Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Wydział Informatyki

Maciej Tomczuk

nr albumu: 24022

kierunek studiów: Informatyka

specjalność: Inżynieria oprogramowania

forma studiów*: niestacjonarne*

ANALIZA I PORÓWNANIE WYBRANYCH WŁAŚCIWOŚCI JĘZYKA PYTHON NA POZIOMIE KODU POŚREDNIEGO

ANALYSIS AND COMPARISON OF VARIOUS FEATURES OF PYTHON LANGUAGE, PERFORMANCE AND BYTECODE INSPECTION IN APPLICATION

praca dyplomowa magisterska

napisana pod kierunkiem:

dr inż. Piotra Błaszyńskiego

Katedra inżynierii Oprogramowania

Data wydania tematu pracy: 15.06.2016

Data złożenia pracy:

Szczecin, 2017

OŚWIADCZENIE

AUTORA PRACY DYPLOMOWEJ

Oświadczam, że praca dyplomowa magisterska pt.

"Analiza i porównanie wybranych właściwości języka Python na poziomie kodu pośredniego"

napisana pod kierunkiem:

dr inż. Piotra Błaszyńskiego

jest w całości moim samodzielnym autorskim opracowaniem, sporządzonym przy wykorzystaniu wykazanej w pracy literatury przedmiotu i materiałów źródłowych.

Złożona w dziekanacie **Wydziału Informatyki**

treść mojej pracy dyplomowej w formie elektronicznej jest zgodna z treścią w formie pisemnej.

Oświadczam ponadto, że złożona w dziekanacie praca dyplomowa ani jej fragmenty nie były wcześniej przedmiotem procedur procesu dyplomowania związanych z uzyskaniem tytułu zawodowego w uczelniach wyższych.

.....…
podpis dyplomanta

Szczecin, dn. .....

Spis treści

Streszczenie

Głównym celem pracy było porównanie kodów pośrednich dwóch wersji języka Python: 2.7.12 oraz 3.5.2 oraz wyodrębnienie jego wybranych cech, rzutujących na wykonanie tego kodu.

Praca składa się z sześciu rozdziałów. W pierwszym autor opisuje mechanikę działania języka Python oraz wprowadza czytelnika w sposób wykonywania kodu pośredniego w wirtualnej maszynie tego języka. W drugim nacisk kładziony jest na sam kod pośredni, zawartość plików *.pyc*, analizowany jest także przykładowy program. W trzeciej części pracy przedstawiane są różnice pomiędzy dwoma wybranymi wersjami języka Python: wybrane nowe i zmienione elementy, a także nowe, zmienione oraz usunięte instrukcje kodu pośredniego w dwóch wcześniej wspomnianych wersjach interpretera. W rozdziale czwartym opisywane są wybrane cechy języka Python, pogarszające jego wydajność. Analizowane są cechy takie jak dynamiczne typowane, zliczanie referencji czy opakowywanie liczb i struktur danych, które znacząco wpływają na nieefektywne wykonanie kodu bajtowego przez maszynę wirtualną. W rozdziale piątym autor skupia się na opisie projektów przyspieszających wykonanie programów napisanych w języku Python, a także sposobach optymalizacji wykonania kodu pośredniego. W ostatniej części opisywana jest aplikacja porównująca kody pośrednie języka Python, użyte narzędzia i biblioteki oraz wnioski i dalsze plany rozwoju napisanej aplikacji.

Słowa kluczowe

Python, kod pośredni, disassembler

Abstract

The main purpose of the work was to compare the bytecode of two Python language versions: 2.7.12 and 3.5.2. The target was also to distinguish its selected features that affect the execution of this code.

The work consists of six chapters. In the first one author describes the mechanics of the Python language and introduces the reader into the way of executing the intermediate code in the Python virtual machine. In the second one, the emphasis is put on the intermediate code itself, the contents of the *.pyc* files, the sample program is also analyzed. The third part of the paper presents differences between two selected versions of Python language: selected new and changed elements, as well as new, changed and removed bytecode instructions in the previously mentioned versions of the interpreter. The fourth chapter describes selected features of the Python language that make its efficiency worse. Features such as dynamic typing, reference counting and packaging of numbers and data structures are analyzed, which significantly affect bytecode execution by the virtual machine. In the fifth chapter, the author focuses on the description of projects accelerating the execution of programs written in Python, as well as ways to optimize the implementation of the intermediate code. The last part covers application comparing Python language bytecode, tools and libraries used, as well as applications and further development plans for a written application.

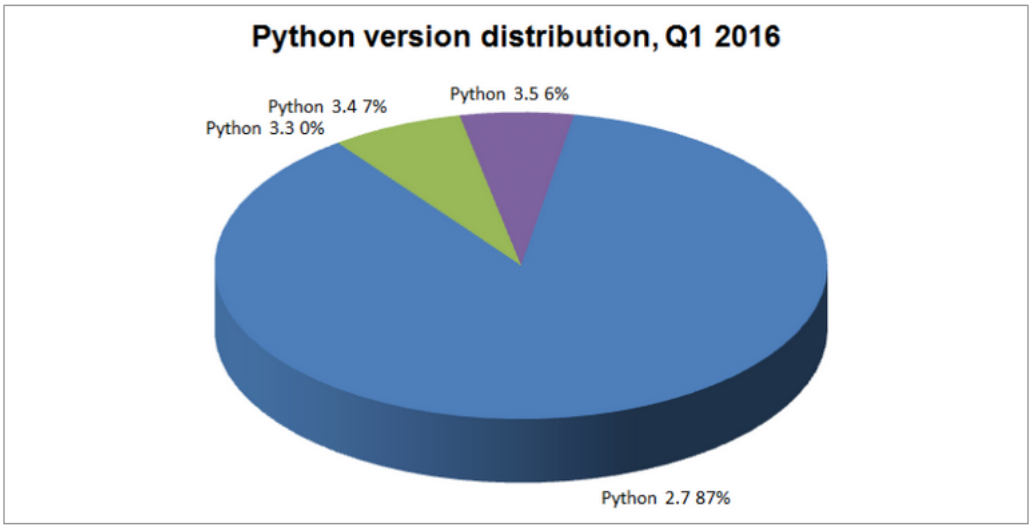
Keywords

Python, bytecode, disassembler

Wprowadzenie

Python jest interpretowanym, dynamicznie typowanym obiektowym językiem programowania stworzonym przez Guido van Rossuma w 1990 roku. Posiada w pełni dynamiczny system typów i automatyczne zarządzanie pamięcią, jest zatem podobny do takich języków, jak Tcl, Perl, Scheme czy Ruby. Python rozwijany jest jako projekt Open Source, zarządzany przez organizację non-profit Python Software Fundation. Python jest aktywnie rozwijany i posiada szerokie grono użytkowników na całym świecie,a jego prosta i czytelna składnia ułatwia utrzymywanie, używanie i rozumienie kodu. Język ten posiada duże wsparcie dla modułów i pakietów, co zachęca do modularyzacji programów i ponownego użycia kodu. Aplikacje napisane w Pythonie działają pod wieloma systemami takimi jak Windows, Linux/Unix czy Mac OS X. Możliwe jest także pisanie aplikacji na systemy mobilne, takie jak Android. Dostępne są także implementacje Pythona w Javie (Jython) i .NET (IronPython) działające wszędzie tam, gdzie dostępne są te platformy. Istnieją także implementacje języka Python napisane w samym Pythonie (PyPy) używające kompilatora typu JIT (ang. *Just-in-Time compiler*), której interpreter został napisany w języku RPython będący podzbiorem Pythona. Rozwijane są także kompilatory optymalizujące, takie jak *Numba*, ulepszające wyniki czasowe programów napisanych w Pythonie.

Język Python rozwija się w ogromnym tempie, a liczba jego użytkowników rośnie z dnia na dzień. Na dzień dzisiejszy istnieją dwa największe odłamy tego języka: wersja *2.x* oraz *3.x*. Wersja 3.x nie jest jednak kompatybilna wstecznie, a różnice pomiędzy dwoma odłamami są całkiem spore. Mimo, że wersja 2.7 (aktualnie najpopularniejsza) będzie rozwijana do 2020 roku, w pierwszym kwartale 2016 roku starsza wersja interpretera deklasowała nowsze wersje:



Rysunek 1. Wykres ukazujący podział rynku języka Python na poszczególne wersje w pierwszym kwartale 2016 roku

Źródło:
https://www.linkedin.com/pulse/software-stacks-market-share-first-quarter-2016-alex-anikin

Statystyki pokazują jednak, że Python w wersji 3.x z roku na rok zyskuje coraz większą popularność.

W tej pracy autor skupi się na analizie pośredniej formy kodu przekazywanej do interpretera języka Python - kodu bajtowego. Jako, że różnice składniowe we wspomnianych wersjach Pythona sa liczne, autor postarał się wybrać najciekawsze przykłady ukazujące w jaki sposób kod pośredni może wpływać na wydajność napisanego programu, w jakim stopniu na nią wpływa oraz przeprowadzić analizę porównawczą dwóch wersji interpretera języka Python.

Analiza kodu pośredniego jest używana między innymi w narzędziach takich, jak *coverage* (moduł służący do sprawdzania pokrycia kodu przez testy) czy narzędziach służących do analizy kodu pod kątem plagiatów. Praca ta będzie skupiać się na różnicach kodu bajtowego pomiędzy dwoma popularnymi wersjami języka Python: 2.7.12 oraz 3.5.2.

Aplikacja napisana na potrzeby tej pracy opiera się o dwie wersje języka Python: 2.7.12 oraz 3.5.2. Jej działanie opiera się o generowanie kodu bajtowego dla wspomnianych wersji interpretera języka Python z pomocą wbudowanego modułu *dis* z plików z rozszerzeniem *.pyc*, a także tworzeniu grafów przepływu sterowania (ang. *Control Flow Graph, CFG*) z pomocą modułu *flare bytecode graph*. Możliwe jest także porównanie czasów wykonania poszczególnych programów, co zostało osiąganięte dzięki wbudowanemu w system Linux programowi *time*.

Rozdział 1

Zasada działania języka Python

1.1. Wstęp

Niektóre języki programowania, takie jak C++, są tłumaczone na intrukcje zrozumiałe dla sprzętu na którym operuje program. Natomiast inne języki programowania (na przykład Java lub Python) są przetwarzane w sposób zrozumiały dla wielu architektur: mowa tutaj o maszynach wirtualnych, które mogą być napisane w języku niskopoziomowym, takim jak C (kompatybilny z większością architektur) lub w języku sprzętowym urządzenia dla uzyskania maksymalnej prędkości wykonywania kodu. W ten sposób możliwe jest uruchomienie programów napisanych w tych językach na każdej architekturze, w której zaimplementowana jest maszyna wirtualna danego języka.

Kompilacja ma jedną niewątpliwą zaletę: produkuje programy znacznie szybsze od tych wykonywanych przez interpretery / maszyny wirtualne. Z drugiej strony, stworzenie kompilatora dla danej architektury jest dużym przedsięwzięciem, a kompilacja programów może być czasochłonna. Maszyny wirtualne są o wiele prostsze do przeniesienia na nową architekturę sprzętową; potrzebny jest tylko jeden program wykonujący kod pośredni danego języka. Niestety, to podejście powoduje pogorszenie parametrów czasowych wykonywanych programów, niezależnie od tego jak dobrze zostanie napisany interpreter; zazwyczaj programy operujące na maszynach wirtualnych wykonują się wolniej od programów języków kompilowanych od dwóch do nawet dziesięciu i więcej razy dłużej.

W tym rozdziale autor przybliży proces kompilacji i interpretacji kodu źrodłowego języka Python; wiedza ta jest niezbędna, aby efektywnie porównywać różne aspekty dwóch wersji interpretera. Istotną jest informacja w którym momencie generowany jest kod pośredni; w pracy autor postara się odpowiedzieć na poniższe, istotne pytania:

1. Jaki procent czasu całkowitego, poprzez przetwarzanie kodu źródłowego zajmuje kompilacja?
2. W jaki sposób interpreter wpływa na wydajność wykonywanych programów?
3. Jakie są metody optymalizacji kodu bajtowego?

1.1.1. CPython

Istnieje wiele alternatywnych implementacji języka Python. W tej pracy, kiedykolwiek zostanie napisane "Python", autor miał na myśli oficjalną i stale rozwijaną implementację tego języka, to znaczy CPython, napisaną w języku C [1].

1.1.2. Etapy przetwarzania kodu wejściowego

Twórcy języka Python podzielili proces przetwarzania kodu wejściowego na poszczególne etapy [2]:

1. Analiza kodu źrodłowego i konwersja do drzewa składniowego (ang. *parse tree*)
2. Transformacja drzewa składniowego do abstrakcyjnego drzewa składniowego (ang. *AST*)
3. Konwersja drzewa AST do tzw. grafu CFG (ang. *Control Flow Graph, CFG*)
4. Produkcja kodu pośredniego bazując na grafie CFG.
5. Interpretacja kodu pośredniego przez maszynę wirtualną.

Rysunek 1.1. Proces kompilacji kodu źródłowego języka Python

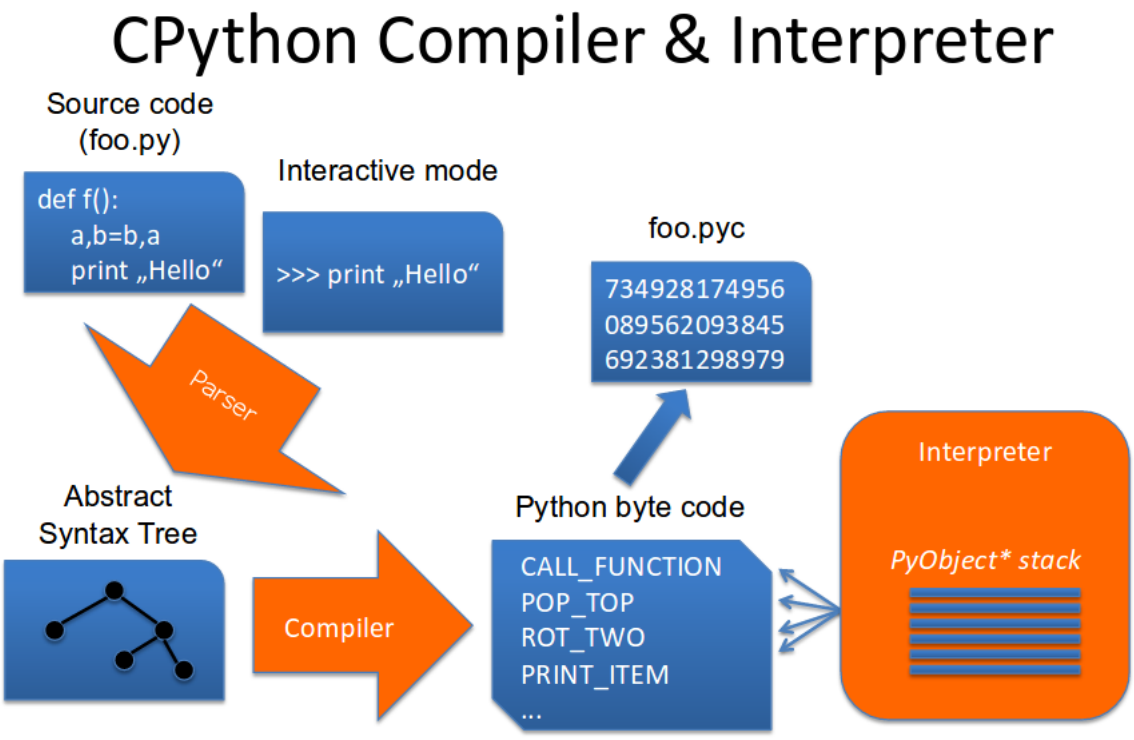
Źródło: Własne

Analiza leksykalna jest pierwszym podetapem analizy kodu źrodłowego; polega na konwersji napisanego kodu do tak zwanych znaków składniowych (ang. tokens). Analiza składniowa (ang. parsing) na podstawie otrzymanych znaków składniowych generuje strukturę ukazującą jak powiązane są ze sobą znaki, czyli abstrakcyjne drzewo składniowe (ang. *Abstract Syntax Tree*). W tym momencie kompilator przetwarza abstrakcyjne drzewo składniowe na postać obiektów kodu (ang. *code objects*) i konwertuje je do kodu pośredniego. Na samym końcu interpreter, czyli wirtualna maszyna Pythona (ang. Python Virtual Machine), który oparty jest o klasyczny stos, wykonuje poszczególne intrukcje kodu pośredniego korzystając z metod tej struktury danych.

