במערך, אך

בפועל משתמשים רק ב-30 ביט מתוכם, ולכן "ספרה" מוגדרת כמספר באורך 30 ביט. הסיבה ל"בזבוז" 2 הביטים היא, על פי התיעוד, שהספריה המובנית marshal מצפה שמספר הביטים בכל ספרה יהיה כפולה של 15. שיהיה.

האובייקט PyLongObject בזיכרון

נסתכל על אובייקטים של כמה מספרים לדוגמא בזיכרון, למשל על המספר 42-:

```
Address           Hex                                                ASCII
00000287C0E       C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe   ..cÿ...
00000287C0E       >>> x = -42
00000287C0E       >>> hex(id(x))
00000287C0E       '0x287c0e98810'
00000287C0E       >>>
00000287C0E987F0  01 00 00 00 00 00 00 00 10 80 13 63 FF 7F 00 00  ...........cÿ...
00000287C0E98800  10 00 00 00 00 00 00 00 B0 2A F3 00 1F 0A 00 00  ........°*ó.....
00000287C0E98810  01 00 00 00 00 00 00 00 10 80 13 63 FF 7F 00 00  ...........cÿ...
00000287C0E98820  0A 00 00 00 00 00 00 00 2A 00 00 00 87 02 00 00  ........*.......
00000287C0E98830  01 00 00 00 00 00 00 00 10 80 13 63 FF 7F 00 00  ...........cÿ...
00000287C0E98840  08 00 00 00 00 00 00 00 00 00 10 00 87 02 00 00  ................
00000287C0E98850  01 00 00 00 00 00 00 00 10 80 13 63 FF 7F 00 00  ...........cÿ...
```

שדה ה-lv_tag (בכחול) נמצא לאחר שדות ה-ob_refcnt וה-ob_type הסטנדרטיים. מכיון שהמספר שלילי, 2 הביטים הנמוכים בשדה הם 10. הביט הבא (reserved) הוא 0 תמיד. שאר הביטים מייצגים את כמות הספרות במספר, ומכיוון שהוא מורכב מספרה אחת, ביטים אלה מכילים את הערך 01...0. בסך הכל שדה ה-lv_tag הוא 01010...0 שמתאים לערך 0xA כפי שרואים בפועל. לאחר מכן נמצא המערך ob_digit שמכיל איבר אחד של ספרה - והערך שלה הוא 0x2A שמתאים למספר 42 שלנו.

אם נסתכל על אובייקט שמתאים למספר שגודלו 31 ביטים ומעלה, נראה שבמערך ה-ob_digit שלו יש יותר מספרה אחת. למשל המספר 0x12345678ABCDEF:

```
Address           Hex                                                ASCII
00000287C10E      C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe   3Á....
00000287C10E      >>> x = 0x12345678ABCDEF
00000287C10E      >>> hex(id(x))
00000287C10E      '0x287c10e4ff0'
00000287C10F      >>>
00000287C10E4FD0  50 4F 0E C1 87 02 00 00 10 80 13 63 FF 7F 00 00  PO.Á.......cÿ...
00000287C10E4FE0  08 00 00 00 00 00 00 00 10 30 00 00 87 02 00 00  .........0......
00000287C10E4FF0  01 00 00 00 00 00 00 00 10 80 13 63 FF 7F 00 00  ...........cÿ...
00000287C10E5000  10 00 00 00 00 00 00 00 EF CD AB 38 59 D1 48 00  ........ïÍ«8YÑH.
00000287C10E5010  D0 4F 0E C1 87 02 00 00 10 80 13 63 FF 7F 00 00  ÐO.Á.......cÿ...
00000287C10E5020  08 00 00 00 00 00 00 00 10 30 00 00 59 D1 48 00  .........0..YÑH.
```

מכיוון שהמספר חיובי, שני הביטים הנמוכים בשדה ה-lv_tag הם 00. הביט הבא הוא 0. הפעם המספר דורש שתי ספרות ולכן שני הביטים הבאים הם 10. בסך הכל שדה ה-lv_tag הוא 010000...0 שמתאים לערך 0x10 שרואים בפועל.

ניתן גם לראות שבספרה הראשונה נמצאים רק 30 הביטים הנמוכים (0x38ABCDEF), ובספרה השניה מופיעים השאר (0x48D159). זה נובע מהעובדה ש-2 הביטים הנמוכים בכל ספרה "מבוזבזים".

צורה נוחה לראות בדיוק מתי מתווספת ספרה למספר היא שימוש בפונקציה sys.getsizeof. אם נסתכל על גודל של מספר בעל 30 ביטים נקבל את הגודל 28:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> sys.getsizeof(2**30-1)
28
>>>
```

הסיבה לכך היא:

- 8 בתים ל-ob_refcnt.
- 8 בתים ל-ob_type.
- 8 בתים ל-lv_tag.
- 4 בתים לספרה הראשונה של המספר.

לעומת זאת, אם נסתכל על גודל של מספר בעל 31 ביטים נקבל את הגודל:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> sys.getsizeof(2**30)
32
>>>
```

מכיוון שעכשיו צריך 2 ספרות כדי לייצג את המספר, נוספים 4 בתים בשביל הספרה השניה.

אופטימיזציה של PyLongObject על מספרים קטנים

לפני שנסכם חלק זה, כדאי לציין שקיימת אופטימיזציה מסוימת על מספרים קטנים. כשמדפיסים את הכתובות של האובייקטים שמתאימים למספרים שבין 250 ל-263 זה הפלט שמתקבל:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> for i in range(250, 264):
...  print(i, hex(id(i)))
...
250 0x7ffe4e8f1aa0
251 0x7ffe4e8f1ac0
252 0x7ffe4e8f1ae0
253 0x7ffe4e8f1b00
254 0x7ffe4e8f1b20
255 0x7ffe4e8f1b40
256 0x7ffe4e8f1b60
257 0x26e7cad4eb0
258 0x26e7cad4d70
259 0x26e7cad4eb0
260 0x26e7cad4d70
261 0x26e7cad4eb0
262 0x26e7cad4d70
263 0x26e7cad4eb0
```

נראה שהמספרים 250 עד 256 נמצאים אחד אחרי השני בזיכרון, בהפרש של 0x20 בתים אחד מהשני. שאר המספרים נמצאים במקום אחר לגמרי בזיכרון, ובכתובות שחוזרות על עצמן. כלומר האובייקטים של מספרים אלו מוקצים ומשתחררים דינמית בהתאם לשימוש בהם.

הזכרתי קודם שקיימים אובייקטים שמכונים "immortal", וטווח המספרים -5 עד 256 הם חלק מהאובייקטים האלו. אובייקטים אלו מוגדרים פעם אחת בצורה סטטית בקוד של CPython ולעולם לא משתחררים מהזיכרון. מדובר באופטימיזציה של CPython על מספרים קטנים שמשתמשים בהם יחסית הרבה, והיא חוסכת הקצאות זיכרון מיותרות. כשב-Interpreter נעשה שימוש במספר בטווח זה, במקום להקצות אובייקט עבורו ולשחרר אותו בסוף השימוש, מוחזר מצביע לאובייקט ה-immortal הסטטי המתאים.

אובייקטים אלה, ועוד אחרים, מוגדרים בקובץ pycore_global_objects.h תחת struct בשם _Py_static_objects. המספרים הקטנים מוגדרים במערך בשם small_ints:

