digital whisper
Technologic Papers

הקדמה ל-CPython

מאת דניאל אזולאי

הקדמה

אין היום כמעט אף אחד שמתעסק בטכנולוגיה ולא מכיר את שפת התיכנות פייתון. פייתון (או באנגלית Python) היא אחת משפות התכנות הפופלריות ביותר כיום וידועה בעיקר בגלל ה-syntax הפשוט, האקוסיסטים, הקהילה, והמגון הרב של החבילות שיש לה להציע, אשר כתוצאה מכך מושכות המון מתכנתים מתחומים שונים כגון: למידת מכונה, מדע נתונים, פיתוח צד שרת, אבטחת מידע, ועוד.

פייתון היא שפה "dynamically-typed" משמע שלמשתנים בפייתון אין טיפוס, ורק בהגדרת המשתנה (בזמן הריצה) מוגדר טיפוס בשונה משפות סטטיות כמו ג'אווה, שלכל משתנה יש טיפוס, ורק אותו משתנה יכול להכיל את הערך באותו טיפוס שהוגדר לו. בנוסף לכך, פייתון היא שפה garbage-collected.
ניתן להריץ קוד פייתון בכל מערכת הפעלה: Windows, MacOS, Linux, Unix או בקציור- cross platform.

מאמר זה הוא הקדמה למשימוש הפנימי של שפת פייתון עם CPython. את המאמר כתבתי על מנת לנסות להבין מה קורה מאוחרי השפה ולהעריך את הפשטות שפייתון ובכללי שפות HL מציעות לנו כיום.

מה זה CPython?

CPython משמש כ-Interpreter הדיפולטיבי וה"רישמי" של פייתון הכתוב בשפת C ומכיל את המימוש המלא של השפה. המטרה של הקומפיילר, או במקרה הזה, של CPython הוא לקחת שפה אחת ולהמיר אותה לשפה אחרת, אפשר לחשוב על CPython כמו מתרגם.

CPython מספקת לנו אבסטרקציה למרכבות של שפת C ושל המערכת הפעלה עצמה, הוא מאפשר לנו לכתוב קוד פייתון באופן פרודוקטיבי, ומוריד את הצורך לנהל זיכרון באופן ידני, כמובן שהאבסטרציה והפשטות שפייתון מציעה באה עם מחיר - ביצעויים נמוכים יותר משפות low-level.

ישנם ספריות בפייתון built-in הנתכבו ב-C, למשל כל מה שקשור ל-networking ו-filesystem נכתב ב-C, כיוון שהכל יותר קרוב ל"ברזלים".

מבנה התיקיות של CPython

על מנת שנוכל להבין איך פייתון עובדת, ואת תהליך ההרצה של פייתון, צריך קודם כל לדבר על מבנה התקיות ב-CPython ולנסות לסכם חלק ממה שהתיקיות האלו מכילות.

ישנה אפשרות לקמפל את CPython לוקאלית, הקומפילציה משתנה בין מערכות הפעלה שונות (יש את הפקודות לקימפול בקישור שצירפתי, ב-MacOS למשל ניתן לקמפל עם make) ולאחר סיום הקימפול שנמשך מספר דקות יהיה אפשר לראות קובץ בינארי חדש בשם python.exe. ניתן להוריד את CPython מה-Github הרישמי שלהם כאן

זהו מבנה התיקיות:

cpython/	
Doc	Source for the documentation
Grammar	The computer-readable language definition
Include	The C header files
Lib	Standard library modules written in Python
Mac	macOS support files
Misc	Miscellaneous files
Modules	Standard library modules written in C
Objects	Core types and the object model
Parser	The Python parser source code
PC	Windows build suppport files for older versions of Windows
PCBuild	Windows build support files
Programs	Source code for the 'python' executable and other binaries
Python	The CPython interpreter source code
Tools	Standalone tools useful for building or extending CPython
m4	Custom scripts to automate configuration of the makefile

Doc: התיעוד הרשימי של שפת פייתון, כל הווריציאות השונות של ה-interpreters בפייתון משתמשים באפיון זה והוא קריא לנו, לבני אדם (reStructured Text). ניתן למצוא קבצים ב-Doc/refrence קבצים המכילים מידע על כל הפיצרים שיש לפייתון, כולל פיצרים חדשים שיש לשפה.

דוגמא: with statement

מיקום: Doc/reference/compound_stmts.rst

הסבר: בקובץ זה רואים את איך ה-with בפייתון נראה מבחינת תחביר, ויותר למטע הסבר על איך ה-context manager עובד ביחד איתו - ברגע שיוצאים מהבלוק של ה-with או ברגע ש"הפונקצית קסם" __enter__ עושה raise ל-exception אז תהיה קריאה ל-__exit__ שם אפשר לדוגמא לסגור קובץ או סוקט פתוח.