1.2. Analiza kodu źrodłowego i konwersja do drzewa składniowego

1.2.1. Analiza leksykalna

Często zamiast próbować bezpośrednio przeanalizować strumień tekstu, można podzielić tekt wejściowy na serię znaków składniowych, w których zawarte są słowa kluczowe języka, łańcuchy znakowe, operatory i identyfikatory. Podczas kolejnego kroku, wspomniane znaki składniowe mogą być znacznie efektywniej przetwarzane przez parser. Proces transformacji tekstu wejściowego do znaków składniowych jest znany



Rysunek 1.2. Kompilacja i interpretacja kodu źrodłowego języka Python

Źródło: `https://troeger.eu/files/teaching/pythonvm08.pdf`

jako analiza leksykalna (ang. *tokenization, scanning*). Główną funkcją analizatora leksykalnego jest funkcja *PyTokenizer_Get* w pliku *Parser/tokenizer.c*. Funkcja ta jest wywoływana wielokrotnie z głównej funkcji analizy składniowej, parsetok w pliku *Parser/parsetok.c*, która z kolei jest wykorzystywana przez kilka funkcji parsujących wyższego poziomu. Oparta jest ona o skończony automat stanów, który jest ekwiwalentem wyrażeń regularnych; język Python wspiera wyrażenia regularne, w związku z czym naturalnym jest używanie ich w celu produkcji znaków składniowych. Dokładny opis całego procesu opisuje dokument [4].

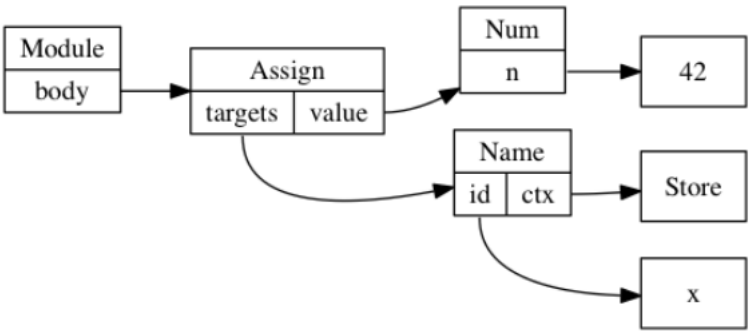
1.2.2. Analiza składniowa (parsowanie)

Aby kompilator mógł zrozumieć znaczenie programu źródłowego, strumień znaków składniowych wygenerowany przez analizator leksykalny musi zostać przekształcony w pewien zrozumiały dla kompilatora rodzaj struktury. Zajmuje się tym analizator składniowy (parser), który na podstawie dostarczonego strumienia znaków składniowych w oparciu o reguły zadeklarowane w gramatyce Pythona produkuje abstrakcyjne drzewo składniowe (ang. *AST*). Język Python używa własnego generatora analizatora składniowego aby automatycznie wygenerować parser bazując na aktualnej gramatyce języka [3].

1.2.3. Abstrakcyjne drzewa składniowe

Wyjściową strukturą parsera jest AST, które można uznać za wysokopoziomową reprezentację struktury programu bez konieczności zawierania kodu źródłowego; można je traktować jako abstrakcyjną reprezentację kodu źródłowego. Specyfikacja węzłów AST języka Python jest określona przy pomocy języka Zephyr Abstract Syntax Definition Language (ASDL) [5].

Rysunek 1.3 przedstawia przykładowe abstakcyjne drzewo składniowe. AST jest reprezentowalny poprzez listę obiektów, w której każdy obiekt może zawierać podlisty referencji do kolejnych obiektów. Każda lista lub lista podrzędna zawiera dowolną liczbę węzłów (ang. *nodes*), które reprezentują fragmenty języka. Węzły te są instancjami klas modułu *ast* z biblioteki standardowej.



Rysunek 1.3. Przykładowe abstrakcyjne drzewo składniowe

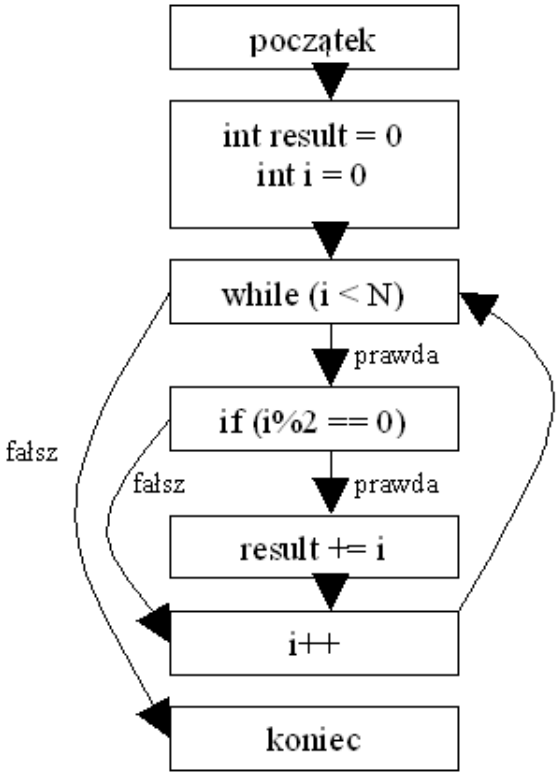
Źródło: `https://julien.danjou.info/blog/2015/python-ast-checking-method-declaration`

1.2.4. Graf przepływu sterowania CFG

Graf przepływu sterowania CFG (ang. *Control Flow Graph*) jest wykresem ukazującym wszelki możliwy przepływ informacji w programie przy użyciu podstawowych bloków zawierających kod pośredni. Graf przepływu sterowania jest statyczną reprezentacją programu, więc reprezentuje wszystkie przejścia programu. Na przykład dla instrukcji if / else zawiera jej obydwie gałęzie, choć wiadomo, że zawsze wykonuje się dokładnie jedna z nich. Cykl w grafie mówi, że w programie występuje pętla (zwłaszcza, jeśli jest to cykl pomiędzy końcem i początkiem bloku). Cykle pozwalają interpreterowi wykryć niejawnie zapisane pętle. [8]

1.3. Generowanie kodu pośredniego

Następną fazą kompilacji jest generacja kodu pośredniego - wejściowym argumentem jest abstrakcyjne drzewo składniowe skonstruowane w poprzedniej fazie, "produkowany" natomiast jest tzw. *PyCodeObject*. PyCodeObject jest niezależną jednostką wykonywanego kodu, zawierającą wszystkie dane i kod potrzebne do wykonania przez interpreter języka Python. PyCodeObjects są obiektami reprezentującymi wykony-



Rysunek 1.4. Przykładowy graf przepływu sterowania

Źródło: `https://upload.wikimedia.org/wikipedia/commons/7/78/Cfg.png`

walny kod pośredni. Kompilator wykonuje poniższe czynności w celu wygenerowania kodu pośredniego(funkcje te znajdują się w *Python/compile.c*) [6]:

1. Sprawdzenie konstrukcji z nowszych wersji Pythona - plik *future.c*
2. Budowa tabeli symboli (ang. *symbol table*) - plik *symtable.c*
3. Generacja kodu dla poszczególnych bloków - funkcja *compiler_mod()*
4. Złożenie poszczególnych bloków w końcową postać kodu pośredniego - funkcja *assemble()*
5. Optymalizacja kodu pośredniego (ang. *peephole optimizations*) - plik *peephole.c*

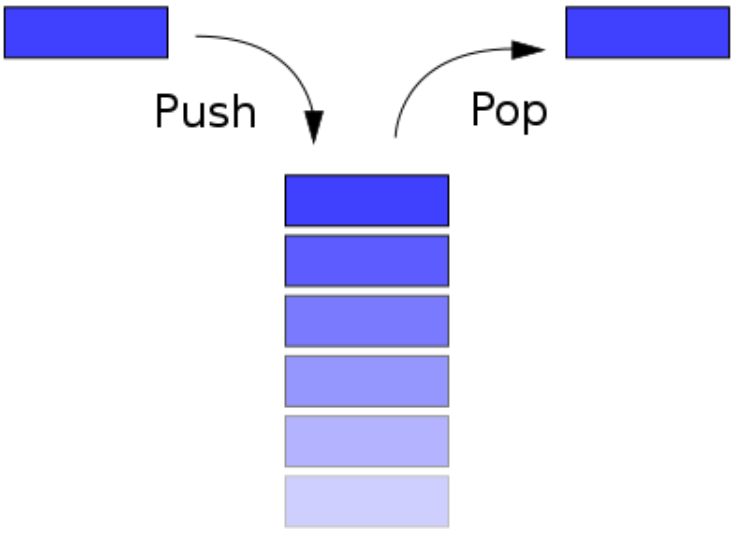
1.4. Wykonanie kodu pośredniego

Za wykonanie kodu pośredniego jest odpowiedzialny interpreter. Interpreter jest oparty o strukturę stosu; interpreter jest nazywany inaczej wirtualną maszyną Pythona.

1.4.1. Wirtualna maszyna Pythona

Wirtualna maszyna Pythona (ang. *Python Virtual Machine, PVM*) wykonuje instrukcje zawarte w kodzie pośrednim. Jest ona odseparowana od wiedzy na temat

reprezentacji struktur Pythona w kodzie źródłowym; nie jest ona także związana z kodem języka Python. Jak wcześniej wspomniano, oparta jest o klasyczną strukturę stosu. Głównymi operacjami wykonywanymi na strukturze stosu są odłożenie obiektu na stos (ang. *push*), ściągnięcie obiektu ze stosu i zwrócenie jego wartości (ang. *pop*).



Rysunek 1.5. Główne operacje wykonywane na strukturze stosu

Źródło:
`https://upload.wikimedia.org/wikipedia/commons/2/29/Data_stack.svg`

Rozdział 2

Więcej o kodzie pośrednim

Kod pośredni (kod bajtowy) uzyskiwany w procesie kompilacji opisanym w poprzednim rozdziale jest głównym tematem tej pracy i na wszelkich aspektach (nowe lub zmienione instrukcje, ich wpływ na wydajność) z nim związanych autor skupia się najbardziej. Czym zatem jest kod pośredni?

Kod źródłowy języka Python jest kompilowany do postaci kodu pośredniego (ang. *bytecode*), wewnętrznej reprezentacji Pythonowego programu w interpreterze CPython. Kod bajtowy jest przechowywany w plikach zawierających rozszerzenie *.pyc*, ten typ rozszerzenia jest także najbardziej rozpowszechnioną formą. Istnieją inne rozszerzenia plików w których zawarty jest kod pośredni, zostaną one opisane w kolejnym podrozdziale.

Dzięki produkcji plików .pyc wielokrotne wykonanie tego samego programu jest szybsze, ponieważ możliwe jest wtedy ominięcie procesu kompilacji kodu źródłowego Pythona. "Język pośredni" znajdujący się w tych plikach jest uruchamiany na maszynie wirtualnej, która wykonuje wszystkie instrukcje tego języka. Należy jednak pamiętać, że instrukcje kodu pośredniego mogą się różnić pomiedzy wydaniami języka Python; autor w pracy skupi się na wskazaniu różnic kodu bajtowego pomiędzy dwoma wersjami języka Python: 2.7.12 oraz 3.5.2.

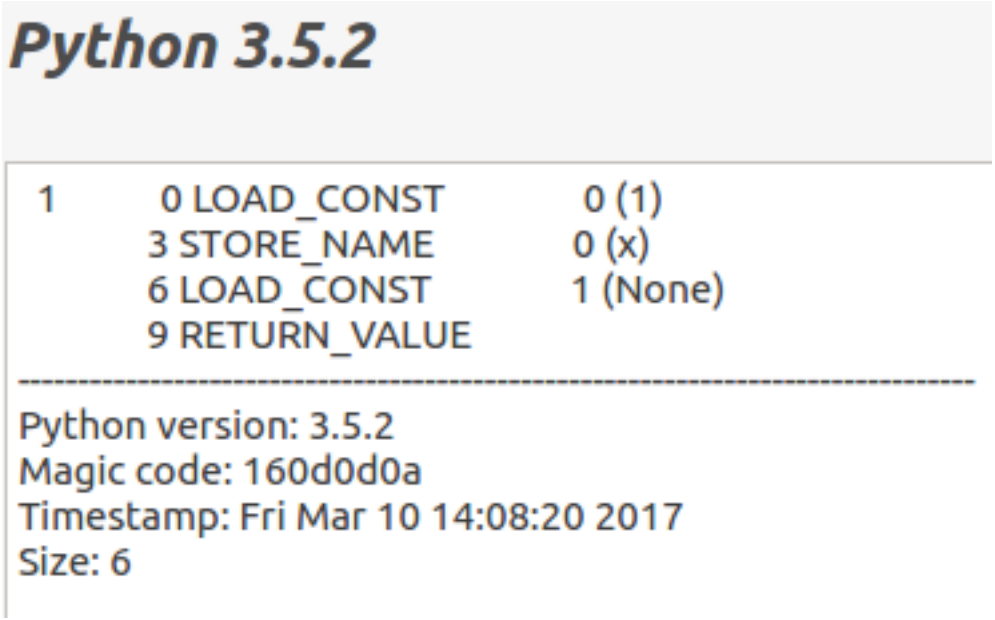
2.1. Zawartość plików .pyc

Struktura plików z rozszerzeniem .pyc jest następująca:

- pierwsze cztery bajty to tzw. *magic tag* będący identyfikatorem wersji używanego interpretera,
- kolejne czetry bajty to czas utworzenia / modyfikacji danego pliku (ang. *modification timestamp*), pomagające w podjęciu decyzji, czy powinna zajść powtórna kompilacja kodu źródłowego,
- reszta pliku to sam kod pośredni (w formie *marshalled code object*), którego format jest dyktowany przez *magic tag*.

Zaletami generowania plików .pyc są, między innymi, możliwość rezygnacji z kompilacji kodu źrodłowego w przypadku wykonywania wielokrotnie tego samego programu (oszczędność czasowa) oraz wprowadzenie pewnego rodzaju zabezpieczenia przed odczytaniem kodu źródłowego; z praktyki jednak wiadomo, że pliki .pyc są łatwe

do zdekodowania i odczytania dokładnych czynności wykonywanych przez program. Wadą jest brak kompatybilności kodów pośrednich dla różnych wersji interpretera Pythona (*magic tag* jest inny, w związku z tym nie jest on rozpoznawany przez inną wersję interpretera).



Rysunek 2.1. Przykładowy wynik disasemblacji pliku .pyc

Źródło: Własne

2.2. Moduł dis

Modułem pomagającym w disasemblacji kodu źródłowego języka Python jest *dis*, który pozwala na rozłożenie go na pojedyncze instrukcje przeznaczone dla maszyny wirtualnej Pythona. Jest możliwe, między innymi, przekazanie pojedynczej funkcji, a także i całego programu napisanego w języku Python. Moduł pozwala na głęboką analizę pisanego kodu oraz efektywną optymalizację. Dokładna dokumentacja tego modłu wraz z przykładowymi zastosowaniami znajduje się na oficjalnej stronie języka Python [7].

2.3. Składnia i analiza przykładowego programu

Zdisasemblowany kod wygląda prawie jak instukcje pisane w języku assembler przeznaczone dla mikroprocesora. Rozkład przykładowego kodu źródłowego ukazany jest na kodzie źródłowym funkcji 2.1:

Wynik działania modułu dis ukazany jest na listingu 2.2:

Disasemblacja kodu Pythonowego dostarcza następujących informacji:

1. Linię kodu źródłowego, która została przetłumaczona na dany kod bajtowy,
2. Adres każdej instrukcji kodu pośredniego,
3. Nazwę każdej wygenerowanej instrukcji,

Listing 2.1. Prosta funkcja obliczająca sumę dwóch podanych argumentów

```
1 def f(a, b):
2   res = a + b
3   return res
```

Listing 2.2. Kod pośredni funkcji f wygenerowany przez moduł dis

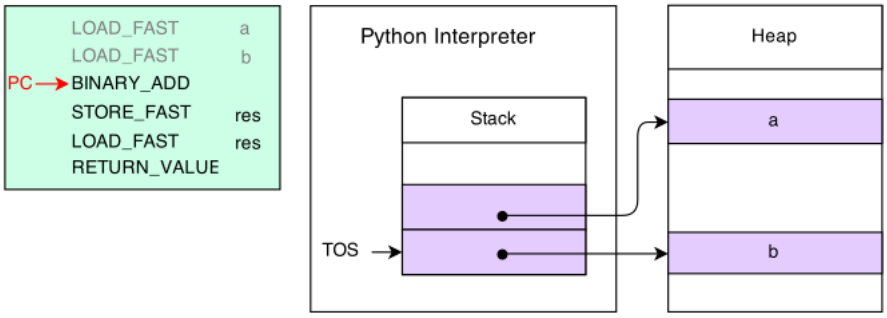
```
1      linia     adres opcode                oparg
2 ------------------------------------------------------
3         2         0 LOAD_FAST             0 (a)
4                   3 LOAD_FAST             1 (b)
5                   6 BINARY_ADD
6                   7 STORE_FAST            2 (res)
7
8         3        10 LOAD_FAST             2 (res)
9                  13 RETURN_VALUE
```

4. Indeksy argumentów podawanych do funkcji,
5. Mapowanie zawierające indeksy oraz nazwy odpowiadających zmiennych.

Ciało funkcji składa się z dwóch linii. Są one reprezentowane przez dwa bloki kodu pośredniego, jak widać na powyższym listingu. Pierwszą informacją, którą możemy odczytać jest kolumna *linia*, która wskazuje na miejsce w kodzie źródłowym z którego został wygenerowany dany blok kodu bajtowego. W większości przypadków pojedyncza linia kodu źródłowego jest tłumaczona na wiele instrukcji kodu pośredniego. W tym przypadku, z trzech linii kodu źródłowego zostało wygenerowanych sześć pojedynczych instrukcji kodu bajtowego. Proste równanie z drugiej linii zostało zamienione na aż cztery instrukcje kodu pośredniego.