```
37  struct _Py_static_objects {
        Eric Snow, 15 months ago | 1 author (Eric Snow)
38      struct {
39          /* Small integers are preallocated in this array so that they
40           * can be shared.
41           * The integers that are preallocated are those in the range
42           * -_PY_NSMALLNEGINTS (inclusive) to _PY_NSMALLPOSINTS (exclusive).
43           */
44          PyLongObject small_ints[_PY_NSMALLNEGINTS + _PY_NSMALLPOSINTS];
45
46          PyBytesObject bytes_empty;
            Eric Snow, 2 years ago | 1 author (Eric Snow)
47          struct {
48              PyBytesObject ob;
49              char eos;
50          } bytes_characters[256];
51
52          struct _Py_global_strings strings;
```

ועם הקבועים:

```
20  #define _PY_NSMALLPOSINTS           257
21  #define _PY_NSMALLNEGINTS           5
```

בהמשך נשתמש באובייקטים של מספרים קטנים אלה למימוש הלוגיקה של חישוב פונקציית קולץ.

בחזרה למימוש פקודת ה-Bytecode החדשה

לאחר כל ההקדמה הזו על אובייקטים, אפשר לחזור למימוש פקודת ה-Bytecode החדשה.

הוספת פונקציה חדשה לטבלת הפונקציות PyNumberMethods

כאמור, כל סוג אובייקט קובע לעצמו איזה מהפונקציות מבין אלו שבטבלה PyNumberMethods הוא מממש. נוסיף לטבלה זו רשומה חדשה, שתהיה מצביע לפונקציית Collatz שאובייקט יכול לממש. באובייקט מסוג PyLongObject נכתוב ברשומה זו מצביע לפונקציה, ובכל שאר סוגי האובייקטים המצביע הזה יהיה מאופס. הגדרת הטבלה נמצאת בקובץ object.h. נוסיף בה מצביע חדש בשם nb_collatz מסוג unaryfunc:

```
61  typedef struct {
62      binaryfunc nb_add;
63      binaryfunc nb_subtract;
64      binaryfunc nb_multiply;
65      binaryfunc nb_remainder;
66      binaryfunc nb_divmod;
67      ternaryfunc nb_power;
68      unaryfunc nb_negative;
69      unaryfunc nb_positive;
70      unaryfunc nb_absolute;
71      unaryfunc nb_collatz;
72      inquiry nb_bool;
73      unaryfunc nb_invert;
74      binaryfunc nb_lshift;
75      binaryfunc nb_rshift;
76      binaryfunc nb_and;
77      binaryfunc nb_xor;
78      binaryfunc nb_or;
```

לאחר מכן יש ללכת לכל האובייקטים שמשתמשים בטבלה זו, ולכל אחד מהם להגדיר את הערך 0 במקום שמתאים ל-nb_collatz בטבלה. למשל בקובץ boolobject.c:

```
122  static PyNumberMethods bool_as_number = {
123      0,                          /* nb_add */
124      0,                          /* nb_subtract */
125      0,                          /* nb_multiply */
126      0,                          /* nb_remainder */
127      0,                          /* nb_divmod */
128      0,                          /* nb_power */
129      0,                          /* nb_negative */
130      0,                          /* nb_positive */
131      0,                          /* nb_absolute */
132      0,                          /* nb_collatz */
133      0,                          /* nb_bool */
134      (unaryfunc)bool_invert,     /* nb_invert */
135      0,                          /* nb_lshift */
136      0,                          /* nb_rshift */
137      bool_and,                   /* nb_and */
138      bool_xor,                   /* nb_xor */
```

יוצא מן הכלל יהיה כמובן האובייקט PyLongObject.

בטבלת ה-PyNumberMethods שבקובץ longobject.c נגדיר מצביע לפונקציה חדשה בשם long_collatz במקום שמתאים ל-nb_collatz:

```
6256  static PyNumberMethods long_as_number = {
6257      (binaryfunc)long_add,       /*nb_add*/
6258      (binaryfunc)long_sub,       /*nb_subtract*/
6259      (binaryfunc)long_mul,       /*nb_multiply*/
6260      long_mod,                   /*nb_remainder*/
6261      long_divmod,                /*nb_divmod*/
6262      long_pow,                   /*nb_power*/
6263      (unaryfunc)long_neg,        /*nb_negative*/
6264      long_long,                  /*tp_positive*/
6265      (unaryfunc)long_abs,        /*tp_absolute*/
6266      (unaryfunc)long_collatz,    /*nb_collatz*/
6267      (inquiry)long_bool,         /*tp_bool*/
6268      (unaryfunc)long_invert,     /*nb_invert*/
6269      long_lshift,                /*nb_lshift*/
6270      long_rshift,                /*nb_rshift*/
6271      long_and,                   /*nb_and*/
6272      long_xor,                   /*nb_xor*/
6273      long_or,                    /*nb_or*/
```

ניצור את הפונקציה הזו באותו קובץ, ובתור התחלה נכתוב לה לוגיקה שתחזיר תמיד את המספר הקבוע 1337, רק כדי לראות שהכל עובד כמו שצריך בינתיים:

```
4889  static PyObject *
4890  long_collatz(PyLongObject *v)
4891  {
4892      return PyLong_FromLong(1337);
4893  }
```

בנוסף למה שעשינו עד כה, נצטרך להגדיר מתי נרצה לקרוא לפונקציה שנמצאת במצביע החדש שהוספנו לטבלה, nb_collatz. כזכור, בקובץ bytecodes.c הגדרנו שכשה-Interpreter יריץ את פקודת ה-Bytecode החדשה, תיקרא הפונקציה PyNumber_Collatz. עדיין לא הגדרנו אותה.

הפונקציות מהצורה PyNumber_Name מוגדרות בקובץ abstract.c בעזרת המאקרו UNARY_FUNC:

```
1380  UNARY_FUNC(PyNumber_Negative, nb_negative, __neg__, "unary -")
1381  UNARY_FUNC(PyNumber_Positive, nb_positive, __pow__, "unary +")
1382  UNARY_FUNC(PyNumber_Invert, nb_invert, __invert__, "unary ~")
1383  UNARY_FUNC(PyNumber_Absolute, nb_absolute, __abs__, "abs()")
```

מאקרו זה מקבל את שם הפונקציה PyNumber_Name שרוצים להגדיר, ואת השדה המתאים למצביע הפונקציה בטבלה PyNumberMethods שרוצים להתאים לה, ומגדיר את הפונקציה.

נוסיף לקובץ זה שורה חדשה עבור הפונקציה PyNumber_Collatz שתקרא לפונקציה שמתאימה ל- nb_collatz בטבלה:

```
1380    UNARY_FUNC(PyNumber_Negative, nb_negative, __neg__, "unary -")
1381    UNARY_FUNC(PyNumber_Positive, nb_positive, __pow__, "unary +")
1382    UNARY_FUNC(PyNumber_Invert, nb_invert, __invert__, "unary ~")
1383    UNARY_FUNC(PyNumber_Absolute, nb_absolute, __abs__, "abs()")
1384    UNARY_FUNC(PyNumber_Collatz, nb_collatz, __collatz__, "unary $")
```

בנוסף, יש להוסיף את חתימת הפונקציה בקובץ abstract.h בדומה לחתימות שאר הפונקציות בו:

```
491   /* Returns the absolute value of 'o', or NULL on failure.
492
493      This is the equivalent of the Python expression: abs(o). */
494   PyAPI_FUNC(PyObject *) PyNumber_Absolute(PyObject *o);
495
496   /* Returns the Collatz value of 'o', or NULL on failure.
497
498      This is the equivalent of the Python expression: $o. */
499   PyAPI_FUNC(PyObject *) PyNumber_Collatz(PyObject *o);
500
```

Constant Folding

לפני שנקמפל ונבדוק שכל השינויים שהכנסנו עד כה מתנהגים כצפוי, נשאר לנו עוד דבר אחד לעשות.

בתהליך הקומפילציה של קוד Python ישנה אופטימיזציה ברמת ה-AST שנקראת Constant Folding. כשקוד Python מכיל ביטוי מתמטי עם מספרים קבועים, אז במקום לקמפל את הקוד לפקודות Bytecode שמבצעות את החישוב בזמן ריצה ב-Interpreter, החישוב מתבצע בזמן הקומפילציה של הקוד ומשתמשים בתוצאת החישוב ב-Bytecode שנוצר.

דוגמא לכך היא קוד שמכיל למשל את הביטוי:

```
x = 3 + 4
```

לכאורה צריך לקמפל את הקוד הזה לפקודות:

```
PUSH 3, PUSH 4, BINARY_OP +, POP x
```

במקום זאת, בזמן הקומפילציה של הקוד מתבצע החישוב 3 + 4 = 7 והקוד מתקמפל לפקודות:

```
PUSH 7, POP x
```

ניתן לראות זאת בפועל:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> dis.dis("x = 3 + 4")
  0           RESUME                   0

  1           LOAD_CONST               0 (7)
              STORE_NAME               0 (x)
              RETURN_CONST             1 (None)
>>>
```

האופטימיזציה הזאת קורית בפונקציה ast_foldexpr שבקובץ ast_opt.c. במקרה שהביטוי מכיל אופרטור אונרי, כמו במקרה של האופרטור החדש שהוספנו, הפונקציה שנקראת היא fold_unaryop. להלן סוף הפונקציה:

```
115     typedef PyObject *(*unary_op)(PyObject*);
116     static const unary_op ops[] = {
117         [Invert] = PyNumber_Invert,
118         [Not] = unary_not,
119         [UAdd] = PyNumber_Positive,
120         [USub] = PyNumber_Negative,
121     };
122     PyObject *newval = ops[node->v.UnaryOp.op](arg->v.Constant.value);
123     return make_const(node, newval, arena);
124 }
```

בקטע קוד זה ישנה טבלת פונקציות קטנה שיש בה מיפוי בין אופרטור אונרי לפונקציה שיש להריץ כשרוצים לבצע את החישוב המתאים לו בזמן קומפילציה. הקוד בוחר את הפונקציה המתאימה, קורא לה, ומחזיר את תוצאת החישוב כקבוע חדש. נוסיף לה התייחסות לאופרטור החדש ואת הפונקציה החדשה המתאימה לו:

```
115     typedef PyObject *(*unary_op)(PyObject*);
116     static const unary_op ops[] = {
117         [Invert] = PyNumber_Invert,
118         [Not] = unary_not,
119         [UAdd] = PyNumber_Positive,
120         [USub] = PyNumber_Negative,
121         [Collatz] = PyNumber_Collatz,
122     };
123     PyObject *newval = ops[node->v.UnaryOp.op](arg->v.Constant.value);
124     return make_const(node, newval, arena);
125 }
```

תוצאת הביניים

עד כה עשינו הרבה שינויים ולכן זהו זמן טוב לבדוק שהכל עובד כצפוי. כשמקמפלים את הקוד ומריצים, מקבלים את התוצאה:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> dis.dis("$x")
  0           RESUME                   0

  1           LOAD_NAME                0 (x)
              UNARY_COLLATZ
              RETURN_VALUE
>>>
```

רואים שאכן הקוד שלנו מתקמפל לפקודת ה-Bytecode החדשה שיצרנו, כבר מגניב!

כשמריצים את הפקודה רואים שהיא אכן עובדת כצפוי, ללא הלוגיקה האמיתית, ומחזירה את הקבוע 1337:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> x = 5
>>> $x
1337
>>>
```

כלומר ה-Interpreter גם יודע איך להריץ את הפקודה החדש שיצרנו!

הפונקציה שלנו long_collatz אכן נקראת כשה-Interpreter מריץ את פקודת ה-Bytecode החדשה, ובנוסף לכך ניתן לראות גם את האופטימיזציה של החישוב בזמן קומפילציה במקום בזמן ריצה:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> dis.dis("$5")
  0           RESUME                   0

  1           RETURN_CONST             0 (1337)
>>>
```

במקום לקמפל ביטוי שמכיל מספר קבוע לפקודת ה-Bytecode החדשה, הקומפיילר מריץ את ה"חישוב" שלנו בזמן קומפילציה ומשתמש בתוצאה ב-Bytecode שנוצר. אמנם הקומפיילר לא מספיק חכם להבין שב"מימוש" הנוכחי של הפקודה ניתן לבצע את האופטימיזציה גם אם המספר אינו קבוע, אבל לפחות הוא מנסה.

מימוש לוגיקת הפקודה החדשה באובייקט PyLongObject

כל התהליך הארוך שעברנו עד כה נועד להביא אותנו למצב שה-Interpreter יריץ את הפונקציה long_collatz על אובייקט מסוג מספר שלם, כשהוא מבצע את פקודת ה-Bytecode שהוספנו. הגיע הזמן להכניס בה את הלוגיקה שרצינו מלכתחילה - חישוב פונקציית הקולץ של המספר.

הלוגיקה שנרצה היא די פשוטה:

- אם מספר הקלט שלילי או אפס - נגדיר את תוצאת החישוב להיות אפס. כמובן שניתן גם לבחור כל ערך אחר, להשאיר את המספר כפי שהוא, או לבצע חישוב אחר כרצוננו.
- אחרת:
  - אם המספר זוגי - נחלק אותו בשתיים.
  - אם המספר אי-זוגי - נכפיל אותו בשלוש, ונוסיף אחד.

להלן הקוד המתאים:

```
4889  static PyObject *
4890  long_collatz(PyLongObject *v)
4891  {
4892      PyObject* res;
4893      if (_PyLong_IsNegative(v) || _PyLong_IsZero(v))
4894          return _PyLong_FromUnsignedChar(0);
4895
4896      if ((v->long_value.ob_digit[0] & 1) == 0) {
4897          // even
4898          res = long_div((PyObject *)v, _PyLong_FromUnsignedChar(2));
4899      } else {
4900          // odd
4901          PyObject* temp = long_mul(v, (PyLongObject *)_PyLong_FromUnsignedChar(3));
4902          res = long_add((PyLongObject *)temp, (PyLongObject *)_PyLong_FromUnsignedChar(1));
4903          Py_DECREF(temp);
4904      }
4905      return res;
4906  }
```

והסבר לכל חלק בו:

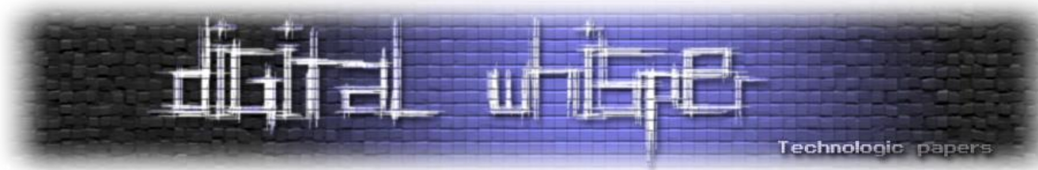
ה-if הראשון בפונקציה מתמודד עם המקרה שמדובר במספר שלילי או אפס - ובמקרה שכזה מוחזר האובייקט שמתאים למספר 0. הפונקציה _PyLong_FromUnsignedChar מקבלת מספר קטן ומחזירה את האובייקט הסטטי שמתאים לאותו מספר ממערך ה-small_ints שראינו קודם. אנו קוראים לפונקציה זו מספר פעמים במימוש שלנו, ומכיוון שהאובייקט המוחזר הוא immortal, אין צורך להתייחס לעדכון שדה ה-ob_refcnt שלו.

ה-if הבא הוא בדיקה האם המספר הוא זוגי או אי-זוגי. לפי התיעוד של האובייקט PyLongObject, מכיוון שתמיד יש לפחות ספרה אחת במספר המיוצג, ניתן תמיד לגשת לשדה ob_digit[0]. להנחה זו יש יוצא דופן אחד שהוא המקרה שבו המספר המיוצג הוא 0. לכן לפני שבודקים האם המספר זוגי או אי-זוגי, מה שמצריך לגשת ל-ob_digit, יש לבדוק האם הוא אפס. ה-if הראשון של הפונקציה מתמודד עם זה ולכן בשלב זה ניתן לגשת ל-ob_digit בשביל בדיקת הזוגיות.

ראינו קודם שמספר מיוצג כ-Little Endian, כלומר הספרה הנמוכה במספר היא הראשונה במערך ob_digit. לכן כדי לבדוק האם המספר המיוצג הוא זוגי או אי-זוגי, מספיק להסתכל על הביט הנמוך בספרה הראשונה - אם הוא אפס אז המספר זוגי, ואם הוא אחד אז המספר אי זוגי.

במקרה שהמספר זוגי, ערך החזרה של הפונקציה הוא ערך החזרה של הפונקציה long_div - שנקראת עם אובייקט הקלט ועם הקבוע 2. הפונקציה long_div מקצה אובייקט חדש שמכיל את תוצאת החישוב של חלוקת המספר ב-2, ומחזירה מצביע אליו.

במקרה שהמספר אי-זוגי, אנו משתמשים בפונקציה long_mul - שנקראת עם אובייקט הקלט ועם הקבוע 3. גם במקרה זה הפונקציה מחזירה אובייקט חדש שמכיל את תוצאת חישוב הכפל, שנכנס לתוך משתנה בשם temp. לאחר מכן אנו משתמשים בפונקציה long_add - שנקראת עם temp ועם הקבוע 1. ערך החזרה של פונקציה זו הוא מצביע לאובייקט חדש, והוא גם ערך החזרה של הפונקציה שלנו.



לבסוף מקטינים את שדה ה-ob_refcnt של האובייקט temp. בניגוד למקרה שבו המספר זוגי, במקרה שהמספר אי-זוגי אנחנו יוצרים אובייקט חדש זמני בשם temp. בגלל שסיימנו להשתמש בו ולא נצטרך אותו בהמשך, לפני שנצא מהפונקציה נצטרך להקטין לו את שדה ה-ob_refcnt כדי שהאובייקט ישוחרר מהזיכרון. אם לא נעשה זאת, יהיו לנו אובייקטים בזיכרון שלא ישוחררו לעולם ולא יהיו מצביעים אליהם בכלל.

מיפוי האופרטור החדש ל-Magic Method

Magic Method הוא כינוי לפונקציות מחלקה ב-Python שהשם שלהן מתחיל ומסתיים בשני תווי underscore. למשל __eq__, __add__, __str__, וכו'. כשמחלקה ממשת פונקציה כזו, Python יודע לקרוא לפונקציה המתאימה כשמשתמשים באובייקט מהמחלקה בצורה מסוימת. למשל אם מחלקה ממשת את הפונקציה __add__, אז כשקוד ה-Python יכיל פעולת חיבור בין אובייקט מהמחלקה לאובייקט אחר, תיקרא הפונקציה הזו במקום הפונקציה ברירת המחדל של Python, אם היא קיימת בכלל.

כדי למפות בין מצביע הפונקציה nb_collatz בטבלת ה-PyNumberMethods, לבין שם ה-Magic Method הרצוי, יש לבצע את שני השינויים הבאים:

1. בקובץ typeobject.c להוסיף slot להגדרת הפונקציה slot_nb_collatz:

```
8644    SLOT0(slot_nb_negative, __neg__)
8645    SLOT0(slot_nb_positive, __pos__)
8646    SLOT0(slot_nb_absolute, __abs__)
8647    SLOT0(slot_nb_collatz, __collatz__)
```

וגם להוסיף שימוש במאקרו UNSLOT כדי למפות בין:

- המחרוזת שהיא שם הפונקציה הרצוי __collatz__.
- שם שדה המצביע לפונקציה nb_collatz, כפי שהגדרנו בטבלת ה-PyNumberMethods.
- ה-slot שיצרנו, slot_nb_collatz.

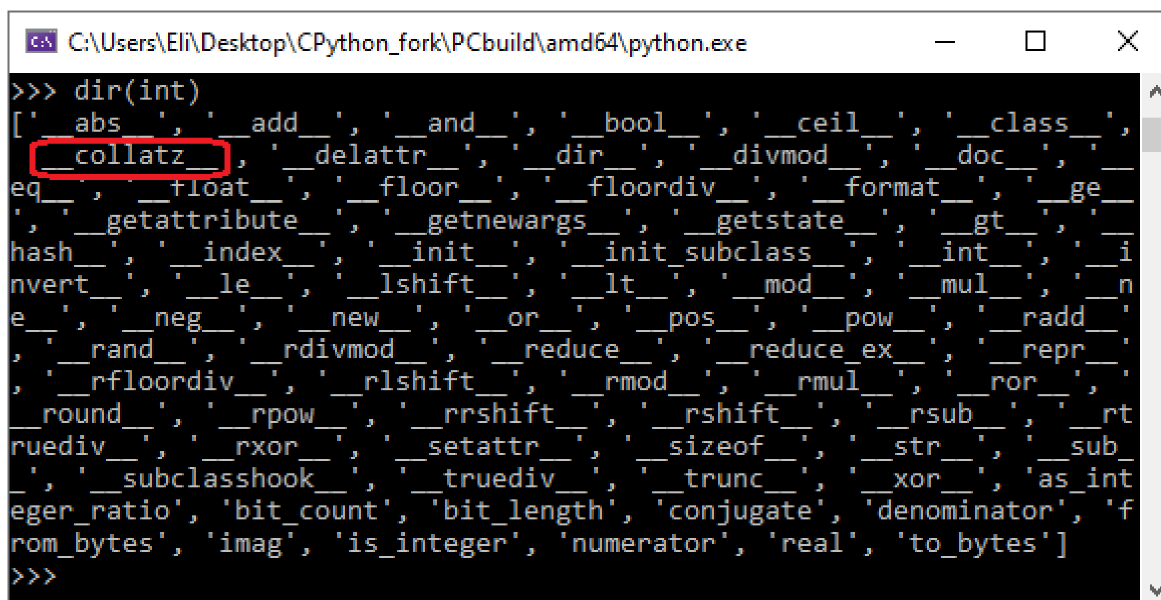
```
9569        UNSLOT(__neg__, nb_negative, slot_nb_negative, wrap_unaryfunc, "-self"),
9570        UNSLOT(__pos__, nb_positive, slot_nb_positive, wrap_unaryfunc, "+self"),
9571        UNSLOT(__abs__, nb_absolute, slot_nb_absolute, wrap_unaryfunc,
9572               "abs(self)"),
9573        UNSLOT(__collatz__, nb_collatz, slot_nb_collatz, wrap_unaryfunc, "$self"),
```

2. לערוך את הסקריפט generate_global_objects.py כך שיכלול את המחרוזת __collatz__ בנוסף למחרוזות שמות הפונקציות האחרות:

```
44      # from SLOT* in Objects/typeobject.c
45      '__abs__',
46      '__collatz__',
47      '__add__',
```

סקריפט זה רץ באופן אוטומטי כשמקמפלים את CPython, מה שגורם לעדכון של מספר קבצי h. בפרויקט.

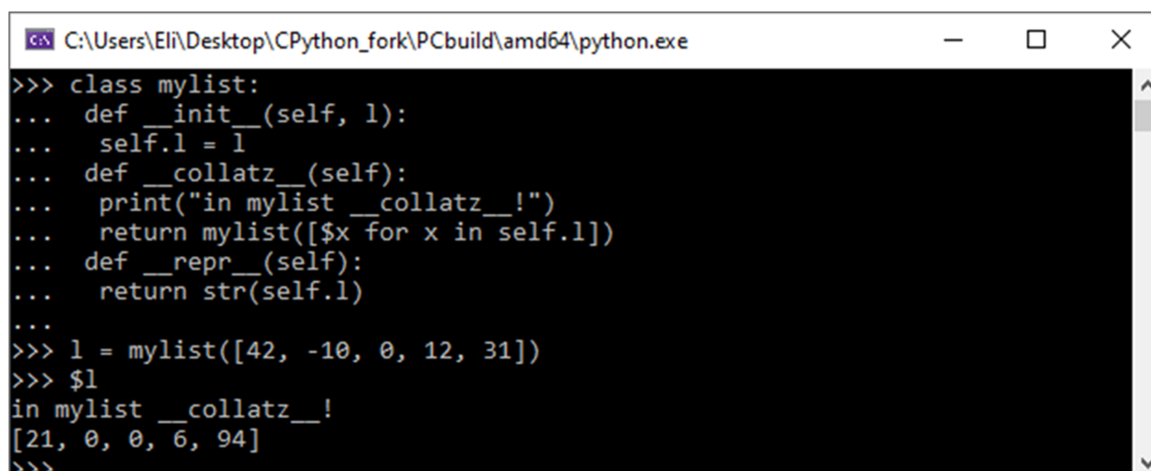
לאחר פעולות אלה, ניתן להגדיר פונקציית מחלקה בשם __collatz__ שתיקרא כשמשתמשים באופרטור $ על אובייקט מהמחלקה. אם לדייק יותר, הפונקציה תיקרא כשה-Interpreter ינסה לבצע את פקודת ה-Bytecode החדשה UNARY_COLLATZ על אובייקט מהמחלקה. בנוסף ניתן לראות שהפונקציה __collatz__ התווספה לרשימת הפונקציות של האובייקט int:



```
>>> dir(int)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__collatz__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__getstate__', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'as_integer_ratio', 'bit_count', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'is_integer', 'numerator', 'real', 'to_bytes']
>>>
```

התוצאה

התוצאה הברורה היא שאכן הוספנו פקודת Bytecode חדשה בהצלחה והיא עובדת ומגניבה:



```
>>> class mylist:
...  def __init__(self, l):
...   self.l = l
...  def __collatz__(self):
...   print("in mylist __collatz__!")
...   return mylist([$x for x in self.l])
...  def __repr__(self):
...   return str(self.l)
...
>>> l = mylist([42, -10, 0, 12, 31])
>>> $l
in mylist __collatz__!
[21, 0, 0, 6, 94]
>>>
```

נשווה זאת עם מה שעשינו במאמר הקודם. במאמר הקודם קימפלנו אופרטור חדש לפקודות Bytecode קיימות, ולכן אם היינו לוקחים את קובץ ה-pyc. שקומפל מהקוד שלנו ומנסים להריץ אותו על Interpreter "רגיל", הוא היה עובד. לעומת זאת, השינוי שעשינו במאמר זה שינה את ה-Bytecode עצמו ואת ה-Interpreter בהתאם לכך. אם היינו כותבים קוד שמכיל את פקודת ה-Collatz החדשה, ולוקחים את קובץ ה-pyc. שנוצר ומנסים להריץ אותו על Interpreter "רגיל", הוא לא היה יודע איך להריץ את הפקודה הזאת.

בנוסף לכך, גרמנו לתופעת לוואי מסוימת. נראה אותה בעזרת שימוש בספריה dis שמאפשרת לראות את מיפוי פקודות ה-Bytecode ל-opcode (בקישור זה קיים גם תיעוד קצר לכל פקודת Bytecode).

לפני השינוי:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> dis.opmap
{'CACHE': 0, 'RESERVED': 17, 'RESUME': 149, 'INSTRUMENTED_LINE': 254, 'BEFORE_ASYNC_WITH': 1, 'BEFORE_WITH': 2, 'BINARY_SLICE': 4, 'BINARY_SUBSC
R': 5, 'CHECK_EG_MATCH': 6, 'CHECK_EXC_MATCH': 7, 'CLEANUP_THROW': 8, 'DELETE_SUBSCR': 9, 'END_ASYNC_FOR': 10, 'END_FOR': 11, 'END_SEND': 12, 'E
XIT_INIT_CHECK': 13, 'FORMAT_SIMPLE': 14, 'FORMAT_WITH_SPEC': 15, 'GET_AITER': 16, 'GET_ANEXT': 18, 'GET_ITER': 19, 'GET_LEN': 20, 'GET_YIELD_FR
OM_ITER': 21, 'INTERPRETER_EXIT': 22, 'LOAD_ASSERTION_ERROR': 23, 'LOAD_BUILD_CLASS': 24, 'LOAD_LOCALS': 25, 'MAKE_FUNCTION': 26, 'MATCH_KEYS':
27, 'MATCH_MAPPING': 28, 'MATCH_SEQUENCE': 29, 'NOP': 30, 'POP_EXCEPT': 31, 'POP_TOP': 32, 'PUSH_EXC_INFO': 33, 'PUSH_NULL': 34, 'RETURN_GENERAT
OR': 35, 'RETURN_VALUE': 36, 'SETUP_ANNOTATIONS': 37, 'STORE_SLICE': 38, 'STORE_SUBSCR': 39, 'TO_BOOL': 40, 'UNARY_INVERT': 41, 'UNARY_NEGATIVE'
: 42, 'UNARY_NOT': 43, 'WITH_EXCEPT_START': 44, 'BINARY_OP': 45, 'BUILD_CONST_KEY_MAP': 46, 'BUILD_LIST': 47, 'BUILD_MAP': 48, 'BUILD_SET': 49,
'BUILD_SLICE': 50, 'BUILD_STRING': 51, 'BUILD_TUPLE': 52, 'CALL': 53, 'CALL_FUNCTION_EX': 54, 'CALL_INTRINSIC_1': 55, 'CALL_INTRINSIC_2': 56, 'C
ALL_KW': 57, 'COMPARE_OP': 58, 'CONTAINS_OP': 59, 'CONVERT_VALUE': 60, 'COPY': 61, 'COPY_FREE_VARS': 62, 'DELETE_ATTR': 63, 'DELETE_DEREF': 64,
'DELETE_FAST': 65, 'DELETE_GLOBAL': 66, 'DELETE_NAME': 67, 'DICT_MERGE': 68, 'DICT_UPDATE': 69, 'ENTER_EXECUTOR': 70, 'EXTENDED_ARG': 71, 'FOR_I
TER': 72, 'GET_AWAITABLE': 73, 'IMPORT_FROM': 74, 'IMPORT_NAME': 75, 'IS_OP': 76, 'JUMP_BACKWARD': 77, 'JUMP_BACKWARD_NO_INTERRUPT': 78, 'JUMP_F
ORWARD': 79, 'LIST_APPEND': 80, 'LIST_EXTEND': 81, 'LOAD_ATTR': 82, 'LOAD_CONST': 83, 'LOAD_DEREF': 84, 'LOAD_FAST': 85, 'LOAD_FAST_AND_CLEAR':
86, 'LOAD_FAST_CHECK': 87, 'LOAD_FAST_LOAD_FAST': 88, 'LOAD_FROM_DICT_OR_DEREF': 89, 'LOAD_FROM_DICT_OR_GLOBALS': 90, 'LOAD_GLOBAL': 91, 'LOAD_N
AME': 92, 'LOAD_SUPER_ATTR': 93, 'MAKE_CELL': 94, 'MAP_ADD': 95, 'MATCH_CLASS': 96, 'POP_JUMP_IF_FALSE': 97, 'POP_JUMP_IF_NONE': 98, 'POP_JUMP_I
F_NOT_NONE': 99, 'POP_JUMP_IF_TRUE': 100, 'RAISE_VARARGS': 101, 'RERAISE': 102, 'RETURN_CONST': 103, 'SEND': 104, 'SET_ADD': 105, 'SET_FUNCTION_
ATTRIBUTE': 106, 'SET_UPDATE': 107, 'STORE_ATTR': 108, 'STORE_DEREF': 109, 'STORE_FAST': 110, 'STORE_FAST_LOAD_FAST': 111, 'STORE_FAST_STORE_FAS
T': 112, 'STORE_GLOBAL': 113, 'STORE_NAME': 114, 'SWAP': 115, 'UNPACK_EX': 116, 'UNPACK_SEQUENCE': 117, 'YIELD_VALUE': 118, 'INSTRUMENTED_RESUME
': 236, 'INSTRUMENTED_END_FOR': 237, 'INSTRUMENTED_END_SEND': 238, 'INSTRUMENTED_RETURN_VALUE': 239, 'INSTRUMENTED_RETURN_CONST': 240, 'INSTRUME
NTED_YIELD_VALUE': 241, 'INSTRUMENTED_LOAD_SUPER_ATTR': 242, 'INSTRUMENTED_FOR_ITER': 243, 'INSTRUMENTED_CALL': 244, 'INSTRUMENTED_CALL_KW': 245
, 'INSTRUMENTED_CALL_FUNCTION_EX': 246, 'INSTRUMENTED_INSTRUCTION': 247, 'INSTRUMENTED_JUMP_FORWARD': 248, 'INSTRUMENTED_JUMP_BACKWARD': 249, 'I
NSTRUMENTED_POP_JUMP_IF_TRUE': 250, 'INSTRUMENTED_POP_JUMP_IF_FALSE': 251, 'INSTRUMENTED_POP_JUMP_IF_NONE': 252, 'INSTRUMENTED_POP_JUMP_IF_NOT_N
ONE': 253, 'JUMP': 256, 'JUMP_NO_INTERRUPT': 257, 'LOAD_CLOSURE': 258, 'LOAD_METHOD': 259, 'LOAD_SUPER_METHOD': 260, 'LOAD_ZERO_SUPER_ATTR': 261
, 'LOAD_ZERO_SUPER_METHOD': 262, 'POP_BLOCK': 263, 'SETUP_CLEANUP': 264, 'SETUP_FINALLY': 265, 'SETUP_WITH': 266, 'STORE_FAST_MAYBE_NULL': 267}
>>>
```