כך הוא נראה:

```
427 The :keyword:`!with` statement
428 ==============================
429
430 .. index::
431    ! statement: with
432    keyword: as
433    single: as; with statement
434    single: , (comma); with statement
435    single: : (colon); compound statement
436
437 The :keyword:`with` statement is used to wrap the execution of a block with
438 methods defined by a context manager (see section :ref:`context-managers`).
439 This allows common :keyword:`try`...\ :keyword:`except`...\ :keyword:`finally`
440 usage patterns to be encapsulated for convenient reuse.
441
442 .. productionlist:: python-grammar
443    with_stmt: "with" ( "(" `with_stmt_contents` ","? ")" | `with_stmt_contents` ) ":" `suite`
444    with_stmt_contents: `with_item` ("," `with_item`)*
445    with_item: `expression` ["as" `target`]
```

שרשור של כמה with statements:

```
>>> with A() as a, B() as b, C() as c:
...     do_something(a, b, c)
```

```
512    with A() as a, B() as b:
513        SUITE
514
515 is semantically equivalent to::
516
517    with A() as a:
518        with B() as b:
519            SUITE
520
521 You can also write multi-item context managers in multiple lines if the items are surrounded by parentheses.
522 For example::
523
524    with (
525        A() as a,
526        B() as b,
527    ):
528        SUITE
```

דקדוק (Grammar):

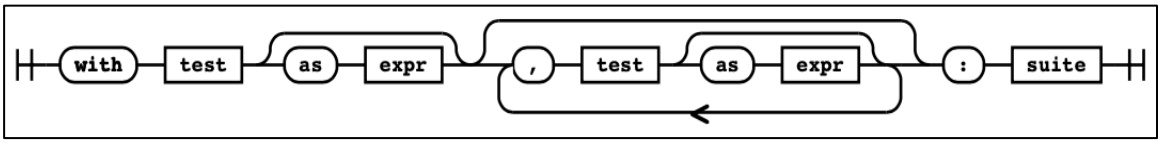
בתקייה זאת סה"כ 2 קבצים: python.gram ו-Tokens.

- python.gram: הקובץ כתוב ב-BNF (Backus-Naur From) והוא אחראי על המבנה של איך אנחנו כותבים בפייתון, אם עד עכשיו ראינו את התיעוד של ה-with statement כאן כבר יש סוג של מימוש בשפת BNF:

```
with_stmt: 'with' with_item (',' with_item)*  ':' suite
with_item: test ['as' expr]
```

תרגום קצר: מתחיל ב-with, הפסיק מייצג שיש אפשרות להוסיף יותר מ-with אחד, ה-suite מייצג את הקוד שיכנס בתוך הבלוק. בשורה השנייה ה-test מייצג מילות מפתח שהן לא חובה (as).

תיאור ויזואלי:



- Tokens: קובץ המכיל את השם של הטוקן ואת מה שהטוקן מייצג, קיים מגרסת פייתון 3.8, ה-tokens עוזרים בתהלך ה-parsing של הקוד שלנו באמצעות ה-tokenizer.

דוגמא ל-Tokens בפייתון:

```
 9  LPAR        '('
10  RPAR        ')'
11  LSQB        '['
12  RSQB        ']'
13  COLON       ':'
14  COMMA       ','
15  SEMI        ';'
16  PLUS        '+'
17  MINUS       '-'
18  STAR        '*'
19  SLASH       '/'
20  VBAR        '|'
```

אשתמש במודול tokenize המובנה בפייתון על מנת שנראה איך הקוד שכתבנו מתפרש ל-tokens, איך ה-tokenizer מפרש אותו:

```
❯ bat hello_world.py
File: hello_world.py
def greet():
    print("Hello, World")

danielazulay@Daniels-MacBook-Pro ~
❯ python3 -m tokenize hello_world.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,9:            NAME           'greet'
1,9-1,10:           OP             '('
1,10-1,11:          OP             ')'
1,11-1,12:          OP             ':'
1,12-1,13:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           OP             '('
2,10-2,24:          STRING         '"Hello, World"'
2,24-2,25:          OP             ')'
2,25-2,26:          NEWLINE        '\n'
3,0-3,0:            DEDENT         ''
3,0-3,0:            ENDMARKER      ''
```

בדוגמא רואים את המיספור של ה-tokens, העמודה הראשונה מייצגת את איפה שהטוקן נמצא, והעמודה השנייה מייצגת את השם של הטוקן, והערך של אותו טוקן בעמדה השלישית.

הברירת מחדל של CPython הוא UTF-8, ובסוף הפונקציה מתווסף שורה חדשה, מומלץ לחסוך את העובדה ל-CPython ולהוסיף שורה חדשה אחרי כל פונקציה או בכללי להשתמש בלינטרים למינהם (pylint) שיעזרו לנו בניתוח סטטי של הקוד. ה-ENDMARKER מייצג את הסוף של הקובץ.

Parser: אחרי שהסברתי בצורה כללית את תקיית ה-Grammar וה-Tokens נדבר על ה-Parser שיש בפייתון. בתקייה יש את pegen והוא ה-parser generator ש-CPython משתמש בו על מנת לעשות parsing לקוד שלנו ולבדוק את תקינות ה-Grammar (ה-Grammar של פייתון כמובן) על ידי חוקים מסויימים שיש ב-pegn (rules names). pegn כתוב בפייתון והוא נוצר בתקיית ה-peg_generator. הוא ממוקם ב:

Tools/peg_generator/pegen/parser_generator.py

```
class ParserGenerator:

    callmakervisitor: GrammarVisitor

    def __init__(self, grammar: Grammar, tokens: Set[str], file: Optional[IO[Text]]):
        self.grammar = grammar
        self.tokens = tokens
```

```
def validate_rule_names(self) -> None:
    for rule in self.rules:
        if rule.startswith("_"):
            raise GrammarError(f"Rule names cannot start with underscore: '{rule}'")
```

תרומות ופיצרים ל-CPython:

דרך נוספת לתרום ל-CPython (חוץ מללכת לדף גיטהאב) היא ע"י חיפוש ב-Source code של CPython. חוץ מתרומות ניתן גם למצוא מימושים שדורשים שיפור אם מחפשים סמיילי עצוב/TODOs:

```
danielazulay@Daniels-MacBook-Pro  ~/tools/cpython  main
❯ grep -r "TODO" *
Doc/tools/rstlint.py:# TODO: - wrong versions in versionadded/chang
Doc/tools/extensions/pyspecific.py:# TODO: Use sphinx.domains.pytho
Doc/install/index.rst:.. TODO: Fill in XXX comments
Doc/library/zoneinfo.rst:    .. TODO: Add "See `cache_behavior`_" r
Doc/library/zoneinfo.rst:    .. TODO: Add "See `cache_behavior`_" r
Doc/whatsnew/3.8.rst:.. TODO: Pablo will sprint on docs at PyCon US
Doc/extending/embedding.rst:.. TODO: threads, code examples do not
Doc/howto/regex.rst:.. TODO:
Include/internal/pycore_runtime.h:    /* TODO: Given interp_main, i
Include/internal/pycore_initconfig.h:extern int _Py_global_config_i
s ship.
Lib/zoneinfo/_zoneinfo.py:    # TODO: These are not actually epoch
Lib/nntplib.py:# TODO:
Lib/test/test_poplib.py:                # TODO: SSLError does not e
Lib/test/test_gettext.py:# TODO:
Lib/test/_test_multiprocessing.py:    # TODO: add more tests for de
Lib/test/_test_multiprocessing.py:        # TODO: fix https://bugs.
Lib/test/test_peg_generator/test_first_sets.py:        # TODO: Vali
Lib/test/test_nntplib.py:# TODO:
Lib/test/test_runpy.py:# TODO: Use self.addCleanup to get rid of a
```

```
danielazulay@Daniels-MacBook-Pro  ~/tools/cpython  main
❯ grep -r ":-(" *
Include/modsupport.h:   11-Oct-1996  GvR     renamed Py_Ellipses to
Include/modsupport.h:   7-Nov-1995   GvR     Keyword arguments (sho
Lib/test/test_statistics.py:        # we differ by a very slight am
Lib/test/test_statistics.py:        # causes me headaches :-(
Lib/test/test_bdb.py:        # test_pdb does not reset Breakpoint c
Lib/test/test_zipapp.py:        # once for each file written :-(
Lib/test/support/__init__.py:        # Windows implementation: don'
Lib/test/test_asyncio/test_unix_events.py:        # This is a bit o
Lib/plistlib.py:_dateParser = re.compile(r"(?P<year>\d\d\d\d)(?:-(?
)?)?)?)?)?Z", re.ASCII)
Lib/idlelib/TODO.txt:- Grail uses a module called History and so do
Lib/idlelib/ChangeLog:  (Not much has changed :-( )
Lib/idlelib/tooltip.py:        # event and it reappears, and so on
Lib/aifc.py:            #DEBUG: SGI's soundeditor produces a bad si
Misc/HISTORY:   (Not much has changed :-( )
Misc/HISTORY:eons ago, but I never regenrated glmodule.c :-( )
Misc/HISTORY:recomputed after such assignments (but not for derived
Misc/HISTORY::-( ).
Misc/HISTORY:same time, it is documented...:-( ).
Misc/HISTORY:on the Sun the provided strtod.c dumps core :-(
```

עוד מימושים של פייתון

- IronPython - מימוש של שפת פייתון ל-NET. נכתב ב-C#
- Jython - מימוש של פייתון, רץ על ה-JVM של ג'אווה, הקוד פייתון שלנו מתקמפל ל-Bytecode של ככה שנוכל לרוץ על ה-JVM
- Nuitka - מדובר בקומפיילר של פייתון, שנכתב בעצמו בפייתון. אם אי פעם תרצו לקבל קובץ הרצה של הקוד פייתון שלכם, זה המקום
- PyPy - מימוש של שפת פייתון התומך ב-JIT, כמעט פי 5 יותר מהיר מ-CPython.

אלו המימושים הנפוצים ביותר לפייתון, ישנם כמובן עוד מימושים שונים, אבל אין בהם כבר תמיכה, וכתוצאה מכך גם הגרסא של השפה עצמה לא עדכנית ונוטה לכיוון Python2 שפחות מומלץ.

שינוי מגבלת ב-CPython:

אפשר לשנות פונקציונליות של CPython עם פייתון. לדוגמא, המגבלה של רקורסיה בפייתון היא 1000, 10^3.

ניתן לשנות זאת על ידי sys.setrecursionlimit(N). המגבלה עצמה קיימת על מנת למנוע stack overflow:

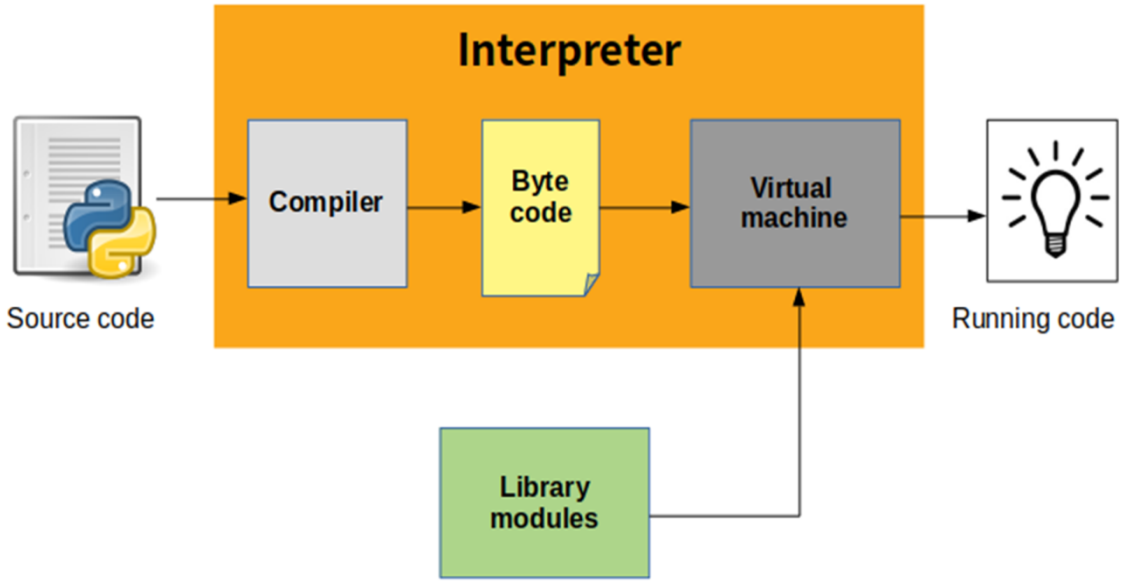
```
>>> import sys
>>>
>>> print("maximum:", sys.getrecursionlimit())
maximum: 10000
>>> sys.setrecursionlimit(10**4)
>>> print("new maximum:", sys.getrecursionlimit())
new maximum: 10000
```