Interpreter języka Python odczytuje instrukcje jedną po drugiej i generuje rezultat, wykonując wszystkie operacje zawarte w kodzie. Wszystkie referencje do obiektów sa przechowywane na wewnętrznym stosie, natomiast same obiekty i struktury danych w wewnętrznym kopcu [9].

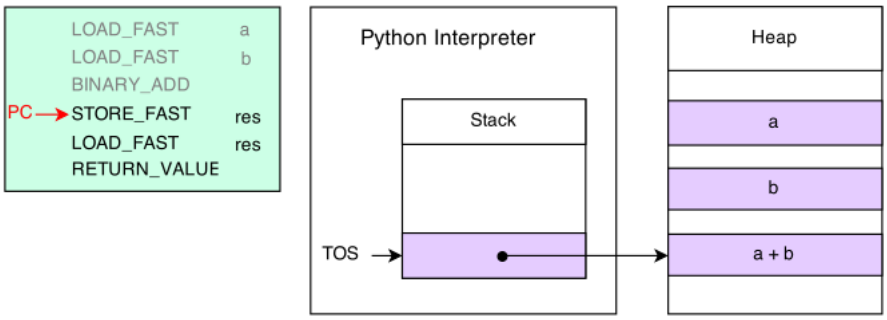
Rysunek 2.2 przedstawia jak wykonywane są instrukcje kodu pośredniego powyższego programu; TOS oznacza szczyt stosu (ang. *top of stack*), PC jest to licznik programu (ang. *program counter*); instrukcje ładujące (np. *LOAD_FAST*) wypychają referencje do obiektów na stos maszyny wirtualnej. W związku z tym, gdy pierwsze dwie instrukcje wykonają się, referencje do obiektów a i b znajdą się na stosie. Jako, że obiekt b został załadowany jako ostatni, szczyt stosu wskazuje na obiekt b. Licznik programu wskazuje wtedy na następną instrukcję kodu pośredniego do wykonania:



Rysunek 2.2. Operacje ładowania obiektów wykonywane w maszynie wirtualnej oraz na kopcu

Źródło: [10]

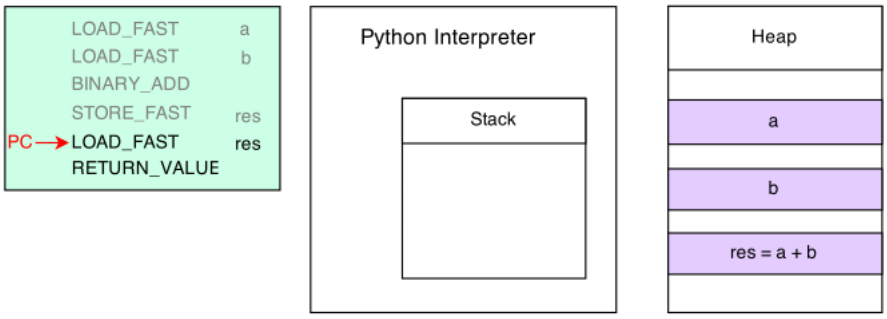
Następnym krokiem jest operacja sumowania binarnego (*BINARY_ADD*); operacje binarne usuwają dwie najwyższe referencje do obiektów ze stosu, obliczają wynik, po czym zapisują go na szczycie stosu. Operacje te ukazuje rysunek 2.3:



Rysunek 2.3. Operacje binarne wykonywane w maszynie wirtualnej oraz na kopcu

Źródło: [10]

Operacja przypisania (*STORE_FAST*) usunęła referencję znajdującą się na szczycie stosu i przypisała tę referencję do obiektu *res*, który od tego momentu zawiera w sobie wynik sumowania. Jako, że wszystkie referencje ze stosu zostały usunięte, jest on pusty:



Rysunek 2.4. Operacje wykonywane w maszynie wirtualnej oraz na kopcu

Źródło: [10]

Rysunek 2.4 nie wyjaśnia w jaki sposób są wykonywane dwie ostatnie instrukcje. Ostatnia operacja załadowania zmiennej do pamięci umieszcza referencję do obiektu przechowującego wynik działania na stosie. Ostatnią operacją jest zwrócenie referencji znajdującej się na szczycie stosu.

Wszystkie aktualne instrukcje kodu pośredniego znajdują się w dokumentacji modułu dis [7].

Rozdział 3

Różnice pomiędzy poszczególnymi wersjami języka Python

Ten rozdział opisuje różnice pomiędzy wersjami 2.7.12 oraz 3.5.2 języka Python. Analiza dotyczy zarówno wybranych elementów składniowych, jak i kodu pośredniego.

3.1. Nowe i zmienione elementy języka Python

Poniższa sekcja opisuje wybrane elementy języka Python różniące się w wersjach 2.7.12 oraz 3.5.2. [11]

3.1.1. Funkcja print()

Jedną z najbardziej znanych różnic pomiędzy dwoma wersjami języka jest zmiana wyrażenia print w Pythonie 2 na funkcję; w Pythonie 3, aby użyć tej funkcji, należy po słowie kluczowym print użyć nawiasów, tak jak w każdej innej funkcji lub metodzie. Warto nadmienić, że Python w wersji 2 obsługuje zarówno składnię z i bez nawiasów. Python w wersji 3 natomiast nie pozwala na użycie składni bez nawiasów po słowie kluczowym print.

Wspomniane różnice są ukazane na poniższych listingach:

3.1.1.1. Python 2.7.12

Listing 3.1. Program pokazujący działanie wyrażenia print dla Pythona 2.7

```
1 print 'Python', python_version()
2 print 'Witaj, swiecie!'
3 print('Witaj, swiecie!')
4 print "tekst", ; print 'wydrukuj wiecej tekstu w tej samej linii'
```

Listing 3.2. Wynik działania programu w Pythonie 2.7

```
1 Python 2.7.12
2 Witaj, swiecie!
3 Witaj, swiecie!
4 tekst wydrukuj wiecej tekstu w tej samej linii
```

3.1.1.2. Python 3.5.2

Przykładowe użycie funkcji *print* w Pythonie 3.5.2:

Listing 3.3. Funkcja print w Pythonie 3.5

```
1 print('Python', python_version())
2 print('Witaj, swiecie!')
3
4 print("Troche tesktu,", end="")
5 print(' jeszcze wiecej przykladowego tekstu')
```

Listing 3.4. Wynik działania programu w Pythonie 3.5

```
1 Python 3.5.2
2 Witaj, swiecie!
3 Troche tesktu, jeszcze wiecej przykladowego tekstu
```

Próba nieprawidłowego użycia funkcji *print* w Pythonie 3.5.2 kończy się niepowodzeniem:

Listing 3.5. Przykład nieprawidłowego użycia funkcji print w Pythonie 3.5

```
1 print 'Hello, World!'
```

Listing 3.6. Zwracany błąd dla Pythona 3

```
  File "<ipython-input-3-139a7c5835bd>", line 1
    print 'Hello, World!'
                        ^
SyntaxError: invalid syntax
```

3.1.2. Funkcje range() i xrange()

Funkcje wbudowane *range()* i *xrange()* służą do tworzenia list lub obiektów zawierających ciągi arytmetyczne.

Funkcja *xrange()* jest bardzo popularna w wersji Pythona 2.x w celu stworzenia iterowalnego obiektu, na przykład w pętli for czy tzw. "list / set-dictionary comprehension". Zachowanie funkcji jest dość podobne do generatorów, ale w tym przypadku możliwa jest nieskończona iteracja.

W Pythonie w wersji 3 funkcja range() została zaimplementowana ponownie i zachowuje się jak funkcja *xrange()*; wspomniana funkcja xrange() została wycofana w tej wersji języka i próba skorzysania z niej skończy się fiaskiem (*xrange()* powoduje *NameError* w 3 wersji języka Python).

Przykładowe użycie funkcji *range()* i *xrange()* w Pythonie 2.7.12:

Listing 3.7. Funkcje range() i xrange()

```
import timeit

n = 10000
def test_range(n):
    return for i in range(n):
        pass

def test_xrange(n):
    for i in xrange(n):
        passbachelor

print 'Python', python_version()

print '\ntiming range()'
%timeit test_range(n)

print '\n\ntiming xrange()'
%timeit test_xrange(n)
```

Dzięki tak zwanemu *lazy-evaluation*, funkcja *xrange()* jest szybsza od funkcji *range()* jeśli użytkownik chce chce iterować tylko raz (na przykład w pętli for). Z drugiej jednak strony, nie jest polecana wielokrotna iteracja, jako że generacja startuje za każdym razem od nowa.

3.1.2.1. Python 2.7.12

Listing 3.8. Funkcje range() i xrange()

```
Python 2.7.12

timing range()
1000 loops, best of 3: 433 us per loop

timing xrange()
1000 loops, best of 3: 350 us per loop
```

3.1.2.2. Python 3.5.2

Przykładowe użycie funkcji *range()* w Pythonie 3.5.2:

Listing 3.9. Wynik mierzenia czasu wykonania dla funkcji range w Pythonie 3.5

```
Python 3.5.2

timing range()
1000 loops, best of 3: 520 us per loop
```

Próba użycia nieistniejącej już funkcji *xrange()* skończy się niepowodzeniem:

Listing 3.10. Próba użycia nieistniejącej funkcji xrange() w Pythonie 3.5

```
print(xrange(10))
```

Listing 3.11. Niepowodzenie wykonania funkcji xrange() w Pythonie 3.5

```
---------------------------------------------------------------------------
NameError
Traceback (most recent call last)

<ipython-input-5-5d8f9b79ea70> in <module>()
----> 1 print(xrange(10))


NameError: name 'xrange' is not defined
```

3.1.3. Obsługa wyjątków

Błędy wykryte podczas wykonywania programu nazywane są wyjątkami; obsługa wyjątków jest kolejną z funkcjonalności, które zostały zmienione; w wersji języka 3.x koniecznie jest użycie słowa kluczowego *as*:

3.1.3.1. Python 2.7.12

Przykładowa obsługa wyjątku "NameError":

Listing 3.12. Obsługa wyjątków w Pythonie 2.7.12

```
print 'Python', python_version()
try:
    let_us_cause_a_NameError
except NameError, err:
    print err, '--> our error message'
```

Listing 3.13. Wynik działania obsługi wyjątków w Pythonie 2.7.12

```
Python 2.7.12
name 'let_us_cause_a_NameError' is not defined --> our error message
```

3.1.3.2. Python 3.5.2

W Pythonie 3.x konieczne jest użycie słowa kluczowego *as*:

Listing 3.14. Obsługa wyjątków w Pythonie 3.5

```
print('Python', python_version())
try:
    let_us_cause_a_NameError
except NameError as err:
    print(err, '--> our error message')
```

Listing 3.15. Wynik działania obsługi wyjątków w Pythonie 3.5

```
Python 3.5.2
name 'let_us_cause_a_NameError' is not defined --> our error message
```

3.1.4. Funkcja next() i metoda .next()

Jako, że funkcja *next()* i metoda *.next()* są kolejnymi szeroko stosowanymi elementami języka Python, warto jest opisać kolejną zmianę dotyczącą tej części języka. Służą one do uzyskiwania kolejnego elementu iteratora; podczas, gdy w Pythonie 2.7 możliwe jest użycie zarówno funkcji jak i metody, w Pythonie 3.x metoda *.next()* została usunięta.

3.1.4.1. Python 2.7.12

Przykładowe użycie funkcji *next()* i metody *.next()* w Pythonie 2.7:

Listing 3.16. Przykładowy program używający funkcji next() i .next()

```
1 print 'Python', python_version()
2
3 my_generator = (letter for letter in 'abcdefg')
4
5 next(my_generator)
6 my_generator.next()
```

Listing 3.17. Wynik działania funkcji next() i metody .next() w Pythonie 2.7

```
1 Python 2.7.12
2
3
4
5
6
7 'b'
```

3.1.4.2. Python 3.5.2

Przykładowe użycie funkcji *next()* w Pythonie 3.5:

Listing 3.18. Użycie funkcji next() w Pythonie 3.5

```
1 print('Python', python_version())
2
3 my_generator = (letter for letter in 'abcdefg')
4
5 next(my_generator)
```

Listing 3.19. Wynik działania funkcji next() w Pythonie 3.5

```
Python 3.5.2
```

```
'a'
```

Próba użycia metody *.next()* powoduje wystąpienie błędu *AttributeError*:

Listing 3.20. Próba użycia metody .next() w Pythonie 3.5

```
my_generator.next()
```

Listing 3.21. Niepowodzenie użycia metody .next() w Pythonie 3.5

```
---------------------------------------------------------------------------
AttributeError
Traceback (most recent call last)

<ipython-input-14-125f388bb61b> in <module>()
----> 1 my_generator.next()


AttributeError: 'generator' object has no attribute 'next'
```

3.1.5. Zwracanie iterowalnych obiektów zamiast list

W Pythonie 3.x niektóre funkcje i metody zwracają iterowalne obiekty zamiast list, tak jak to miało miejsce w Pythonie 2.x. W przeciwieństwie do generatorów, możliwa jest wielokrotna iteracja po tych obiektach; nie jest ona jednakże wydajna. Najczęściej jednak wykonywana jest jednokrotna iteracja.

Dla użytowników, którzy potrzebują instancji list, możliwa jest późniejsza konwersja iterowalnego obiektu do listy korzystając z funkcji *list()*.

3.1.5.1. Python 2.7.12

Listing 3.22. Użycie funkcji range() i ukazanie zwracanego typu w Pythonie 2.7

```
1 print 'Python', python_version()
2
3 print range(3)
4 print type(range(3))
```

Listing 3.23. Przykład zwracanego obiektu dla funkcji range() w Pythonie 2.7

```
1 Python 2.7.12
2 [0, 1, 2]
3 <type 'list'>
```

3.1.5.2. Python 3.5.2

Przykład kodu ukazującego zwracany typ dla funkcji *range()* w Pythonie 3.5.2 ukazany jest na listingach 3.24 i 3.25:

Listing 3.24. Użycie funkcji range() i ukazanie zwracanego typu w Pythonie 3.5

```
1 print('Python', python_version())
2
3 print(range(3))
4 print(type(range(3)))
5 print(list(range(3)))
```

Listing 3.25. Typ zwracanego obiektu dla funkcji range() w Pythonie 3.5

```
1 Python 3.5.2
2 range(0, 3)
3 <class 'range'>
4 [0, 1, 2]
```

Poniżej ukazane są przykłady kolejnych funkcji i metod które nie zwracają list w wersji Pythona 3.x:
1. funkcja zip()
2. funkcja map()
3. funkcja filter()
4. słownikowa metoda .keys()
5. słownikowa metoda .values()
6. słownikowa metoda .items()

3.2. Nowe, usunięte i zmienione instrukcje kodu pośredniego w wersji Pythona 3.5

Poniższa sekcja ukazuje wszelkie zmiany w strukturze kodu pośredniego wprowadzone w wersji Pythona 3.5 [7]

3.2.1. Nowe instrukcje kodu pośredniego w wersji języka 3.5

3.2.1.1. Generatory

GET_YIELD_FROM_ITER

Kod bajtowy generowany przy użyciu konstrukcji *yield from*, której użycie możliwe jest tylko i wyłącznie wewnątrz funkcji. Jeśli na górze stosu (ang. *Top of stack, TOS*) znajduje się iterator lub tak zwany obiekt *coroutine*, są one zostawiane bez zmian. W innym przypadku uruchamiane jest działanie wbudowanej funkcji *iter*, która zwraca obiekt iteratora *TOS = iter(TOS)*.

Przykładowy kod powodujący użycie wyżej wymienionej instrukcji kodu pośredniego:

Listing 3.26. Funkcja używająca generatorów w Pythonie 3.5

```
1 def func():
2     g = [(yield from range(10))]
```

Wykonanie powyższego kodu spowoduje powstanie kodu pośredniego:

Listing 3.27. Kod pośredni funkcji używającej generatorów w Pythonie 3.5

```
1   2           0 LOAD_GLOBAL              0 (range)
2               3 LOAD_CONST               1 (10)
3               6 CALL_FUNCTION            1 (1 positional,
4                                             0 keyword pair)
5               9 GET_YIELD_FROM_ITER
6              10 LOAD_CONST               0 (None)
7              13 YIELD_FROM
8              14 BUILD_LIST               1
9              17 STORE_FAST               0 (g)
10             20 LOAD_CONST               0 (None)
11             23 RETURN_VALUE
```

3.2.1.2. Operacje binarne

Operacje binarne usuwają element znajdujący się na górze stosu (*TOS*) i kolejny po nim. Po skończeniu wykonywania działania, zwracają wynik z powrotem na stos.

BINARY_MATRIX_MULTIPLY

Implementacja binarnego mnożenia macierzy. Zwraca TOS = TOS1 @ TOS.