אחרי השינוי:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> dis.opmap
{'CACHE': 0, 'RESERVED': 17, 'RESUME': 149, 'INSTRUMENTED_LINE': 254, 'BEFORE_ASYNC_WITH': 1, 'BEFORE_WITH': 2, 'BINARY_SLICE': 4, 'BINARY_SUBSC
R': 5, 'CHECK_EG_MATCH': 6, 'CHECK_EXC_MATCH': 7, 'CLEANUP_THROW': 8, 'DELETE_SUBSCR': 9, 'END_ASYNC_FOR': 10, 'END_FOR': 11, 'END_SEND': 12, 'E
XIT_INIT_CHECK': 13, 'FORMAT_SIMPLE': 14, 'FORMAT_WITH_SPEC': 15, 'GET_AITER': 16, 'GET_ANEXT': 18, 'GET_ITER': 19, 'GET_LEN': 20, 'GET_YIELD_FR
OM_ITER': 21, 'INTERPRETER_EXIT': 22, 'LOAD_ASSERTION_ERROR': 23, 'LOAD_BUILD_CLASS': 24, 'LOAD_LOCALS': 25, 'MAKE_FUNCTION': 26, 'MATCH_KEYS':
27, 'MATCH_MAPPING': 28, 'MATCH_SEQUENCE': 29, 'NOP': 30, 'POP_EXCEPT': 31, 'POP_TOP': 32, 'PUSH_EXC_INFO': 33, 'PUSH_NULL': 34, 'RETURN_GENERAT
OR': 35, 'RETURN_VALUE': 36, 'SETUP_ANNOTATIONS': 37, 'STORE_SLICE': 38, 'STORE_SUBSCR': 39, 'TO_BOOL': 40, 'UNARY_COLLATZ': 41, 'UNARY_INVERT':
 42, 'UNARY_NEGATIVE': 43, 'UNARY_NOT': 44, 'WITH_EXCEPT_START': 45, 'BINARY_OP': 46, 'BUILD_CONST_KEY_MAP': 47, 'BUILD_LIST': 48, 'BUILD_MAP':
49, 'BUILD_SET': 50, 'BUILD_SLICE': 51, 'BUILD_STRING': 52, 'BUILD_TUPLE': 53, 'CALL': 54, 'CALL_FUNCTION_EX': 55, 'CALL_INTRINSIC_1': 56, 'CALL
_INTRINSIC_2': 57, 'CALL_KW': 58, 'COMPARE_OP': 59, 'CONTAINS_OP': 60, 'CONVERT_VALUE': 61, 'COPY': 62, 'COPY_FREE_VARS': 63, 'DELETE_ATTR': 64,
 'DELETE_DEREF': 65, 'DELETE_FAST': 66, 'DELETE_GLOBAL': 67, 'DELETE_NAME': 68, 'DICT_MERGE': 69, 'DICT_UPDATE': 70, 'ENTER_EXECUTOR': 71, 'EXTE
NDED_ARG': 72, 'FOR_ITER': 73, 'GET_AWAITABLE': 74, 'IMPORT_FROM': 75, 'IMPORT_NAME': 76, 'IS_OP': 77, 'JUMP_BACKWARD': 78, 'JUMP_BACKWARD_NO_IN
TERRUPT': 79, 'JUMP_FORWARD': 80, 'LIST_APPEND': 81, 'LIST_EXTEND': 82, 'LOAD_ATTR': 83, 'LOAD_CONST': 84, 'LOAD_DEREF': 85, 'LOAD_FAST': 86, 'L
OAD_FAST_AND_CLEAR': 87, 'LOAD_FAST_CHECK': 88, 'LOAD_FAST_LOAD_FAST': 89, 'LOAD_FROM_DICT_OR_DEREF': 90, 'LOAD_FROM_DICT_OR_GLOBALS': 91, 'LOAD
_GLOBAL': 92, 'LOAD_NAME': 93, 'LOAD_SUPER_ATTR': 94, 'MAKE_CELL': 95, 'MAP_ADD': 96, 'MATCH_CLASS': 97, 'POP_JUMP_IF_FALSE': 98, 'POP_JUMP_IF_N
ONE': 99, 'POP_JUMP_IF_NOT_NONE': 100, 'POP_JUMP_IF_TRUE': 101, 'RAISE_VARARGS': 102, 'RERAISE': 103, 'RETURN_CONST': 104, 'SEND': 105, 'SET_ADD
': 106, 'SET_FUNCTION_ATTRIBUTE': 107, 'SET_UPDATE': 108, 'STORE_ATTR': 109, 'STORE_DEREF': 110, 'STORE_FAST': 111, 'STORE_FAST_LOAD_FAST': 112,
 'STORE_FAST_STORE_FAST': 113, 'STORE_GLOBAL': 114, 'STORE_NAME': 115, 'SWAP': 116, 'UNPACK_EX': 117, 'UNPACK_SEQUENCE': 118, 'YIELD_VALUE': 119
, 'INSTRUMENTED_RESUME': 236, 'INSTRUMENTED_END_FOR': 237, 'INSTRUMENTED_END_SEND': 238, 'INSTRUMENTED_RETURN_VALUE': 239, 'INSTRUMENTED_RETURN_
CONST': 240, 'INSTRUMENTED_YIELD_VALUE': 241, 'INSTRUMENTED_LOAD_SUPER_ATTR': 242, 'INSTRUMENTED_FOR_ITER': 243, 'INSTRUMENTED_CALL': 244, 'INST
RUMENTED_CALL_KW': 245, 'INSTRUMENTED_CALL_FUNCTION_EX': 246, 'INSTRUMENTED_INSTRUCTION': 247, 'INSTRUMENTED_JUMP_FORWARD': 248, 'INSTRUMENTED_J
UMP_BACKWARD': 249, 'INSTRUMENTED_POP_JUMP_IF_TRUE': 250, 'INSTRUMENTED_POP_JUMP_IF_FALSE': 251, 'INSTRUMENTED_POP_JUMP_IF_NONE': 252, 'INSTRUME
NTED_POP_JUMP_IF_NOT_NONE': 253, 'JUMP': 256, 'JUMP_NO_INTERRUPT': 257, 'LOAD_CLOSURE': 258, 'LOAD_METHOD': 259, 'LOAD_SUPER_METHOD': 260, 'LOAD
_ZERO_SUPER_ATTR': 261, 'LOAD_ZERO_SUPER_METHOD': 262, 'POP_BLOCK': 263, 'SETUP_CLEANUP': 264, 'SETUP_FINALLY': 265, 'SETUP_WITH': 266, 'STORE_F
AST_MAYBE_NULL': 267}
```

הפקודה החדשה, שה-opcode שלה הוא 41, "נכנסה באמצע" וגרמה להזזה של רוב הפקודות שאחריה באחד. הדבר הזה גורם לכך שקוד Bytecode שקומפל לפני השינוי שלנו לא ירוץ כמו שצריך על ה-Interpreter שמכיל את השינוי שלנו, וגם להפך - קוד Bytecode שקומפל לאחר השינוי שלנו לא ירוץ כמו שצריך על Interpreter שלא מכיל את השינוי שלנו. גם אם הקוד כלל לא מכיל את פקודת ה-Bytecode החדשה שהוספנו.

ניתן לראות בפועל שכשלוקחים קובץ pyc. שקומפל לאחר השינוי שלנו ומנסים להריץ אותו על Interpreter ללא השינוי שלנו, מתקבלת השגיאה הבאה:

```
Command Prompt

C:\Users\Eli\Desktop\CPython_fork>PCbuild\amd64\python.exe -m compileall script.py
Compiling 'script.py'...