הרצה בפייתון

הקוד שאנחנו כותבים בפייתון עובר מספר תהליכים עד שהתוכנית שלנו תוכל לרוץ, גם המהדר (compalier) והמפרש (interpreter) נכנסים בסיפור, דבר המשפיעה כמובן על הביצועים של השפה. באופן כללי, ההרצה מכילה 3 שלבים עקריים, שבכול אחד מן השלבים יש תת שלבים, אסכם את ה-3 שלבים:

1. **שלב ה-Initialization:** בשלב זה, CPython מאתחל את כל מבני הנתונים על מנת להריץ תוכנית בפייתון כולל built-in types כגון: exceptions, classes, numerics, sequences ואלוקצית זיכרון
2. **שלב הקימפול או שלב ה"תרגום":** CPython מבצע פרסור ובונה Abstract Syntax Tree (AST) ומגנרנט לנו את ה-Bytecode הנוצר על ידי ה-AST, כולל מגוון אופטימיזציות שונות
3. **שלה ההידור:** פה רץ ה-Bytecode על ידי המהדר, אפשר לחשוב על CPython כ-VM שמריץ את ה-Bytecode ככה שנהוג להסתכל על CPython כמפרש ומהדר.

תרשים כללי המסכם את השלבים:



Abstract Syntax Tree:

ה-AST יכול לעזור לנו לנתח את ה-source code, הוא בנוי על מבנה נתונים מסוג tree המכיל nodes והוא מציג את הקוד שלנו במבנה של עץ, בכל node יש statements, variables או אפילו expressions. בסופו של דבר המטרה היא לקחת את העץ הזה ולהמיר אותו ל-Bytecode על מנת שהמחשב יכול להריץ את הקוד שלנו.

כמה שימושים נופוצים ב-AST:

- כתיבה של Custom Linting עם ספריית AST ו-flake8
- code formatters כמו black

- הפרויקט שכל מי שכתב טסטים מכיר; pytest משתמש ב-AST על מנת להציג שגיאות ברגע שאחד מן ה-assertions נכשל בטסטים

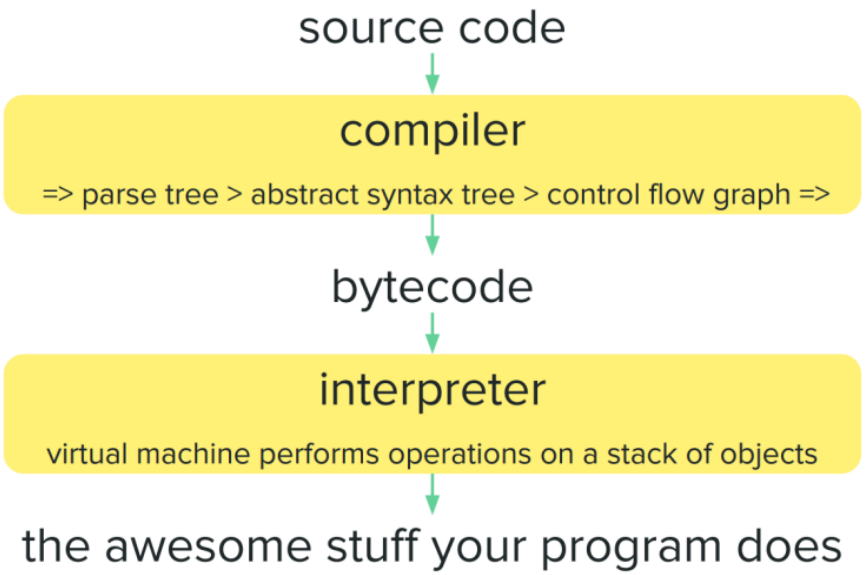
```
❯ bat example.py
        File: example.py

1
2       def say_hello():
3           return "hello"

danielazulay@Daniels-MacBook-Pro  ~/tools
❯ python3 -m ast example.py
Module(
   body=[
      FunctionDef(
         name='say_hello',
         args=arguments(
            posonlyargs=[],
            args=[],
            kwonlyargs=[],
            kw_defaults=[],
            defaults=[]),
         body=[
            Return(
               value=Constant(value='hello'))],
         decorator_list=[])],
   type_ignores=[])
```

איפה ה-AST נכנס בתהליך ההרצה?