Przykładowy program ukazujący nową instrukcję bajtową:

Listing 3.28. Kod ukazujący operator mnożenia macierzy w Pythonie 3.5

```
1 import random
2
3 a = [random.randint(1, 100) for i in range(1,11)]
4 b = [random.randint(1, 100) for i in range(1,11)]
5
6 c = a @ b
```

Z powyższego kodu źródłowego wyprodukowany został poniższy kod bajtowy:

Listing 3.29. Kod pośredni powstały w wyniku skompilowania przykładowego kodu

```
1   1           0 LOAD_CONST               0 (0)
2               3 LOAD_CONST               1 (None)
3               6 IMPORT_NAME              0 (random)
4               9 STORE_NAME               0 (random)
5
6   3          12 LOAD_CONST               2 (<code object <listcomp>
7                                              at 0x7f53df3eeb70,
8                                          file "python_3_examples/sample.py",
9                                              line 3>)
10              15 LOAD_CONST               3 ('<listcomp>')
11              18 MAKE_FUNCTION            0
12              21 LOAD_NAME                1 (range)
13              24 LOAD_CONST               4 (1)
14              27 LOAD_CONST               5 (11)
15              30 CALL_FUNCTION            2 (2 positional,
16                                             0 keyword pair)
17              33 GET_ITER
18              34 CALL_FUNCTION            1 (1 positional,
19                                             0 keyword pair)
20              37 STORE_NAME               2 (a)
21
22  4           40 LOAD_CONST               6 (<code object <listcomp>
23                                             at 0x7f53df400540,
24                                         file "python_3_examples/sample.py",
25                                             line 4>)
26              43 LOAD_CONST               3 ('<listcomp>')
27              46 MAKE_FUNCTION            0
28              49 LOAD_NAME                1 (range)
```

```
29                 52 LOAD_CONST               4 (1)
30                 55 LOAD_CONST               5 (11)
31                 58 CALL_FUNCTION            2 (2 positional,
32                                                0 keyword pair)
33                 61 GET_ITER
34                 62 CALL_FUNCTION            1 (1 positional,
35                                                0 keyword pair)
36                 65 STORE_NAME               3 (b)
37
38   6             68 LOAD_NAME                2 (a)
39                 71 LOAD_NAME                3 (b)
40                 74 BINARY_MATRIX_MULTIPLY
41                 75 STORE_NAME               4 (c)
42                 78 LOAD_CONST               1 (None)
43                 81 RETURN_VALUE
```

3.2.1.3. Operacje miejscowe

Operacje te działają podobnie to operacji binarnych z tą różnicą, że wynik operacji jest zachowywany do już istniejącej zmiennej, która przesuwana jest na szczyt stosu.

INPLACE_MATRIX_MULTIPLY

Instrukcja ta, mimo że już wprowadzona do Pythona 3.5, nie jest jeszcze wspierana:

Listing 3.30. Próba użycia nie wspieranego jeszcze operatora miejscowego mnożenia macierzy

```
1 >>> import numpy as np
2 >>>
3 >>> a = np.random.randn(4, 10)
4 >>> b = np.random.randn(10, 5)
5 >>> a @= b
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8 TypeError: In-place matrix multiplication is not (yet) supported.
9 Use 'a = a @ b' instead of 'a @= b'.
```

3.2.1.4. Odpakowywanie zmiennych

Poniższe instrukcje kodu pośredniego zostały zaproponowane przez Joshuę Landau w PEP 448 i wprowadzone w wersji interpretera 3.5 [12]. Od tej wersji Pythona, możliwe jest odpakowywanie (ang. *unpacking*) więcej niż jednej zmiennej:

BUILD_TUPLE_UNPACK(count)

Instrukcja zdejmuje ze stosu iterowalne obiekty, łączy je w pojedynczą krotkę i zwraca rezultat na szczyt stosu.

Przykładowy program ukazujący nowe możliwości interpretera w tym zakresie:

Listing 3.31. Program ukazujący nowe możliwości odpakowywania krotek

```
1 a = [1,2,3]
2 b = [3,4,5]
3 c = 7
4
5 def tuple_unpack(a,b,c):
6     return (*a, *b, c)
```

Listing 3.32. Wynik działania programu odpakowującego krotkę

```
1 >>> tuple_unpack(a,b,c)
2 (1, 2, 3, 3, 4, 5, 7)
```

Listing 3.33. Kod bajtowy funkcji odpakowaującej krotkę

```
1   2           0 LOAD_FAST                0 (a)
2               3 LOAD_FAST                1 (b)
3               6 LOAD_FAST                2 (c)
4               9 BUILD_TUPLE              1
5              12 BUILD_TUPLE_UNPACK       3
6              15 RETURN_VALUE
```

BUILD_LIST_UNPACK(count)

Stosując zapis *[*x, *y, *z]*, możliwe jest podobne działanie do odpakowywania tupli, z tym, że na szczyt stosu zwracana jest lista:

Listing 3.34. Przykładowy program ukazujący funkcję do odpakowywania listy

```
1 my_tuple = (11, 12, 45)
2 my_list = ['something', 'or', 'other']
3 my_range = range(5)
4
5 def list_unpack(my_tuple, my_list, my_range):
6     return [*my_tuple, *my_list, *my_range]
```

Listing 3.35. Wywołanie funkcji zwracającej listę

```
1 >>> list_unpack(my_tuple, my_list, my_range)
2 [11, 12, 45, 'something', 'or', 'other', 0, 1, 2, 3, 4]
```

Listing 3.36. Kod bajtowy funkcji zwracającej odpakowaną listę

```
1   2           0 LOAD_FAST                0 (my_tuple)
2               3 LOAD_FAST                1 (my_list)
3               6 LOAD_FAST                2 (my_range)
4               9 BUILD_LIST_UNPACK        3
5              12 RETURN_VALUE
```

BUILD_SET_UNPACK(count)

Operacja podobna do *BUILD_TUPLE_UNPACK* i *BUILD_LIST_UNPACK*, z tym, że na szczyt stosu zwracany jest zbiór danych (ang. *set*). Dzięki tej instrukcji możliwe jest odpakowywanie zbiorów danych w następujący sposób: {*x, *y, *z}.

Listing 3.37. Przykładowy program ukazujący działanie odpakowywania zbioru danych

```
1 my_list = [1,2,3]
2 my_tuple = (3,4,5)
3 my_set = {5,6,7}
4
5 def unpack_set(my_list, my_tuple, my_set):
6     return {*my_list, *my_tuple, *my_set}
```

Listing 3.38. Wywołanie funkcji rozpakowującej zbiór danych

```
>>> unpack_set(my_list, my_tuple, my_set)
{1, 2, 3, 4, 5, 6, 7}
```

Kod pośredni dla funkcji *unpack_set* wygląda następująco:

```
>>> dis.dis(unpack_set)
  2           0 LOAD_FAST                0 (my_list)
              3 LOAD_FAST                1 (my_tuple)
              6 LOAD_FAST                2 (my_set)
              9 BUILD_SET_UNPACK         3
             12 RETURN_VALUE
```

BUILD_MAP_UNPACK(count)

Instrukcja usuwa ze stosu tyle referencji, ile otrzymała w wywołaniu, po czym łączy obiekty w jeden słownik. Dzięki tej instrukcji możliwe jest łączenie i odpakowywanie słowników w następujący sposób: {***x, **y, **z*}.

Listing 3.39. Program ukazujący łączenie słowników w Pythonie 3.5

```
first_dict = {1:'e', 2:'f'}
second_dict = {2: '3', 3:'es', 4:'x'}

def unpack_map(first_dict, second_dict):
    return {**first_dict, **second_dict}
```

Wywołanie funkcji da następujący wynik (wartość klucza *2* z pierwszego słownika została nadpisana):

Listing 3.40. Wywołanie funkcji odpakowującej słownik

```
>>> unpack_map(first_dict, second_dict)
{1: 'e', 2: '3', 3: 'es', 4: 'x'}
```

Użycie funkcji *dis* z pakietu o tej samej nazwie wygeneruje kod pośredni w postaci:

Listing 3.41. Wygenerowany kod bajtowy dla funkcji unpack$_m ap()$

```
>>> dis.dis(unpack_map)
  2           0 LOAD_FAST                0 (first_dict)
              3 LOAD_FAST                1 (second_dict)
              6 BUILD_MAP_UNPACK         2
              9 RETURN_VALUE
```

BUILD_MAP_UNPACK_WITH_CALL(oparg)

Instrukcja podobna do *BUILD_MAP_UNPACK*, ale używana w postaci wywołania w funkcji: *f(**x, **y, **z)*. Mniej znaczący (ang. *lowest*) bajt w przekazywanym argumencie jest liczbą mapowań, natomiast relatywna pozycja wywoływanej funkcji jest zakodowana w drugim bajcie parametru *oparg*.

Listing 3.42. Program pokazujący odpakowanie mapy w funkcji

```
1 first_dict = {'1': 'one', '2': 'two'}
2 second_dict = {'3': 'three', '4': 'four'}
3
4 def dict_unpack(first_dict, second_dict):
5     return dict(**first_dict, **second_dict)
```

Dwa słowniki są łączone w jeden, odpakowanie skończyło się sukcesem:

Listing 3.43. Wynik działania programu odpakowującego mapę w funkcji

```
1 >>> dict_unpack(first_dict, second_dict)
2 {'2': 'two', '4': 'four', '3': 'three', '1': 'one'}
```

Jak widać poniżej, instrukcja *BUILD_MAP_UNPACK_WITH_CALL* występuje po użyciu funkcji dis:

Listing 3.44. Kod bajtowy funkcji odpakowującej mapę

```
1 >>> dis.dis(dict_unpack)
2   2           0 LOAD_GLOBAL              0 (dict)
3               3 LOAD_FAST                0 (first_dict)
4               6 LOAD_FAST                1 (second_dict)
5               9 BUILD_MAP_UNPACK_WITH_CALL   258
6              12 CALL_FUNCTION_KW         0 (0 positional,
7                                             0 keyword pair)
8              15 RETURN_VALUE
```

3.2.2. Zmienione instrukcje kodu pośredniego w wersji języka 3.5

3.2.2.1. Build map

BUILD_MAP(count)

Instrukcja służąca do wypychania na stos nowego obiektu mapy. Ze stosu usuwanych jest podwójna ilość przyjmowanych argumentów *count*, w ten sposób, że słownik zawiera w sobie wpisy: *..., TOS3: TOS2, TOS1: TOS*.

W wersji interpretera 3.5 zmiany obejmują sposób tworzenia samego słownika: mapa jest tworzona bezpośrednio z elementów znajdujących się na stosie, zamiast używać instrukcji *STORE_MAP*, czyli tworzenia pustego słownika służącego do przechowania par klucz: wartość.

Przykładowy kod ukazujący funkcję tworzącą słownik:

Listing 3.45. Funkcja tworząca słownik

```
1 def sample():
2     return {1:2, 3:4}
```

Listing 3.46. Kod bajtowy funkcji tworzącej słownik w Pythonie 3.5

```
1 >>> dis.dis(sample)
2   2           0 LOAD_CONST               1 (1)
3               3 LOAD_CONST               2 (2)
4               6 LOAD_CONST               3 (3)
5               9 LOAD_CONST               4 (4)
6              12 BUILD_MAP                2
7              15 RETURN_VALUE
```

Dla kontrastu, jak wygląda kod pośredni tej samej funkcji w Pythonie 2.7.12:

Listing 3.47. Kod bajtowy funkcji tworzącej słownik w Pythonie 2.7

```
1 >>> dis.dis(sample)
2   2           0 BUILD_MAP                2
3               3 LOAD_CONST               1 (2)
4               6 LOAD_CONST               2 (1)
5               9 STORE_MAP
6              10 LOAD_CONST               3 (4)
7              13 LOAD_CONST               4 (3)
8              16 STORE_MAP
9              17 RETURN_VALUE
```

3.2.3. Usunięte instrukcje kodu pośredniego w wersji języka 3.5

3.2.3.1. Stop code

STOP_CODE()

Instrukcja oznaczająca koniec kodu dla kompilatora, nie używana przez interpreter języka Python.

Rozdział 4

Wydajność języka Python

Ponieważ Python jest językiem interpretowanym, a jego najczęściej używana implementacja, CPython jest w wielu przypadkach wolniejsza (nawet do stu razy) od języków niskopoziomowych, takich jak C czy C++, pojawia się wiele pytań dotyczących wydajności tego języka. [13] W tym rozdziale zostanie opisanych kilka czynników wpływających na produktywność interpretera CPython. Autor postara się także wyjaśnić w jakim stopniu tworzenie i wykonywanie kodu pośredniego wpływa na wydajność języka Python.

4.1. Co wpływa na pogorszenie wydajności języka Python?

Istnieje wiele elementów będących przyczynami pogorszenia wydajności języka Python. CPython to oparty na stosie interpreter kodu bajtowego napisany w języku C; wszystkie dane, w tym liczby, są manipulowane przez wskaźniki do struktur sterty, które mają wspólny interfejs PyObject. Nie ma dodatkowych informacji o typie statycznym zmiennych. Wszystkie operacje, w tym podstawowa arytmetyka, są generowane dynamicznie w oparciu o typy argumentów. Automatyczne zarządzanie pamięcią opiera się głównie na liczeniu odwołań (ang. *reference counting*), ze znacznikowym uwalnianiem zasobów (ang. *mark and sweep garbage collecting*) w celu zbierania obiektów cyklicznych.

Benchmark	Ref. counting	Dynamic typing	Number boxing	Container boxing	Call stack	Late binding
crypto_pyaes	7 %	5 %	34 %	24 %	1 %	3 %
fannkuch	5 %	3 %	1 %	5 %	0 %	0 %
float	0 %	1 %	32 %	0 %	2 %	30 %
go	6 %	2 %	2 %	0 %	10 %	16 %
hexiom2	3 %	0 %	2 %	3 %	8 %	18 %
meteor-contest	5 %	0 %	1 %	0 %	2 %	4 %
nbody-modified	2 %	15 %	19 %	12 %	0 %	5 %
pidigits	1 %	0 %	0 %	0 %	0 %	0 %
richards	7 %	0 %	0 %	0 %	10 %	23 %
scimark_fft	7 %	9 %	38 %	24 %	0 %	0 %
scimark_lu	2 %	1 %	6 %	6 %	0 %	16 %
spectral_norm	5 %	15 %	43 %	0 %	6 %	3 %

Rysunek 4.1. Pogorszenie wydajności języka Python przez wbudowane cechy

Źródło: [17]

4.1.1. Dynamicznie typowany

Python jest językiem dynamicznie typowanym, co oznacza, że każda zmienna jest przypisana tylko i wyłącznie do obiektu - w przypadku języków typowanych statycznie każda zmienna jest przypisana zarówno do obiektu, jak i do typu danej zmiennej. Język Python, przez to że jest dynamicznie typowany, pozwala na przykład na przypisanie jednej zmiennej najpierw do obiektu typu zmiennoprzecinkowego, a następnie do obiektu ciągu znakowego:

Listing 4.1. Program pokazujący dynamiczną naturę języka Python

```
>>> variable = 1
>>> type(variable)
<type 'int'>
>>> variable = 'a string'
>>> type(variable)
<type 'str'>
>>> variable = [1,2,3]
>>> type(variable)
<type 'list'>
>>> variable = {1: 'one', 2: 'two'}
>>> type(variable)
<type 'dict'>
```

Dla kontrastu, podczas kompilacji statycznego kodu źródłowego napisanego w C, kompilator ma wiele możliwości, aby usprawnić i zoptymalizować proces wykonywania instrukcji przez procesor, ponieważ typy zmiennych są znane już w trakcie kompilacji:

Listing 4.2. Program pokazujący statyczną naturę języka C

```
#include <stdio.h>

int main(int argc, char **argv)
{
        int var = 1;

        printf("%d\n", var);

        char var = 'a';

        printf("%d\n", var);
}
```

Powyższy program nie skompiluje się z powodu statycznej natury języka C, co ukazuje listing 4.3.

W Pythonie optymalizacja jest znacznie utrudniona, z racji, że funkcjonalność poszczególnych zmiennych może się zmienić w trakcie trwania programu.

Listing 4.3. Niepowodzenie kompilacji w języku C

```
/bin/sh -c '/usr/bin/make -j4 -e -f  Makefile'
----------Building project:[ test - Debug ]----------
make[1]: Entering directory '/home/maciek/Documents/C/test'
/usr/bin/gcc -c  "/home/maciek/Documents/C/test/main.c" -g -O0 -Wall
-o ./Debug/main.c.o -I. -I.
/home/maciek/Documents/C/test/main.c: In function 'main':
/home/maciek/Documents/C/test/main.c:13:7: error: conflicting
types for 'var'
  char var = 'a';
       ^
/home/maciek/Documents/C/test/main.c:9:6: note: previous definition
of 'var' was here
  int var = 1;
      ^
test.mk:95: recipe for target 'Debug/main.c.o' failed
make[1]: *** [Debug/main.c.o] Error 1
make[1]: Leaving directory '/home/maciek/Documents/C/test'
Makefile:4: recipe for target 'All' failed
make: *** [All] Error 2
====2 errors, 1 warnings====
```

4.1.2. Zliczanie referencji

Zliczanie referencji (ang. reference counting) jest jedną z najłatwiejszych metod odśmiecania na potrzeby zarządzania pamięcią.