C:\Users\Eli\Desktop\CPython_fork>PCbuild\amd64\python.exe __pycache__\script.cpython-313.pyc
Hello World!

C:\Users\Eli\Desktop\CPython_fork>PCbuild_clean\amd64\python.exe __pycache__\script.cpython-313.pyc
RuntimeError: Bad magic number in .pyc file
```

היא מתקבלת בגלל שהיינו חכמים ועדכנו את שדה ה-magic_number כדי שבאמת לא יקרה המצב שאנחנו מנסים לגרום לו עכשיו - הרצה של פקודות Bytecode שלא תואמות לאיך שה-Interpreter מפרש אותן. כשמבטלים את עדכון שדה ה-magic_number לאחר השינוי ומאפשרים ל-Interpreter "רגיל" להריץ קובץ pyc. שקומפל אחרי השינוי, אז ה-Interpreter קורס מיד. אותו הדבר קורה גם כשנותנים ל-Interpreter לאחר השינוי להריץ קובץ pyc. שקומפל לפני השינוי.

רעיונות להמשך

כפי שהוספנו פקודת Bytecode חדשה שמבצעת את חישוב פונקציית Collatz, ניתן באופן דומה להוסיף פקודה חדשה שתבצע כל חישוב שנרצה, או פעולות "רגילות" כמו Bitwise XNOR למשל. ניתן גם להוסיף פקודה שמבצעת ביעילות חישוב XOR (או כל חישוב אחר) על שלושה ארגומנטים בבת אחת, במקום להשתמש בשתי פקודות Bytecode נפרדות שמבצעות חישוב על שני ארגומנטים בכל פעם:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> dis.dis("a ^ b ^ c")
  0           RESUME                   0

  1           LOAD_NAME                0 (a)
              LOAD_NAME                1 (b)
              BINARY_OP               12 (^)
              LOAD_NAME                2 (c)
              BINARY_OP               12 (^)
              RETURN_VALUE
>>>
```

בנוסף ניתן להוסיף פקודות שמשפיעות על ה-control flow, למשל פקודת switch-case, שתבצע את חישוב כתובת היעד באופן יעיל ותקפוץ אליו. או פקודת break2 שנמצאת בתוך לולאה מקוננת, ומאפשרת לצאת מהלולאה החיצונית. או פקודת continue2 שתפעל באופן דומה. גם כאן הגבול הוא הדימיון.

כמו כן, כפי שהוספנו לוגיקה חדשה לאובייקט שמייצג מספר, ניתן להשפיע על אובייקטים קיימים ולהוסיף להם יכולות חדשות. למשל באובייקט שמייצג מספרים, לאפשר חלוקה של אפס באפס כך שהתוצאה תהיה מוגדרת להיות אחד:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> 1//0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    1//0
    ~^^~
ZeroDivisionError: integer division or modulo by zero
>>> 0//0
1
>>>
```

או למשל לממש את הפעולה (~) not לאובייקט של מילון, שתעשה "reverse" למילון אם הוא חד-חד ערכי, כלומר תחליף בין המפתחות לערכים. גם כאן הגבול הוא הדימיון. אפשר להמשיך ולהיכנס לעוד פרטי מימוש של אובייקטים שונים, לראות אילו מהפונקציות חסרות בטבלאות הפונקציות ולחשוב על לוגיקה מעניינת שאפשר לממש שם.

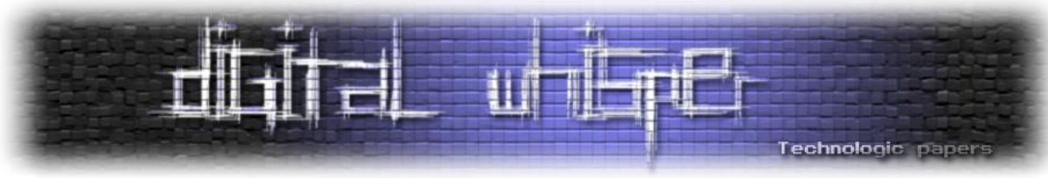
ניתן גם לעשות שינויים מצחיקים - למשל להחליף את ייצוג המחרוזת של אובייקט bool במילה Aladeen במקום במילים True או False:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> True, False
(Aladeen, Aladeen)
>>> 1+1==2, 1+1==3.5
(Aladeen, Aladeen)
>>> "You are HIV " + str(random.choice([True, False]))
'You are HIV Aladeen'
>>>
```

ואפשר גם להוסיף שינויים "זדוניים" שמשפיעים על תוצאות חישוב מסוימות. למשל להגדיר שאם בפעולת חיבור של שני מספרים יצאה התוצאה 21, התוצאה תשתנה להיות 42:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> 18+1
19
>>> 18+2
20
>>> 18+3
42
>>> 18+4
22
>>> 18+5
23
>>>
```

כמובן שניתן להכליל זאת לכל פעולה ולכל ערכים שנרצה.



סיכום

במאמר זה ראינו איך פקודות Bytecode מוגדרות ורצות בפועל, איך להוסיף פקודה חדשה, את ההשלכות של הוספה זו, ואיך להוסיף פונקציית Magic Method. למדנו על אובייקטים ושיטת הירושה שלהם ב-CPython, על סוגי הפונקציונליות שכל סוג אובייקט יכול להחליט איזה הוא בוחר לממש, תוך הוספה בפועל של פונקציונליות חדשה לאובייקט שמייצג מספרים.

בנוסף ראינו שניתן על ידי הסתכלות על אובייקטים בזיכרון, מבלי להסתכל בקוד המקור שלהם, להסיק מה הנתונים שהם מכילים ואיך לוודא זאת בעזרת מעקב אחרי המצביעים שבהם. ראינו גם איך בעזרת פונקציות מהספריה sys אפשר לקבל מושג על המימוש של אובייקטים, מתי מתבצעת בהם ריאלוקציה וכו'.

לבסוף חשבנו על רעיונות מעניינים נוספים לפקודות Bytecode חדשות וללוגיקות חדשות באובייקטים.

במאמר הבא בסדרה ניצור קוד Python שמשנה את ה-Bytecode של עצמו בצורה דינמית, ונמצא באגים במימוש של CPython. זה יהיה מגניב במיוחד כי הדברים האלו יעבדו על גרסאות Python רשמיות, ולא רק על גרסה שערכנו בה שינויים וקימפלנו בעצמנו. Stay tuned!

על המחבר

אני אלי קסקי, בן 29, אוהב לכתוב ולפתור אתגרי CTF, במיוחד בקטגוריות Reverse Engineering וקריפטוגרפיה, ואוהב לעשות דברים מגניבים עם קוד.

לפניות בכל נושא מוזמנים ליצור איתי קשר ב-elikaski94@gmail.com או ב-LinkedIn.