תרשים ההרצה של פייתון שהוצג למעלה לא מראה את התת משימות שוקראות ב-CPython, לכל אחד מן השלבים יש תת שלבים, כמובן שלא נכנס לכל תת משימה, כי נראה שלכל תת משימה אצטרך לכתוב מאמר בנפרד. AST נכנס בשלב של ה-compiler, שממיר אותו בסופו של דבר ל-bytecode:



Bytecode בפייתון

Bytecode היא שפת הביינים של פייתון. ה-Bytecode הוא מה ש-CPython "רואה" בזמן שאנחנו מרצים את הקוד שלנו, אפשר לחשוב על Bytecode כמו Machine code ל-interpreter. ה-Bytecode בסופו של דבר משמש כסט של פעולות.

אם אי פעם ראיתם תקיית __pycache__ או קבצים עם הסיומות .pyc, מדובר בקובץ המשמש כסוג של זיכרון מוטמן (cache) ש-Bytecode משתמש בו ככה שבמידה ואנחנו נריץ את אותו הקוד בפעם השנייה ולא נבצע בו שינוי, בפעם השנייה הוא ירוץ יותר מהר כיוון שאין צורך בלבצע את כל תהליך הקומפילציה מחדש.

Bytecode אינו קריא לבני אדם, אז איך באמת אפשר לקרוא אותו?

dis - bytecode disassembler

יש ספרייה בפייתון בשם dis שמאפשרת לראות את ה-bytecode בפורמט של פעולות שנוצרו, ה-dis מתרגם לנו את ה-Bytecode ככה שנוכל לקרוא אותו בסופו של דבר (מזכיר תהליך dissasembly) במקום להסתכל על ה-Bytecode עצמו שבשבילנו אין לו משמעות:

```
>>> import dis
>>>
>>> def func_example():
...     x = 1
...     y = 1
...     return x + y
...
>>>
>>> dis.dis(func_example)
  2           0 LOAD_CONST               1 (1)
              2 STORE_FAST               0 (x)

  3           4 LOAD_CONST               1 (1)
              6 STORE_FAST               1 (y)

  4           8 LOAD_FAST                0 (x)
             10 LOAD_FAST                1 (y)
             12 BINARY_ADD
             14 RETURN_VALUE
```

עמודה ראשונה: מספר שורה

עמודה שנייה: offest ב-bytes

עמדה שלישית: שם הפעולה

עמדה רביעית: אינדקס הארגומנט

עמדה חמישית: הערך של הארגומנט

פעולות ב-Bytecode

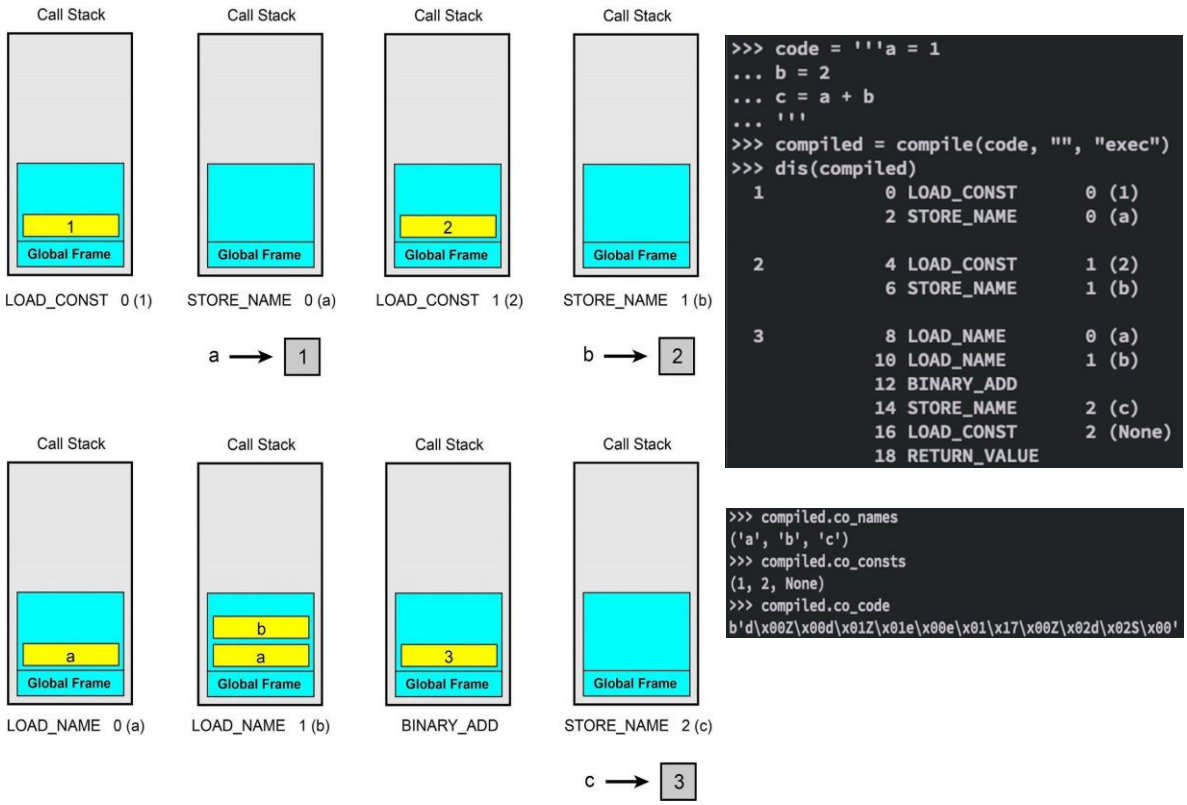
בשביל להבין את הפעולת ב-Bytecode צריך קודם כל להבין איך CPython מנהל את המשתנים והזיכרון. הכל ב-Python הוא object (מבני נתונים, intergers, strings ו-Classes) וכל ערכי ה-objects ב-Python נשמרים ב-heap, בזמן שה-memory address שלהם נשמר ב-stack.