Ponieważ Python intensywnie korzysta z funkcji *malloc()* i *free()*, potrzebuje strategii, aby uniknąć wycieków pamięci, a także sposobu wykorzystania zwolnionej pamięci. Wybrana metoda nazywana jest liczeniem odwołań. W metodzie tej wraz z każdym obiektem skojarzony jest licznik, który służy do zliczania wszystkich aktualnych odwołań do obiektu. Za każdym razem, kiedy tworzona jest nowa referencja do obiektu, licznik jest zwiększany o jeden, natomiast kiedy odwołania te są usuwane, licznik jest zmniejszany o jeden. Kiedy wartość licznika będzie równa zero, oznaczać to będzie, że dany obiekt nie jest już osiągalny, więc można zwolnić przydzieloną pamięć. Jeśli przed skasowaniem obiekt sam używał referencji do innych obiektów, to liczniki odwołań tych obiektów także są zmniejszane; mogą zatem również osiągnąć wartość zero, co spowoduje rekursywne skasowanie kolejnych obiektów [16]

Interpreter CPython obecnie używa schematu zliczania referencji z (opcjonalnie) opóźnionym wykrywaniem cyklicznych referencji, które zbiera większość obiektów, gdy tylko staną się niedostępne, ale nie gwarantuje usuwania obiektów zawierających odwołania cykliczne.

Istnieją dwa makra: *Py_INCREF(x)* i *Py_DECREF(x)*, które obsługują inkrementację i dekrementację licznika odwołań. *Py_DECREF()* również zwalnia obiekt, gdy liczba osiągnie zero. Aby uzyskać elastyczność, nie wywołuje on bezpośrednio funkcji *free()* bezpośrednio - raczej wywołuje funkcję za pośrednictwem wskaźnika w obiekcie. W tym celu (i innych) każdy obiekt zawiera również wskaźnik do jego typu.

Listing 4.4. Struktura PyObject opisująca każdy obiekt w języku Python

```
typedef struct _object {
    _PyObject_HEAD_EXTRA
    Py_ssize_t ob_refcnt;           // Licznik referencji
    struct _typeobject *ob_type;
} PyObject;
```

Jedną z wad zliczania referencji jest fakt, że nie nie radzi sobie z zależnościami cyklicznymi pomiędzy obiektami. Jeśli obiekt A wskazuje na obiekt B, natomiast obiekt B wskazuje z powrotem na obiekt A, to żaden z nich nie zostanie nigdy zwolniony. Wykorzystanie tej metody może także prowadzić do nadmiernej fragmentacji stosu pamięci.

Kolejną wadą jest to, że ta metoda czyszczenia pamięci programu jest wolne; za każdym razem gdy obiekt zostanie przydzielony, odniesiony lub dereferencjowany, zachodzi dodatkowa kontrola; Oznacza to, że zamiast normalnego działania, a następnie wykonywania operacji czyszczenia wszystkich obiektów naraz, jak robią to uwalniacze zasobów w innych językach, licznik odwołań CPythona rodzieli pracę procesora pomiedzy różne czynności i będzie wykonywać operację czyszczenia rzadziej. Niestety oznacza to, że łączna ilość pracy, która jest przeznaczona na uwalnianie zasobów, jest wyższa.

4.1.3. Opakowanie liczb i struktur danych

Opakowywanie i rozpakowywanie to proces przekształcania wartości pierwotnej w obiektową klasę opakowującą (ang. *boxing*) lub konwertowanie wartości z obiektowej klasy opakowującej z powrotem na prymitywną wartość (ang. *unboxing*).

Według badań Gergö Barany'ego opakowanie liczb jako obiektów na kopcu jest najbardziej kosztowną cechą języka Python w kwestii wykonywania obliczeń numerycznych, skutkując w wykonaniu programu dłuższym nawet o 43 %. [17] Boksowanie i rozpakowywanie nie jest z natury złe, ale jest kompromisem. W zależności od implementacji języka rozwiązanie to może być wolniejsze i wymaga więcej pamięci niż tylko używanie prymitywów. Może to jednak również pozwolić na użycie struktur danych na wyższym poziomie i większą elastyczność kodu.

Listing 4.5. Pseudokod przedstawiający proces pakowania i odpakowywania

```
int i=123;
object o=(object)i; //Opakowanie

o=123;
i=(int)o; //Rozpakowanie
```

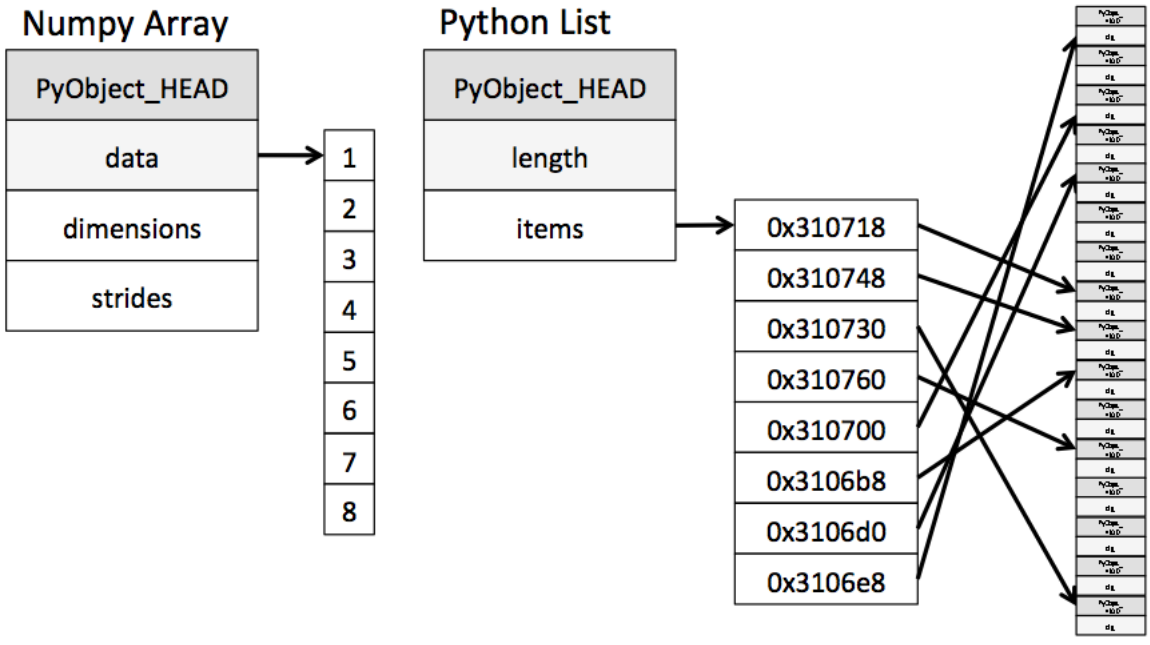
Opakowane wartości to struktury danych, które są minimalnymi "owijkami" (ang. *wrappers*) wokół typów pierwotnych. Wartości opakowane są zwykle przechowywane jako wskaźniki do obiektów na stercie.

W związku z tym wartości zapakowane wykorzystują więcej pamięci i wymagają co najmniej dwóch wyszukiwań pamięci, aby uzyskać do nich dostęp: pierwsze wyszukanie, aby uzyskać wskaźnik, a drugie, aby za pomocą tego wskaźnika dotrzeć do elementu pierwotnego. Z drugiej strony, wartości zapakowane zazwyczaj lepiej działają z innymi typami w systemie. Ponieważ są one pierwszorzędnymi strukturami danych w języku, mają oczekiwane metadane i strukturę, które mają inne struktury danych.

Przez boksowanie kontenerów rozumiemy przede wszystkim dostęp do list i tablic Pythona. Listy w języku Python można indeksować jako kolejne heterogenne sekwencje (wektory) obiektów zapakowanych. Tablice znane z języka C za to są homogenicznymi wektorami odpakowanych liczb. Gdy potrzebny jest dostęp do listy lub tablicy według indeksu numerycznego, indeks musi zostać rozpakowany, aby wykonać obliczenia adresu w celu uzyskania dostępu do podstawowej pamięci[18]; w przypadku tablic dodatkowo elementy znajdujące się pod danym numerem muszą być sprawdzane pod względem ich typu. Zarówno w przypadku dostępu do list, jak i do tablic kontrolowane jest, czy dany indeks nie wychodzi poza ramy dostępnych adresów w pamięci.

Dla przykładu, tablica w bibliotece NumPy stworzonej do obliczeń numerycznych w najprostszej formie to obiekt Pythona zbudowany wokół tablicy C. Oznacza to, że ma wskaźnik do ciągłego bufora danych wartości. Z drugiej strony, lista Pythona ma wskaźnik do ciągłego bufora wskaźników, z których każdy wskazuje obiekt Pythona, który z kolei ma odniesienia do jego danych (w tym przypadku liczb całkowitych).

Oto schemat tego, jak dwa typy struktur mogą wyglądać:



Rysunek 4.2. Porównanie działania tablic pakietu NumPy i list języka Python

Źródło: [20]

4.1.4. Stos wywoławczy

W momencie, gdy funkcja zwraca wynik, interpreter wie, do którego miejsca programu ma wrócić. Dzieje się tak za sprawą wbudowanego stosu wywoławczego (ang. *call stack*). Stos wywołań może składować poniższe dane:

1. ramki wywołań (ang.*call frames*) - jedna dla każdego wywołania funkcji,
2. dane zmiennych lokalnych.

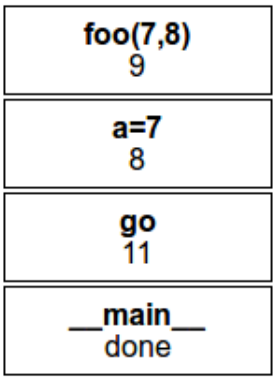
Ramka wywołań to obszar pamięci który jest przetrzymywany na stosie, tak, aby interpreter wiedział jaka funkcja jest aktualnie wykonywana. Ramki wywoławcze są tworzone w momencie gdy funkcja jest wywoływana, a usuwane ze stosu wywoławczego gdy funkcja kończy swe działanie. Na samym początku działania programu, stos wywołań zawiera jedynie ramkę nazwaną *__main__*. W miarę działania programu, więcej ramek jest dokładanych na stos; każda ramka zawiera nazwę wywołanej funkcji oraz linię, do której interpreter ma wrócić po zakończeniu działania tejże funkcji.

Zmienne lokalne także są wypychane na stos w momencie wywołania funkcji, a kiedy funkcja kończy swe działanie są usuwane ze stosu wywołań.

Listing 4.6. Przykładowy kod ukazujący ideę działania stosu wywoławczego

```
1  def foo(x,y):    # <-- w tym miejscu jest interpreter
2      print("x=",x,"y=",y)
3      z=x+y
4      print("z=",z)
5
6  def go():
7      a=7
8      foo(a,a+1)
9
10 go()
```

Stos wywołań wygląda następująco:



Rysunek 4.3. Stos wywołań interpretera chwilę po wywołaniu funkcji foo(x, y)

Źródło: Własne

Interpreter języka Python zarządza stosem wywoławczym na stercie do obsługi współprogramów (ang. *coroutines*) używanych w niektórych technikach programowa-

nia i dla łatwiejszego opisywania wyjątków. Jeśli zamiast zwracać, funkcja wykonuje instrukcję *yield*, jej wykonanie jest zawieszone i można je wznowić później. W tym przypadku jego stan wykonania jest hermetyzowany w ramce stosu, która nie może być usunięta ze stosu wywołań. Z tego powodu należy tworzyć ramki do stosu dynamicznie. W związku z tym, koszt czasowy wsparcia współprogramów, które są rzadkie, jest wysoki i każde wywołanie funkcji powoduje pewne opóźnienia.

4.1.5. Dynamiczne wiązanie

Późne wiązanie lub dynamiczne wiązanie (ang. *late binding, dynamic binding*) to mechanizm interpretera Python, w którym metoda wywoływana na obiekcie lub funkcja wywoływana z argumentami jest wyszukiwana po nazwie w trakcie jej wykonywania. Dla kodu obiektowego, dynamiczne wiązanie to około 30 % całkowitego czasu programu. [17]

Dynamiczny charakter języka Python obejmuje także wywołania funkcji, które zazwyczaj wykorzystują późne wiązanie. Python używa dynamicznego zasięgu zmiennych, więc każde użycie globalnego obiektu spowoduje obciążenie wyszukiwania w globalnej tabeli skrótów(ang. *hash table*) dla bieżącego kontekstu (plus inne wyszukiwania we wbudowanych tabelach, jeśli globalneg wyszukiwanie zawiedzie). [21]

Późne wiązanie ma gorszą wydajność niż wczesne wiązanie (ang. *early binding*) dla wywołania danej metody. W większości implementacji, prawidłowy adres metody musi być sprawdzany po nazwie przy każdym wywołaniu, wymagając stosunkowo drogiego czasowo wyszukiwania słownikowego.

Z drugiej jednak strony, dynamiczne wiązanie umożliwia brak stosowania statycznego sprawdzania typu. Podczas wywoływania opóźnionego, interpreter musi przyjąć, że metoda istnieje. Oznacza to, że prosty błąd składniowy może spowodować zgłoszenie błędu w środowisku wykonawczym. W języku Python spowoduje to wystąpienie wyjątku "NameError":

Listing 4.7. Przykładowy kod ukazujący dynamiczne wiązanie w języku Python

```
1 >>> non_existing_function()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 NameError: name 'non_existing_function' is not defined
```

Rozdział 5

Jak przyspieszyć język Python?

Wydajność Pythona jest dość słaba w porównaniu do nowoczesnych języków takich jak Lua i JavaScript. Dlaczego Python pozostaje tak daleko w tyle w stosunku do innych języków? Interfejs jezyka Python (ang. *application programming interface, API*) i mnogość bibliotek, które sprawiają, że Python jest potężnym językiem również sprawiają, że bardzo trudno jest go efektywnie wykonać. Projekty takie jak Cython, Numpy czy PyPy, który bazuje na kompilatorze czasu rzeczywistego (ang. *just-in-time compiler, JIT*) pokazują, że możliwe jest uruchamianie niektórych programów Pythona nawet do pięćdziesięciu razy szybciej, pozostając wiernym swojej semantyce. [17]

5.1. JIT - just in time compiler

Kompilacja w czasie rzeczywistym (JIT), znana również jako tłumaczenie dynamiczne, jest sposobem wykonywania kodu komputerowego, który polega na kompilacji podczas wykonywania programu zamiast przed wykonaniem. Najczęściej jest to tłumaczenie kodu źródłowego lub częściej kodu bajtowego na kod maszynowy, który jest następnie wykonywany bezpośrednio. System implementujący kompilator JIT zazwyczaj w sposób ciągły analizuje wykonywany kod i identyfikuje części kodu, w których przyspieszenie uzyskane z kompilacji lub rekompilacji przeważy nad kosztami kompilacji tego kodu. [26]

Kompilacja JIT to połączenie dwóch tradycyjnych podejść do tłumaczenia na kod maszynowy - kompilacja z wyprzedzeniem (ang. *ahead of time compilation, AOT*) i interpretacja - i łączy w sobie zalety i wady obydwu metod. Kompilacja JIT łączy szybkość skompilowanego kodu z elastycznością interpretacji, z narzutem wirtualnej maszyny Pythona i kompilacji. [18] Kompilacja JIT jest formą dynamicznej kompilacji i pozwala na optymalizację adaptacyjną, taką jak rekompilacja dynamiczna - tak więc w teorii kompilacja JIT może przynieść szybsze wykonanie niż kompilacja statyczna.

5.1.1. PyPy

PyPy to alternatywna implementacja języka programowania Python napisanego w samym Pythonie. W szczególności jego interpreter jest napisany w języku RPython (który jest podzbiorem języka Python). Wdrożenie interpretera w Pythonie na wysokim poziomie, nad implementacją niskiego poziomu w C, umożliwia szybkie eksperymentowanie z nowymi funkcjami językowymi. [23] Okazuje się, że w pewnych przypadkach jest to korzystne i widoczne w postaci prędkości wykonywania czy zużycia pamięci.

RPython(ang. *Restricted Python*) to język programowania będący statycznie typowanym podzbiorem języka Python. Nazwą tą określa się także projekt pozwalający na manipulację programów RPython. Każdy program napisany w RPythonie jest zarazem poprawnym programem w zwykłym Pythonie.

Interpreter Pypy składa się z następujących komponentów:

1. Kompilatora kodu bajtowego odpowiedzialnego za produkcję kodów obiektów (ang. *code objects*) z kodu źródłowego,
2. Analizatora kodu pośredniego odpowiedzialnego za interpretację kodów obiektów języka Python,
3. Standardowej przestrzeni obiektowej, odpowiedzialnej za tworzenie i manipulowanie obiektami Python widocznymi w aplikacji.

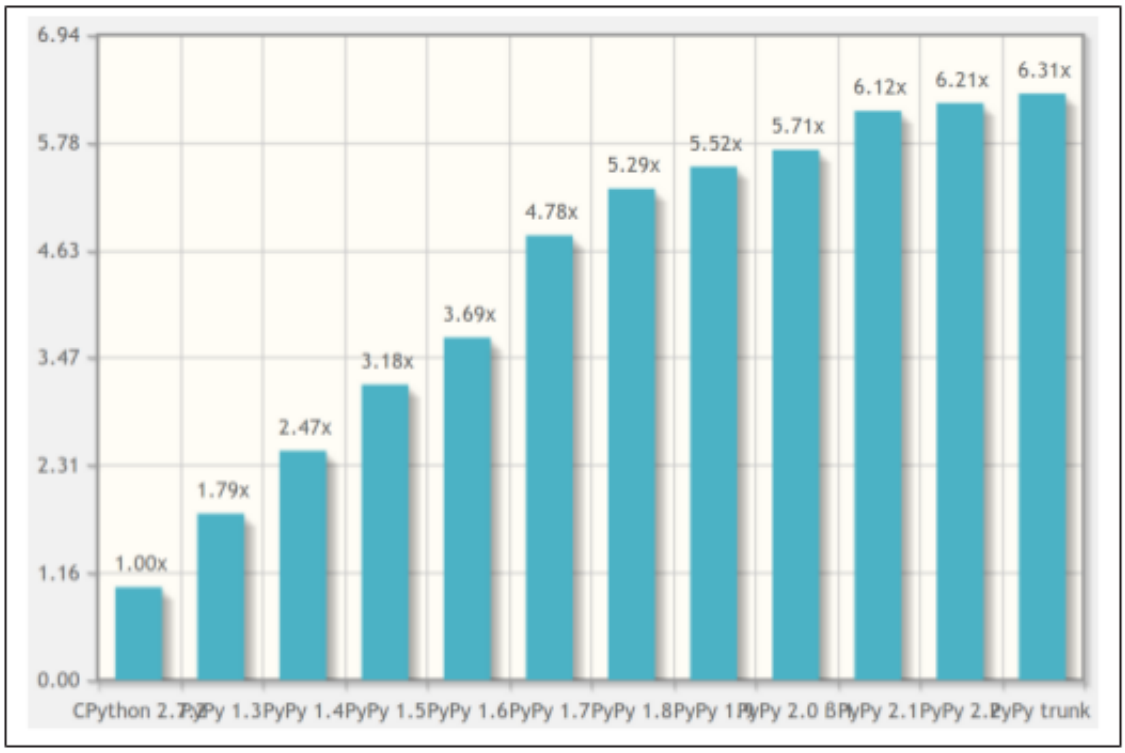
Kompilacja kodu bajtowego jest fazą wstępną, która tworzy zwarty format kodu pośredniego za sprawą łańcucha operacji (tokenizera, analizy leksykalnej, analizatora składni, konstruktora drzewa AST, generatora kodu pośredniego). Ewaluator kodu bajtowego interpretuje owy kod bajtowy; wykonuje większość swojej pracy poprzez delegowanie wszystkich rzeczywistych manipulacji obiektami użytkownika do przestrzeni obiektów. Przestrzeń obiektów można uważać za bibliotekę wbudowanych typów. Definiuje ona implementację obiektów użytkownika, takich jak liczby całkowite i listy, a także operacje między nimi.

Ten podział między ewaluatorem kodu bajtowego i przestrzenią obiektową daje dużą elastyczność. Można podłączyć różne przestrzenie obiektów, aby uzyskać różne lub wzbogacone zachowania obiektów Pythona.

Operacja przetwarzania zaczyna się od kodu źródłowego napisanego w podzbiorze RPython, następnie stosowany jest łańcuch narzędzi tłumaczenia języka RPython, kończąc z PyPy jako binarnym plikiem wykonywalnym.

PyPy nie ingeruje w semantykę języka Python w żaden sposób; ponieważ kopmilator czasu rzeczywistego PyPy pracuje na zasadzie meta-analizatora (ang. *meta-tracer*), nie śledzi on sposobu wykonywania programu użytownika, lecz interpretera wykonującego program. [23] Oznacza to, że kod który produkuje zawiera tylko i wyłącznie operacje na poziomie języka RPython.

Szybkość Pypy z czasem ewoluowała; wykres przedstawiony na rysunku 5.1 ze strony *speed.pypy.org* przedstawia ulepszenia szybkości PyPy wraz z kolejnymi wersjami.



Rysunek 5.1. Wykres ukazujący zmiany w wydajności PyPy w kolejnych wersjach

Źródło: speed.pypy.org

5.1.2. Numba

Numba jest darmowym kompilatorem omptymalizującym dla języka CPython. Używa on infrastruktury kopmilatora LLVM (ang. *Low Level Virtual Machine*) do kompilowania kodu Pythona do kodu maszynowego. Kompilator ten jest napisany w języku C++; Kluczową część systemu kompilacji LLVM stanowi warstwa pośrednia, która pobiera kod pośredni (ang. *Intermediate form, IF*) z kompilatora dla określonego języka programowania i optymalizuje go. Może on zostać później przekonwertowany na kod asemblera dla konkretnej platformy sprzętowej, lub też pozostawiony do późniejszej kompilacji w locie w stylu języka Java. LLVM obsługuje kod pośredni kompilatora GCC, co umożliwia wykorzystanie szerokiej gamy już istniejących kompilatorów dla tamtego projektu. [28]

Numba kompiluje kod Pythona z LLVM do kodu, który może być natywnie wykonywany w środowisku wykonawczym. [28] Dzieje się tak poprzez dekorowanie funkcji Pythona, co pozwala użytkownikom tworzyć natywne funkcje dla różnych typów danych wejściowych lub tworzyć je w locie, co jest ukazane na listingu 5.1.

Dla porówniania, ten sam program napisany w języku CPython jest pokazany na listingu 5.3, a wynik działania tego programu jest ukazany na listingu 5.4.

Jak widać, Numba jest szybsza od CPythona o około tysiąc razy.

Listing 5.1. Przykładowy kod ukazujący dekorator kompilatora Numba

```
import numpy as np
from numba import double
from numba.decorators import jit

@jit(arg_types=[double[:,:], double[:,:]])
def pairwise_numba(X, D):
    M = X.shape[0]
    N = X.shape[1]
    for i in range(M):
        for j in range(M):
            d = 0.0
            for k in range(N):
                tmp = X[i, k] - X[j, k]
                d += tmp * tmp
            D[i, j] = np.sqrt(d)
```

Listing 5.2. Wynik działania programu z wykorzystaniem kompilatora Numba

```
In [2]: import numpy as np
In [3]: X = np.random.random((1000, 3))
In [4]: D = np.empty((1000, 1000))
In [5]: %timeit pairwise_numba(X, D)
100 loops, best of 3: 15.5 ms per loop
```

5.2. Cython

Cython jest nadzbiorem języka programowania Python, zaprojektowanym tak, aby zapewnić wydajność podobną do języka C dla kodu napisanego w języku Python z domieszką C. Cython jest językiem kompilowanym, który generuje moduły rozszerzeń CPython. Te moduły rozszerzeń mogą być następnie ładowane i używane przez zwykły kod Pythona za pomocą instrukcji import. Cython działa, tworząc standardowy moduł Pythona. Jednak zachowanie różni się od standardowego języka Python tym, że kod modułu, pierwotnie napisany w języku Python, jest tłumaczony na język C. [29] Chociaż wynikowy kod jest szybki, wykonuje wiele wywołań w bibliotekach CPython i bibliotekach standardowych CPython w celu wykonania programu. Wybór tego rozwiązania znacznie zaoszczędził na czasie tworzenia Cythona, ale moduły są zależne od interpretera Pythona i standardowej biblioteki. Cython ma interfejs do wywoływania procedur C / C ++ oraz możliwość zadeklarowania statycznego typu parametrów, zmiennych lokalnych i atrybutów klas. Program napisany w języku Cython, który implementuje ten sam algorytm, co odpowiedni program w języku Python, może zużywać mniej zasobów obliczeniowych, takich jak pamięć procesora i cykle przetwarzania, z powodu różnic między modelami wykonania CPython i Cython. Podstawowy program w języku Python jest ładowany i uruchamiany przez maszynę wirtualną CPython, więc zarówno środowisko wykonawcze, jak i sam program zuży-

Listing 5.3. Wynik działania programu z wykorzystaniem kompilatora Numba

```
import numpy as np

def pairwise_python(X, D):
    M = X.shape[0]
    N = X.shape[1]
    for i in range(M):
        for j in range(M):
            d = 0.0
            for k in range(N):
                tmp = X[i, k] - X[j, k]
                d += tmp * tmp
            D[i, j] = np.sqrt(d)
```

Listing 5.4. Wynik działania programu z wykorzystaniem kompilatora Numba

```
In [2]: import numpy as np
In [3]: X = np.random.random((1000, 3))
In [4]: D = np.empty((1000, 1000))
In [5]: %timeit pairwise_python(X, D)
1 loops, best of 3: 12.1 s per loop
```

wają zasoby obliczeniowe. Program Cython jest kompilowany do kodu C, który jest dalej kompilowany do kodu maszynowego, więc maszyna wirtualna jest używana tylko przez krótki czas, gdy program jest ładowany. [30]

Wydajność zależy zarówno od tego, jaki kod C jest generowany przez Cython, jak i od tego, jak ten kod jest kompilowany przez kompilator C.

Pliki Cython mają rozszerzenie .pyx. Najprościej mówiąc, kod Cythona wygląda dokładnie tak, jak kod Pythona. Jednakże, podczas gdy standardowy Python jest dynamicznie wpisywany, w Cythonie typy mogą być opcjonalnie podane, zapewniając lepszą wydajność, umożliwiając przekształcenie pętli w pętle C, jeśli to możliwe. przykład użycia Cythona jest ukazany na listingu 5.5.

Jak widać na listingu 5.6, Cython poradził sobie z przykładowym programem około 30% szybciej od Numby.

5.3. Optymalizacja kodu pośredniego

Szybkość Pythona jest ograniczona ze względu na jego interpreter, a zatem istnieje znacząca potrzeba aby zoptymalizować język. W tej części rozdziału autor przedstawi przykładowe techniki objemujące transformacje kodu pośredniego mogące przyspieszyć wykonywanie programu użytkownika.

Listing 5.5. Przykładowy kod ukazujący użycie języka Cython

```
cimport cython
from libc.math cimport sqrt

@cython.boundscheck(False)
@cython.wraparound(False)
def pairwise_cython(double[:, ::1] X, double[:, ::1] D):
    cdef int M = X.shape[0]
    cdef int N = X.shape[1]
    cdef double tmp, d
    for i in range(M):
        for j in range(M):
            d = 0.0
            for k in range(N):
                tmp = X[i, k] - X[j, k]
                d += tmp * tmp
            D[i, j] = sqrt(d)
```

Listing 5.6. Wynik działnia programu napisanego w Cythonie

```
In [2]: import numpy as np
In [3]: X = np.random.random((1000, 3))
In [4]: D = np.empty((1000, 1000))
In [5]: %timeit pairwise_numba(X, D)
100 loops, best of 3: 9.86 ms per loop
```

5.3.1. Stosowanie funkcji inline oraz rozwijanie pętli

Yosi Ben Asher i Nadav Rotem w swojej pracy *The Effect of Unrolling and Inlining for Python Bytecode Optimizations* zapronowali optymalizację kodu bajtowego. W początkowym etapie optymalizatora, kod bajtowy jest rozszerzany za pomocą funkcji otwarcia (ang. *function inlining*) i "rozwijania" pętli (ang. *loop unrolling*) . Drugi etap transformacji upraszcza kod bajtowy poprzez zastosowanie pełnego zestawu optymalizacji przepływu danych, w tym stałą propagacji, uproszczenia algebraiczne, eliminacja martwego kodu, propagacja kopiowania czy eliminacja wspólnych wyrażeń podrzędnych. [25] O ile te optymalizacje są znane, a ich mechanizm implementacji (analiza przepływu danych) jest dobrze rozwinięta, nie zostały one pomyślnie wdrożone w Pythonie ze względu na jego dynamiczne cechy, które uniemożliwiają ich użycie. Naukowcy zwrócili uwagę na znaczące efekty technik *unrolling*), a następnie *inlining*, a także możliwość zastosowania innych optymalizacji. Wyniki ich eksperymentów wskazują, że te optymalizacje mogą rzeczywiście zostać wdrożone i radykalnie poprawić czasy wykonania programów napisanych w jęyku Python.

Rozwijanie pętli zmniejsza całkowitą liczbę iteracji i może być użyte do wyeliminowania niektórych odwołań do tablicy. Ponadto rozwijanie w pętli zwiększa rozmiar ciał pętli, dając nowe możliwości dla innych optymalizacji, takich jak stała propagacja

(ang. *constant propagation*). Na przykład w języku C rozwijanie pętli *for (i = 1; i ¡n; i ++) A [i] = A [i - 1] + A [i - 2]* trzy razy wygeneruje poniższy kod, podstawi indeksy i użyje wartości tymczasowych (w przypadku, gdy zakres pętli jest małą stałą, można zastosować "pełne" rozwijanie, całkowicie eliminując pętlę):

Listing 5.7. Przykład rozwijania pętli w języku C

```
1  for(i=0;i<n-3;i+=3){
2    A[i] = A[i-1]+A[i-2];
3    A[i+1] = A[i]+A[i-1];
4    A[i+2] = A[i+1]+A[i];
5  }
6
7  int tmp1,tmp2,tmp3;
8  for(i=0;i<n-3;i+=3){
9    tmp1 = A[i-1];
10   tmp2=A[i] = tmp1+A[i-2];
11   tmp3=A[i+1] = tmp2+tmp1;
12   A[i+2] = tmp3+tmp2;
13 }
```

Rozwijanie pętli pozwala na uniknięcie pomyłek wyszukiwań w pamięci podręcznej (ang. *cache memory misses*) na różnych poziomach.

Chybienie pamięci podręcznej to stan, w którym dane żądane do przetwarzania przez komponent lub aplikację nie zostały znalezione w pamięci podręcznej. Powoduje ono opóźnienia w wykonaniu, wymagając od programu lub aplikacji pobrania danych z innych poziomów pamięci podręcznej lub pamięci głównej. Dla każdego nowego żądania procesor przeszuka główną pamięć podręczną, aby znaleźć żądane dane. Jeśli dane nie zostaną znalezione, uznaje się je za nie istniejące w pamięci podręcznej. Chybianie w pamięci podręcznej spowalnia cały proces, ponieważ po pominięciu pamięci podręcznej jednostka centralna (CPU) będzie szukała pamięci podręcznej wyższego poziomu, takiej jak L1, L2, L3 i pamięci o dostępie bezpośrednim (ang. *Random Access Memory, RAM*) dla tych danych [31].

Poniżej podane zostaną uśrednione czasy dostępu do poszczególnych rodzajów pamięci:

1. pamięć podręczna typu L1 : 1 nanosekunda
2. pamięć podręczna typu L2 : 4 nanosekundy
3. pamięć typu RAM: 100 nanosekund

Inną znaną optymalizacją jest funkcja otwarcia (ang. *function inlining, inline expansion*, tłumaczenie autorstwa prof. Jana Bieleckiego), w której wywołanie funkcji jest zastępowane przez kopię jej ciała i wartości parametrów i podstawianie jako wartości początkowe poprzednich parametrów. Główną wadą funkcji inline jest zwiększona liczba błędów pamięci podręcznej spowodowanych przez zwiększony rozmiar kodu. W języku Python rozmiar kodu nie jest brany pod uwagę przez co funkcja wbudowana w Pythonie może być korzystna. Podobnie jak rozwijanie, zwiększa możliwości stosowania innych optymalizacji. Na listingu 5.8 ukazany jest pseudokod pokazujący ideę funkcji inline.

Listing 5.8. Przykładowy pseudokod ukazujący ideę funkcji otwartych

```
def f(x):
  v = 5
  if (x==9):
    return x + v
  return x*3

def g():
  sum = 0
  for i in xrange(n):
    sum += f(7+i)
   return sum

def major_1():
  sum = 0
  for i in xrange(n):
  $inline_x = 7+i
  $local_v = 5
  if ($inline_x==9):
    _inline_return=x+$local_v
    *goto END_TAG
    _inline_return = x*3
    *goto END_TAG
    END_TAG
    sum += _inline_return
  return sum
```

Podobnie jak rozwijanie pętli, stosowanie funkcji otwartych ma kilka problematycznych przypadków:

1. Funkcje zawierające instrukcję *yield* nie mogą zostać zoptymalizowane, ponieważ są zaprojektowane tak, aby były wywoływane kilka razy i nadal wykonywane za każdym razem od ostatniego miejsca, w którym zostały zatrzymane. Dodatkowo funkcję, która ma instrukcję *yield* można wywołać z więcej niż jednego miejsca w programie.
2. Funkcje globalne, które są modyfikowane w czasie wykonywania, mogą nie zostać zoptymalizowane, ponieważ za każdym razem, gdy wywoływana jest inna funkcja, optymalizator nie może wstawiać więcej niż jednej funkcji dla każdej lokalizacji.
3. Funkcje wbudowane (ang. *built-in functions*) które są zaimplementowane w języku C nie mogą zostać zoptymalizowane.