CPython משתמש ב-stack מבוסס Virtual Machine, יצירת משתנים בפייתון יוצרת **Refrence** מה-stack ל-heap. ה-stack הוא מבנה נתונים מסוג LIFO (Last In First Out), ויש לו 2 פעולות: push & pop.

- push: מוסיף איבר ל-stack
- pop: מוציא את האיבר האחרון שנכנס ל-stack

קריאה וכתיבה ב-stack מהירים מאוד עם יעילות O(1). CPython משתמש ב-3 סוגים שונים של stack:

1. call stack: המבנה הנתונים הראשי בזמן הרצה של קוד בפייתון, כל איבר ב-stack נקרא "frame" וכל frame מייצג קריאה לפונקציה/מבציע לאיבר, ככה שהאיבר הראשון שנכנס ל-stack הוא נקודת ההתחלה של התוכנית שלנו
2. evaluation stack: לכל frame כזה הנמצא ב-call stack יש evaluation stack שהמטרה שלו הוא לבצע execution של פונקציה, בנוסף CPython שומר את הפרמטרים הדורשים לפעולה מסויימת בפייתון, כולל התוצאה של אותן פעולות ב-stack זה.
3. block stack: פייתון משתמשת ב-stack הזה על מנת לשמור Refrence למבנים בקוד שלנו כמו: try/execpt, לולאות ו-with blocks. ברגע שיצאנו מלולאה או מבלוק של try/except יתבצע pop ב-stack, ובאותה מידה ברגע שנכנסו לבלוק try/except מתבצע push ל-stack. פייתון משתמשת ב-stack הזה גם בשביל לתמוך בפעולות כמו break על מנת שנוכל לצאת מלולאה.



התמונה השמלאית מייצגת את ה-stack ברגע שוקראים לפונקציה שמריצים את הקוד. התמונה הימינת מייצגת את ה-output של dis ואת ה-attributes שבאים איתו (משתנים, ערכים של משתנים וה-Bytecode עצמו).

הסוגים הם:

- LOAD_CONST: פעולה שעושה push ל-stack, זאת אומרת שעושים push ל- co_consts[0] (שווה ל-1) ולאחר מכן הפעולה חוזרת על עצמה למשתנה b.
- STORE_NAME: מתבצע pop לטופ של ה-stack, במקרה הזה ל-1 ושומר אותו באובייקט ("refrence") ל-1, אותו הדבר למשתנה b. עכשיו יש לנו 2 אובייקטים סה"כ.
- BINARY_ADD: עושה pop ל-2 משתנים הראשונים ב-stack ומבצע פעולת חיבור, לאחר מכן עושה push לתוצאה (c) וקורא ל-STORE_NAME שוב.
- RETURN_VALUE: מחזיר את ה-frame האחרון ב-stack. מכיוון שה-stack ריק, יוחזר None (זאת גם הסיבה שמוחזר None ברגע שקוראים לפונקציה שלא מחזירה כלום).

כל הפעולות ש-CPython מבצע כמו BINARY_ADD, STORE_NAME הן מתורגת על ידי switch-case אחד גדול הכתוב ב-C, שכל case מטפל בפעולה אחרת:

```
switch (opcode) {
    TARGET(NOP) {
        DISPATCH();
    }

    TARGET(RESUME) {
        _PyCode_Warmup(frame->f_code);
        JUMP_TO_INSTRUCTION(RESUME_QUICK);
    }

    TARGET(RESUME_QUICK) {
        PREDICTED(RESUME_QUICK);
        assert(tstate->cframe == &cframe);
        assert(frame == cframe.current_frame);
        if (_Py_atomic_load_relaxed_int32(eval_breaker) && oparg < 2) {
            goto handle_eval_breaker;
        }
        DISPATCH();
    }
```

איך LOAD_CONST נראה?

```
TARGET(LOAD_CONST) {
    PREDICTED(LOAD_CONST);
    PyObject *value = GETITEM(consts, oparg);
    Py_INCREF(value);
    PUSH(value);
    DISPATCH();
}
```