5.3.2. Optymalizacje przepływu danych

Gdy kod bajtowy zostanie poddany dwóm wyżej wymienionym ulepszeniom, możliwa jest dalsza optymalizacja. Z racji, że interpreter kodu bajtowego jest zaimplementowany w oprogramowaniu, w przeciwieństwie do kodu maszynowego, optymalizacje przydziału mocy obliczeniowej nie są korzystne w taki sam sposób, w jaki byłyby one

dla kodu assemblerowego. Druga faza, po rozwinięciu pętli i "otwieraniu" funkcji, koncentruje się na upraszczaniu kodu bajtowego za pomocą dobrze znanych oraz nowych optymalizacji. [17]

Gergö Barany w swoich artykułach *Python Interpreter Performance Deconstructed* oraz *pylibjit: A JIT Compiler Library for Python* opisał napisany przez siebie kompilator czasu rzeczywistego *pylibjit*, dzięki któremu możliwe jest zastosowanie szeregu działań doskonalących kod pośredni oraz wydajność języka Python:

1. Optymalizacja zliczania referencji (ang. *reference counting*) usuwa niepotrzebne operacje inkrementacji i dekrementacji liczników referencyjnych. Pary takich operacji na tym samym obiekcie w bloku podstawowym, bez interweniujących operacji, które mogłyby zmniejszyć licznik referencyjny, są zbędne; można je zidentyfikować i wyeliminować podczas kompilacji. Operacje zliczania referencji, które nie są usuwane, są generowane jako funkcje otwarte, jak w CPython (który używa makr C do inliningu).
2. Statyczne przetwarzanie arytmetycznych opiera się na typach podanych przez użytkownika w celu identyfikacji arytmetyki na opakowanych obiektach (ang. *boxed numbers*) typów *int* lub *float* i generowaniu wywołań bezpośrednio do funkcji specyficznych dla typu, a nie do generycznych funkcje *PyNumber*, które wykonują dynamiczną identyfikację w czasie wykonywania programu.
3. Rozpakowywanie liczb (ang. *Unboxing numbers*) używa adnotacji typu, aby bezpośrednio używać operacji maszynowych na liczbach całkowitych i liczbach zmiennoprzecinkowych, które pasują do rejestrów maszynowych. Zapobiega to przechowywaniu każdej liczby w stosie i zarządzaniu tą pamięcią. W szczególnym przypadku, funkcja kompilatora również zamienia zwyczajne pętle postaci *for i w range(n)*, które normalnie używają iteratora do wyliczenia opakowanych liczb od 0 do n, do zoptymalizowanych, odpakowanych pętli, jeśli licznik *i* jest odpakowaną liczbą maszynową.
4. Odpakowywanie kontenerów, również sterowane przez adnotacje typu, specjalizuje się w dostępach do odczytu i zapisu do obiektów list i tablic oraz do sprawdzania granic (ang. *bound check*) jeśli indeks jest odpakowaną maszynową liczbą całkowitą. Operacje opakowania / rozpakowywania dostępu do tablicy są również eliminowane, jeśli operand jest w inny sposób używany bez opakowania.
5. Usuwanie wywoławczych ramek stosu wykonuje wywołania między dowolnymi dwiema skompilowanymi funkcjami bezpośrednio i wykorzystuje wywoływania maszynowe zamiast dynamicznie przydzielanych ramek stosu. Oznacza to jednak, że nie jest możliwe skompilowanie współprogramów (ang. *coroutines*) lub przechwytywanie wyjątków.
6. Wczesne wiązanie wywołań funkcji (ang. *early function call binding*) rozpoznaje adresy skompilowanych lub wbudowanych funkcji lub metod w czasie kompilacji (kierując się adnotacjami typu dla obiektów odbiorczych metod) i generuje instrukcje maszynowe procesora.

Rozdział 6

Aplikacja porównująca kody pośrednie

W tym rozdziale zostanie opisana aplikacja porównująca kody pośrednie dla języka Python w wersjach *2.7.12* oraz *3.5.2*.

6.1. Użyte narzędzia i biblioteki

6.1.1. PyQt i QtDesigner

PyQt to Pythonowe API dla przenośnego frameworka interfejsów graficznych Qt działającego pod Linuksem, systemami Unixowymi, Mac OS X i MS Windows. PyQt dostępne jest na dwóch licencjach w zależności od zastosowań. Dla projektów o otwartym źródle dostępna jest na licencji *GPL2*, oraz dla projektów komercyjnych na licencji komercyjnej. Do napisania opisywanej aplikacji zostało wykorzystane PyQt w wersji 4.

6.1.2. Virtualenv

Virtualenv to narzędzie do tworzenia izolowanych środowisk języka Python. *Virtualenv* tworzy folder zawierający wszystkie niezbędne pliki wykonywalne do użycia pakietów, których potrzebuje projekt Python. Wirtualne środowisko, w uproszczeniu, jest odizolowaną roboczą kopią Pythona, która pozwala pracować nad konkretnym projektem bez obawy o wpływanie na inne projekty. Umożliwia wiele równoległych instalacji Pythona, po jednym dla każdego projektu. W rzeczywistości nie instaluje oddzielnych kopii Pythona, ale zapewnia sprytny sposób na izolowanie różnych środowisk projektowych.

W projekcie aplikacji porównującej kody pośrednie Pythona w wersji 2.7.12 oraz 3.5.2 są używane dwa środowiska wirtualne, jedno dla każdej wersji interpretera, nazwane odpowiednio *venv_analyzer_2* i *venv_analyzer_3*. W tych właśnie środowiskach wirtualnych są uruchamiane poszczególne programy wybrane przez użytkownika programu. Wirtualne środowisko zapewnia separację od interfejsu graficznego i brak dodatkowych zadań dla wyizolowanego interpretera.

Listing 6.1. Manualna aktywacja środowisk wirtualnych wykorzystywanych w aplikacji

```
user@user:~/.../bytecode_analyzer$
source venv_analyzer_2/bin/activate
(venv_analyzer2) user@user:~/.../bytecode_analyzer$
python --version
Python 2.7.12
(venv_analyzer2) user@user:~/.../bytecode_analyzer$ deactivate
user@user:~/.../bytecode_analyzer$
source venv_analyzer_3/bin/activate
(venv_analyzer3) user@user:~/.../bytecode_analyzer$
python --version
Python 3.5.2
```

6.1.3. Moduł dis

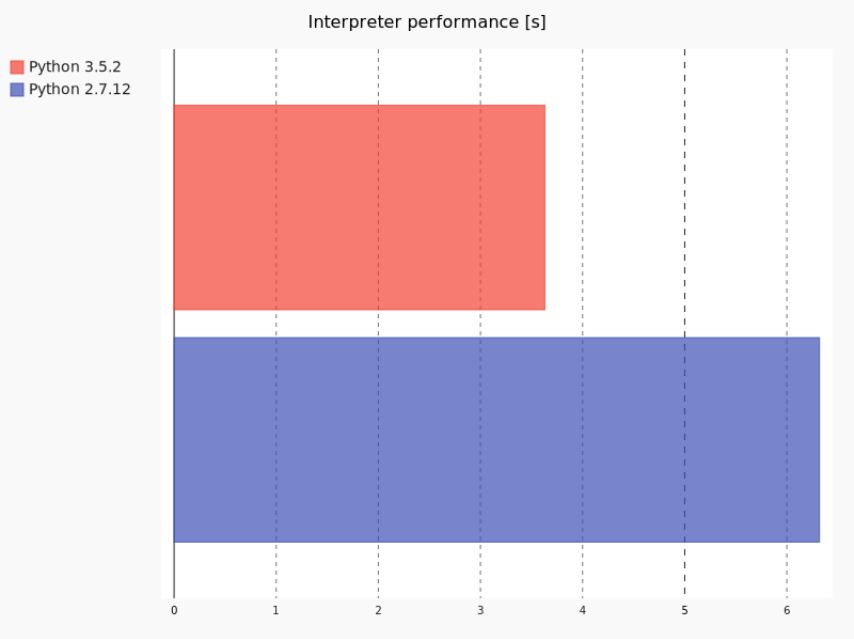
Opisywany wcześniej moduł dis jest wykorzystywany do uzyskiwania kodów pośrednich analizowanych programów. Po wykonaniu testowanego skryptu aplikacja wymusza kompilację i stworzenie pliku z rozszerzeniem *.pyc* za pomocą dyrektywy interpretera *python -m compileall nazwa_wykonywanego_programu*. Stworzony plik, zawierający kod pośredni dla danego interpretera, jest otwierany i dzielony na sekcje (*magic tag*, czas utworzenia / modyfikacji danego pliku, wielkość kodu w przypadku Pythona 3.5.2 oraz sam kod pośredni), po czym wszystkie informacje są przekierowywane do pliku wyjściowego. Opisywana funkcjonalność jest ukazana na listingu 6.2.

6.1.4. Pygal

Pygal jest biblioteką języka Python służącą do dynamicznej generacji wykresów w formacie SVG, rozprzestrzeniany pod licencją *LGPLv3*. W opisywanej aplikacji służy do przedstawiania graficznej reprezentacji czasów wykonywania programów, na przetwarzanie których poświęciła maszyna wirtualna danej wersji języka. Rysunek 6.1 ukazuje przykładowy wykres tworzony na potrzeby aplikacji porównującej kody pośrednie.

6.1.5. Komenda time

Time jest poleceniem występującym w systemach Unixowych, służącym do pomiaru czasu wykonywania danego programu. Wyświetlone statystyki w podstawowym trybie pracy pokazują łączny czas potrzebny na uruchomienie polecenia, ilość czasu spędzonego w trybie użytkownika (ang. *user mode*) oraz ilość czasu spędzonego w jądrze systemu (ang. *kernel mode*). W aplikacji służy do wyświetlania informacji w trybie *verbose*. Listing 6.3 pokazuje przykładowe informacje uzyskane za pomocą polecenia *time*.

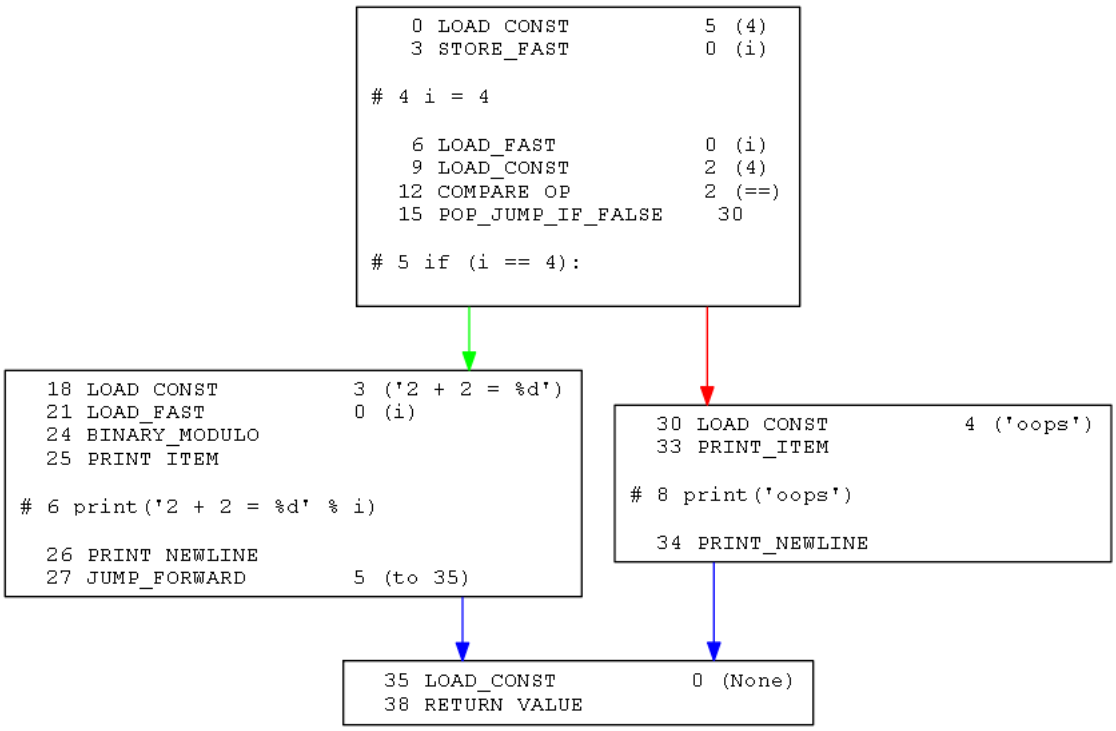


Rysunek 6.1. Przykładowy wykres stworzony za pomocą biblioteki Pygal

Źródło: Własne

6.1.6. Flare bytecode graph

Flare bytecode graph to moduł przeznaczony do modyfikacji kodu bajtowego Pythona. Umożliwia dodawanie lub usuwanie instrukcji z łańcucha kodu bajtowego Python. Zmodyfikowany kod bajtowy może być refaktoryzowany w celu korekcji przesunięć i utworzenia nowego kodu funkcjonalnego kodu. Możliwe jest również tworzenie schematów sterowania (ang. *Control Flow Graph*) za pomocą GraphViz. Wynik disasemblacji na wykresie może obejmować dane wyjściowe z prostego dekompilatora typu *peephole* (ang. *peephole decompiler*). W opisywanym programie moduł służy do generacji schematów sterowania, po czym są one przedstawiane w środowisku graficznym aplikacji. Przykładowy kod oraz wynik jego działania są ukazane na listingu 6.4 oraz rysunku 6.2.



Rysunek 6.2. Przykładowy graf CFG stworzony za pomocą biblioteki flare bytecode graph

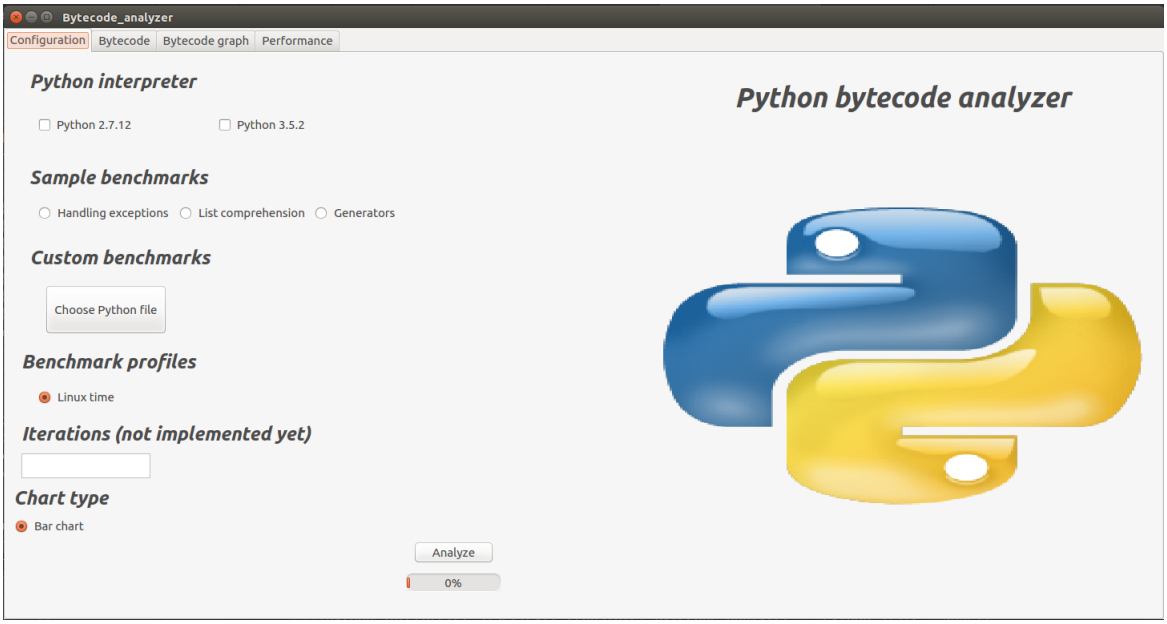
Źródło: https://raw.githubusercontent.com/fireeye/flare-bytecode_graph/master/docs/example_graph.png

6.2. Moduły aplikacji

Poniższa sekcja opisuje poszczególne moduły, na które został podzielony graficzny interfejs użytkownika (ang. *graphical user interface, GUI*).

6.2.1. Interfejs konfiguracyjny

W interfejsie konfiguracyjnym użytkownik ma możliwość wybrania interesujących go opcji porównania kodów pośrednich. Jako pierwsze, należy wybrać wersje interpretera języka, dla których zostanie wykonany test (sekcja *Python interpreter*). Następnie, użytkownik ma do wyboru kilka przykładowych programów (ang. *sample benchmarks*). W przypadku, gdy użytkownik chce wykonać test na własnych programach, również istnieje taka możliwość - należy wtedy skorzystać z sekcji *Custom benchmarks*. W chwili pisania pracy dostępny jest tylko jeden sposób mierzenia czasu wykonania programów, czyli mierzenie z pomocą Linuxowej komendy *time*, oraz typ generowanego wykresu. Moduł konfiguracyjny jest ukazany na rysunku 6.3.