הכל ביחיד - instaviz

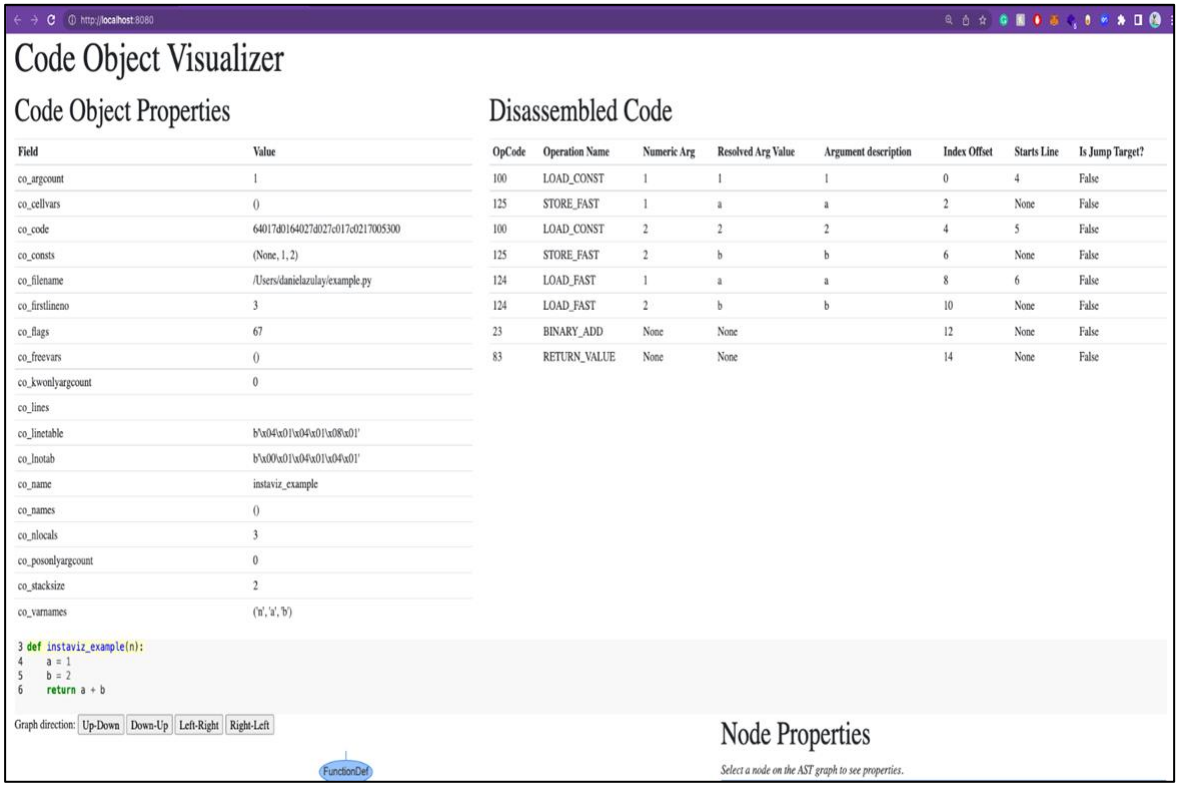
אחרי שהסברתי בקצרה על ה-Bytecode, Abstract Syntax Tree ו-bytecode disassembler כאן אראה איך הכל מתחבר ביחד בצורה ויזואלית וקלה להבנה.

ההמכשה הויזואלית תתבצע באמצעות ספריה בשם instaviz, זה ספריית open source שהמטרה שלה היא לקבל תרשים ויזואלי של הקוד שלנו, מאז שהכרתי את הספרייה הזאת התחלתי להתמש בה בזמן שאני עושה refactoring לקוד, ובזמן שאני מנסה לשפר יעילות אפשר לתעד את התרשימים ולראות את ה-Bytecode לפני והאחרי. מה שנוח בכלי הוא שהכל במקום אחד:Dissasembled Code, Code Object Properties ותרשים בסגנון גרף שהוא מאוד קל להבנה.

ניתן להתקין את הספרייה על ידי הפקודה:

```
pip3 install instaviz
```

Disassembled Code, Object Properties והקוד עצמו:

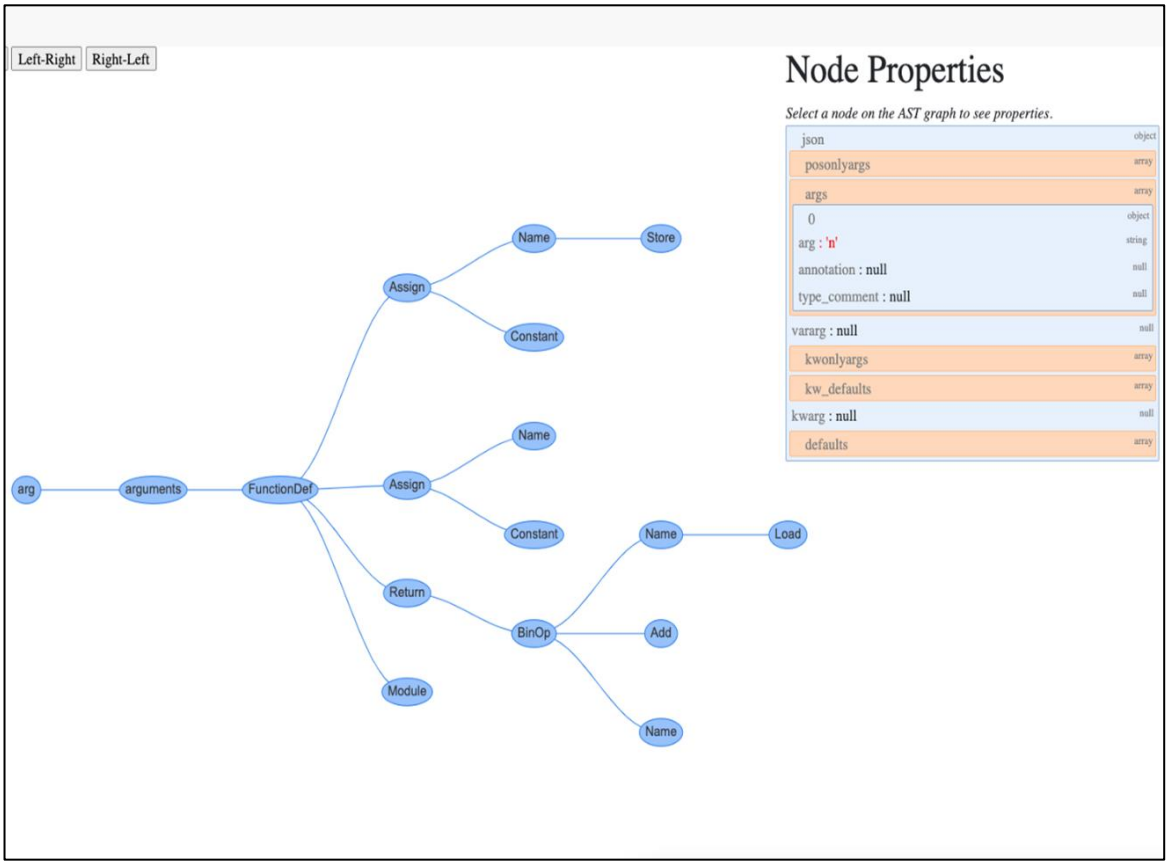


Field	Value
co_argcount	1
co_cellvars	()
co_code	64017d0164027d027c017c0217005300
co_consts	(None, 1, 2)
co_filename	/Users/danielazulay/example.py
co_firstlineno	3
co_flags	67
co_freevars	()
co_kwonlyargcount	0
co_lines	
co_linetable	b'\x04\x01\x04\x01\x08\x01'
co_lnotab	b'\x00\x01\x04\x01\x04\x01'
co_name	instaviz_example
co_names	()
co_nlocals	3
co_posonlyargcount	0
co_stacksize	2
co_varnames	('n', 'a', 'b')

OpCode	Operation Name	Numeric Arg	Resolved Arg Value	Argument description	Index Offset	Starts Line	Is Jump Target?
100	LOAD_CONST	1	1	1	0	4	False
125	STORE_FAST	1	a	a	2	None	False
100	LOAD_CONST	2	2	2	4	5	False
125	STORE_FAST	2	b	b	6	None	False
124	LOAD_FAST	1	a	a	8	6	False
124	LOAD_FAST	2	b	b	10	None	False
23	BINARY_ADD	None	None		12	None	False
83	RETURN_VALUE	None	None		14	None	False

```
3 def instaviz_example(n):
4     a = 1
5     b = 2
6     return a + b
```

הגרף (bytecode operations, arguments, assignments, etc):



על Performance ו-Bytecode

פייתון היא שפה שגם ככה מאוד איטית, מבני נתונים שהם מתאמים לפעולה מסויימת ב-C לא יהיו מתאים לאותו פעולה ב-Python ולכן חשוב לדעת את ה-best practices שיש בפייתון על מנת להפיק מהשפה את המיטב כדי לקבל ביצועים טובים יותר.

אחד מן היתרונות הבולטים של שימוש ב-dis הוא לראות את הפעולות ש-CPython הולך לבצע, הפעולת ב-Bytecode יכולות לעזור לנו לשפר ביצועים בקוד שלנו ולהבין את ה-execution.

לדוגמא השימוש ב-dict לעומת {} או בין list ל-[].

כשקוראים ל-dict מתבצע קריאה לפונקציה CALL_FUNCTION הגורמת לביצוע פעולת push ו-pop ב-stack (הכוונה ב-push זה ה-key וה-value של המילון שנדחפים ל-heap), קריאה ל-counstructor של dict על מנת ליצור אובייקט חדש, וכו. לעומת זאת, קריאה ל-{} תיהיה הרבה יותר קלילה, CPython לא קורא ל-counstructor על מנת ליצור את האוביקט ולא צריך לאתחל מחדש את המילון כמו ב-dict.

ברגע שאנחנו חוסכים בקריאה ל-dict ומתמשים ב-{} הביצועים יהיו משמעותית יותר מהירים.

ה-Bytecode ל-{} יותר קצר:

```
>>> dis('{}')
  1           0 BUILD_MAP                0
              2 RETURN_VALUE
>>> dis('dict()')
  1           0 LOAD_NAME                0 (dict)
              2 CALL_FUNCTION            0
              4 RETURN_VALUE
```

CPython יוצרת אובייקט חדש בכל קריאה ל-dict, בשונה מ-{}:

```
>>> for _ in range(3):
...     print(id({}))
...
4467245120
4467245120
4467245120
>>>
>>> for _ in range(3):
...     print(id(dict()))
...
4470482368
4470482624
4470482432
```

פי 3 יותר מהיר? (פייתון 3.10)

```
>>> timeit.timeit('dict()', number=10000000)
0.5359546210020198
>>> timeit.timeit('{}', number=10000000)
0.18101146800472634
```

אפשר לתת עוד המון דוגמאות לכמה best practices חשובים בפייתון, למשל עם ובלי list comprehension, תנאים (if x is True, if x), שימוש ב-built in funtions כגון len() וכו וכו.

ממליץ בחום על ההרצה הזאת מ-pycon שנותת עוד דוגמאות ל-best practices וביצועים בפייתון.

סיכום

במאמר זה הסברתי באופן כללי על איך Python עובדת מאחורי הקלעים. ראינו את שלבי ההרצה בפייתון, ואיך CPytohn קשור לכל הסיפור, סקרנו מונחים בסיסים כגון AST ו-Bytecode ואיפה הם לוקחים חלק בהרצת הקוד.

נתתי הצצה קטנה למה יש בתוך CPython, ראינו את מבנה התיקיות שלו ונגענו במימושים פנימיים של השפה, איך הקוד שלנו מתפרש עד ל-Bytecode, תחביר נכון ב-Python ואיך CPython מנהל את הזיכרון. עשינו "disassembly" ל-Bytecode על מנת להבין איך CPython רואה את הקוד שלנו ואפילו תיאור וויזואלי של איך תוכנית בפייתון נראית.

לבסוף דיברתי בקצרה על ביצועים בפייתון ונתתי כיוון על איך אפשר לשפר ועד כמה זה חשוב ללכת על ה-best practices.

על הכתוב

דניאל אזולאי, בן 18, חובב צד שרת, אבט"מ, ופייתון.

ניתן לפנות אלי בטוויטר או במייל:

https://twitter.com/danielazulayy

serializator@gmail.com

קישורים

https://github.com/tonybaloney/instaviz

https://ep2016.europython.eu//media/conference/slides/exploring-python-bytecode.pdf

https://devguide.python.org/internals/index.html

https://towardsdatascience.com/understanding-python-bytecode-e7edaae8734d