Rysunek 6.3. Moduł konfiguracyjny aplikacji

Źródło: Własne

6.2.2. Moduł porównujący kody pośrednie

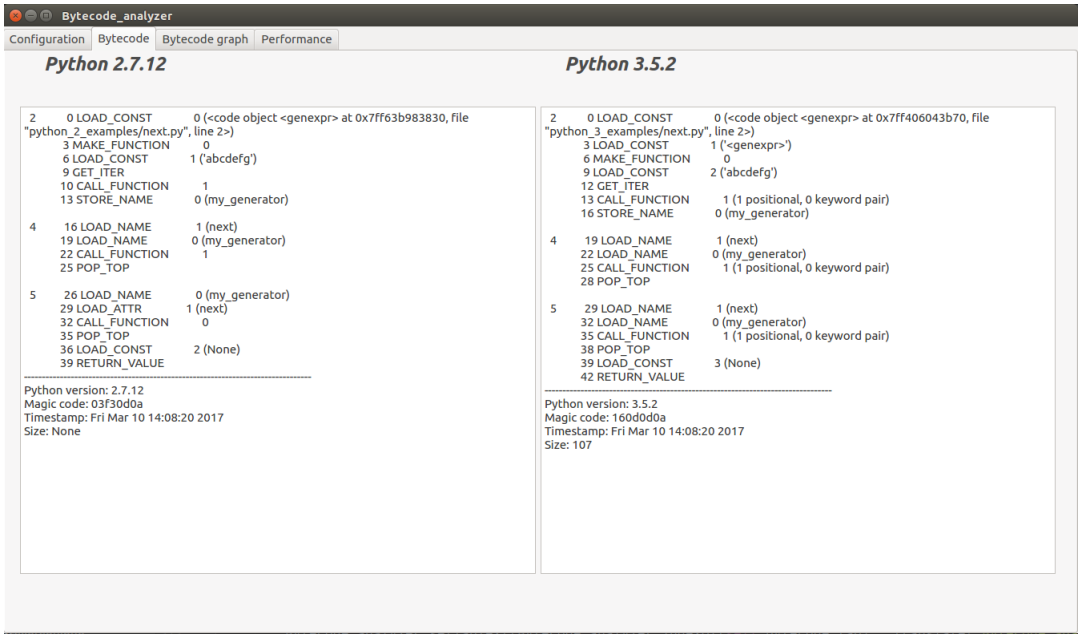
W module porównującym kody pośrednie są ukazane kody bajtowe dla wybranych wcześniej interpreterów; wyodrębnione są sekcje samego kodu, wersji, magicznego kodu charakterystycznego dla każdego interpretera (ang. *magic code*), czas kompilacji programu (ang. *timestamp*) oraz, jeśli to możliwe, rozmiar kodu pośredniego. Rysunek 6.4 przedstawia moduł porównujący kody pośrednie.

6.2.3. Moduł porównujący diagramy CFG

Moduł porównujący diagramy CFG (ang. *Control Flow Graph*) zawiera graficzne reprezentacje stworzone przy pomocy modułu *Flare bytecode graph*. Dzięki takiemu zestawieniu możliwe jest wygodniejsze porównanie większych programów, oraz podzielenie ich na sekcje, przez co analiza i porównywanie kodów pośrednich oraz struktury logicznej testowanych programów jest łatwiejsza. Rysunek 6.5 przedstawia porównanie dwóch grafów dla starszej i nowszej wersji interpretera.

6.2.4. Moduł prównujący prędkość wykonywania programów

Ostatnim modułem programu jest proste porównanie efektywności czasowej każdego interpretera; dzięki bibliotece Pygal oraz komendzie systemu Linux *time* możliwe jest także porównanie interpreterów pod względem szybkości działania. Rysunek 6.6 przedstawia porównanie czasów wykonania programu dla dwóch wersji interpretera.



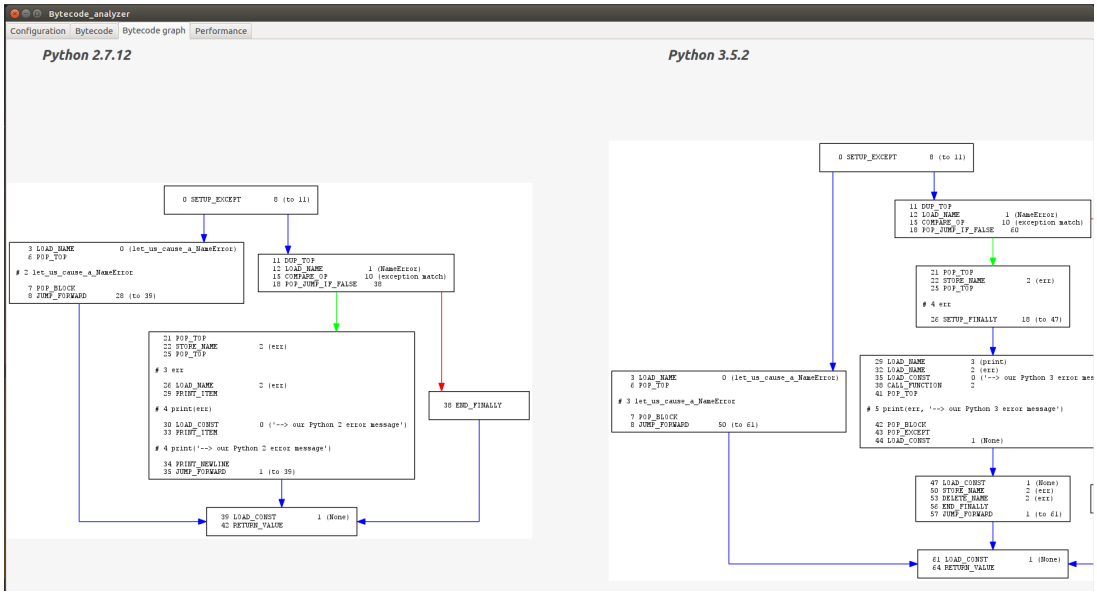
Rysunek 6.4. Moduł porównujący kody pośrednie

Źródło: Własne

6.3. Wnioski i dalsze plany

Istniejąca aplikacja jest w stanie porównać kody pośrednie dla języka Python w wersjach 2.7.12 oraz 3.5.2. Dwie wersje interpretera różnią się między sobą, między innymi z powodu zmienionych, dodanych i usuniętych instrukcji kodu pośredniego, czy dodanie kolejnych informacji do generowanego pliku *.pyc*. Przykładem zmienionej instrukcji kodu bajtowego może być *BUILD_MAP*, poprzez którą zmieniony został mechanizm tworzenia słowników na stosie. Dużą ilość zmian w kodzie pośrednim spowodowało dodanie nowych instrukcji do zbioru, w celu dodania nowych funkcjonalności do języka Python. Przykładem tutaj mogą być zmiany w funkcjach *range()* i *xrange()* czy wprowadzenie zwracania iterowalnych obiektów zamiast list. Python w wersji 3.5.2 jest w wielu przypadkach wolniejszy od interpretera w wersji 2.7.12; widoczne jest to na przykład we wspomnianej funkcji *range()*, która mimo, że generuje bardzo podobny kod pośredni, wykonuje się dłużej w nowszej wersji interpretera. W tym przypadku, różnica polega na zmienionej implementacji typu liczb całkowitych (ang. *integers*) - Python w wersji 3.5.2 używa wyłącznie typu liczbowego o arbitralnym rozmiarze (ang. *arbitrary-sized integer type*) (odpowiednik *long* w 2.x), podczas gdy w Pythonie 2.x dla wartości do sys.maxint używany jest prostszy typ int, który używa typu *long* znanego z języka C. Podobnych różnic jest wiele więcej, wobec czego analiza kodu bajtowego jest tylko jedną z czynności potrzebnych, aby w pełni porównać dwie wersje interpretera. Należy jednak pamiętać, że wyniki wydajnościowe zmieniają się co wersję interpretera, a ulepszenia są cały czas wprowadzane, dzięki czemu nowe wersje Pythona są coraz szybsze.

Istnieje wiele możliwości ulepszenia i uproszczenia interpretera; dowodami na to są liczne prace i artykuły naukowe, których mnogość pozwala na wysnucie wniosku, że



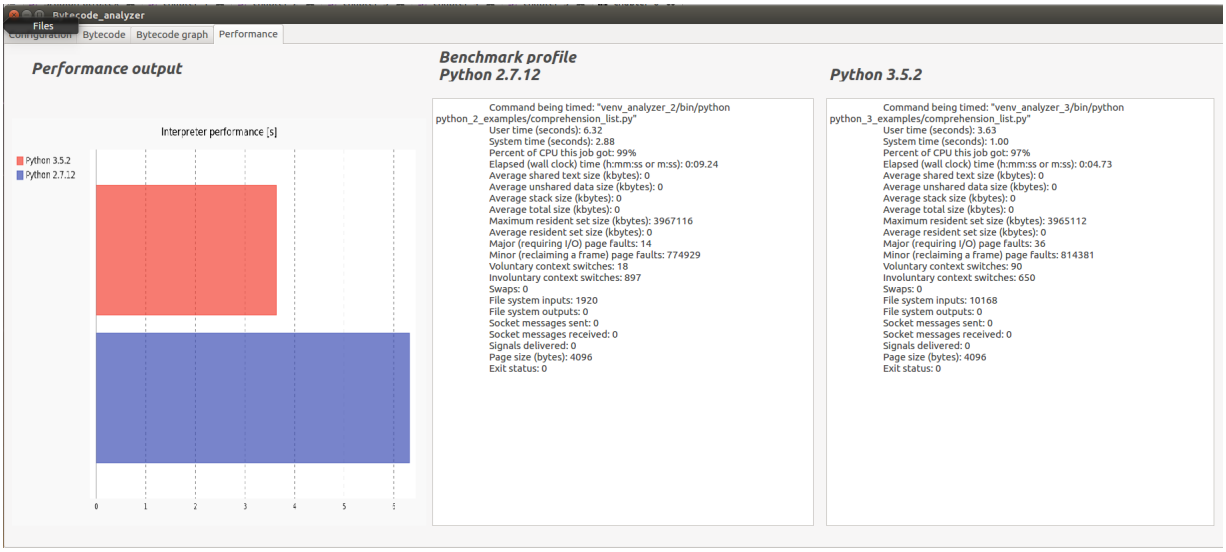
Rysunek 6.5. Porównanie grafow CFG dla dwóch wersji interpretera języka Python

Źródło: Własne

wydajność języka Python powinna być jednym z priorytetów osób projektujących ten język. Mimo, że wydajność Pythona nie jest zachwycająca, należy również pamiętać, że nie został on stworzony w celu szybkiego działania, lecz szybkiego programowania i adaptacji programisty do cudzego kodu; mnogość bibliotek i narzędzi dla języka jest kolejnym dowodem na to, że oszczędzenie czasu programistów jest głównym celem twórców języka Python. Twórca Pythona, Guido van Rossum przyznał, że głównymi celami przy tworzeniu języka Python są: łatwość i intuicyjna składnia, zrozumiały kod w języku angielskim, oparty na zasadzie open source, aby każdy mógł wnieść wkład w jego rozwój, czy przydatność do codziennych celów. Istnieje jednak co najmniej kilka narzędzi pozwalających na przyspieszenie języka Python, takich jak Pypy, Numba czy Cython. W większości narzędzia te oparte są na składni języka Python, wobec czego adaptacja do nowej składni nie jest potrzebna. W przypadku Cythona potrzebne jest zapoznanie się ze specyficzną składnią z pogranicza C i Pythona, opłaca się to jednak ze względu na gigantyczne przyspieszenie względem implementacji CPython.

Istniejąca aplikacja porównująca kody pośrednie może zostać znacznie ulepszona; dodanie modułów porównujących także PyPy oraz Numbę musiałoby uwzględniać znacznie więcej czynników - w obydwu przypadkach generowany kod bajtowy jest bardzo podobny do tego z implementacji CPython; zdecydowana większość operacji przyspieszajacych wykonanie kodu jest wykonywana w kilku etapach, przez co trywialne porównanie kodów bajtowych nie byłoby poprawnym rozwiązaniem. [32] [33]

Następną funkcjonalnością, która mogłaby wzbogacić istniejącą implementację byłoby dodanie modułu *byteplay* [33] lub wykorzystanie w całości możliwości modułu *flare-bytecode graph*. Moduły te pozwalają na dodawanie, usuwanie oraz modyfikację istniejącego kodu bajtowego. Funkcjonalność ta pomogłaby w eksperymentowaniu z instniejącym kodem pośrednim i sprawdzaniu, jak wprowadzone zmiany wpływają na efektywność lub prędkość wykonywania takiego kodu.

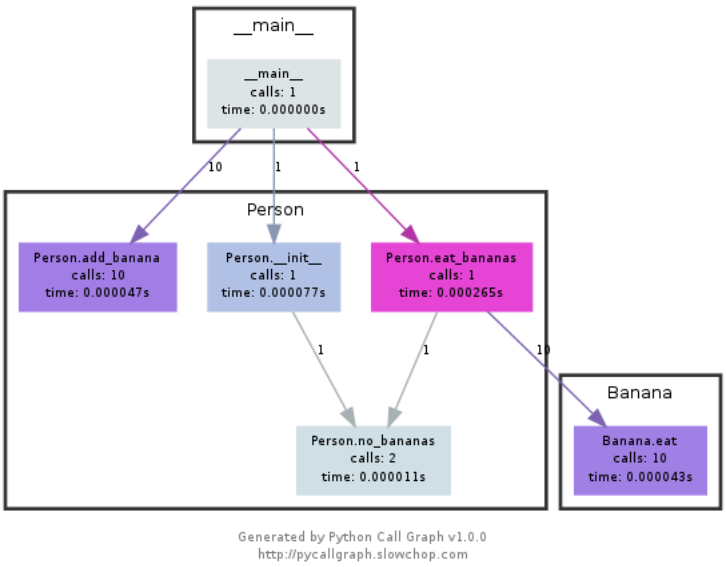


Rysunek 6.6. Moduł porównujący czasy wykonania programów dla każdej wersji interpretera

Źródło: Własne

Kolejną cechą, która zostanie w przyszłości dodana, jest szersza analiza wykonywania kodu pośredniego pod kątem wydajności; inkorporacja modułu *Python Call Graph* wspomogłaby graficzną analizę wyprodukowanego kodu pośredniego wskazując czasy wykonania poszczególnych modułów programu napisanego w języku CPython [34]. Przykładowy graf wyprodukowany przez wspomniany moduł ukazuje rysunek 6.7.

W planach znajduje się także wprowadzenie analizy objętości generowanego kodu pośredniego w stosunku do wykonanej ilości obliczeń; innymi słowy, moduł taki obliczałby efektywność danych instrukcji bajtowych na podstawie wcześniej założonych wag dla różnych rodzajów instrukcji. Jangqiu Shao i Yingxu Wang w swojej pracy *A new measure of software complexity based on cognitive weights* [35] wykonali analizę programów napisanych w języku Java, wyznaczając złożoność wygenerowanego kodu. Podobna funkcjonalność może także zostać wprowadzona dla kodu bajtowego języka CPython.



Rysunek 6.7. Przykładowy graf wygenerowany przy pomocy modułu Python Call Graph

Źródło: http://pycallgraph.slowchop.com/en/master/examples/basic.html

Listing 6.2. Fragment aplikacji ukazujący operacje wykonywane na plikach z rozszerzeniem .pyc

```
def analyze_pyc_file(self, path):
    """Read and redirect a content of the Python's bytecode
        in a .pyc filename to output file.
    """
    filename = open(path, 'rb')

    magic = filename.read(4)
    timestamp = filename.read(4)
    size = None

    if sys.version_info.major == 3 and sys.version_info.minor >= 3:
        size = filename.read(4)
        size = struct.unpack('I', size)[0]

    code = marshal.load(filename)
    self.serialize(code)

    magic = binascii.hexlify(magic).decode('utf-8')
    timestamp = time.asctime(
                time.localtime(struct.unpack('I', b'D\xa5\xc2X')[0]))
    if sys.version_info.major == 3 and sys.version_info.minor >= 3:
        commons.redirect_stdout(filename=cfg.PYC_OUTPUT_FILE_3,
                                func=self.generate_pyc_output,
                                code_obj=code, magic=magic,
                                timestamp=timestamp, size=size)
        self.bytecode_graph(code, '3')
    else:
        commons.redirect_stdout(filename=cfg.PYC_OUTPUT_FILE_2,
                                func=self.generate_pyc_output,
                                code_obj=code, magic=magic,
                                timestamp=timestamp, size=size)
        self.bytecode_graph(code, '2')
    filename.close()
```

Listing 6.3. Przykładowe informacje uzyskiwane podczas wykonywania programu z pomocą polecenia time

```
Command being timed: "venv_analyzer_3/bin/python python_3_examples/comprehen
User time (seconds): 3.63
System time (seconds): 1.00
Percent of CPU this job got: 97%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:04.73
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 3965112
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 36
Minor (reclaiming a frame) page faults: 814381
Voluntary context switches: 90
Involuntary context switches: 650
Swaps: 0
File system inputs: 10168
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
```

Listing 6.4. Przykładowy kod generujący graf CFG

```
import bytecode_graph


def Sample():
    i = 2 + 2
    if i == 4:
        print "2 + 2 = %d" % i
    else:
        print "oops"

bcg = bytecode_graph.BytecodeGraph(Sample.__code__)

graph = bytecode_graph.Render(bcg, Sample.__code__).dot()

graph.write_png('example_graph.png')
```

Bibliografia

[1] Python Software Foundation, *The python language reference / alternate implementations*, Czerwiec 2016. https://docs.python.org/release/3.5.2/reference/introduction.html#alternate-implementations

[2] Python Software Foundation, *Design of the CPython Compiler*, Czerwiec 2016 https://www.python.org/dev/peps/pep-0339/

[3] Python Software Foundation, *ast — Abstract Syntax Trees*, Czerwiec 2016 https://docs.python.org/3.5/library/ast.html

[4] Python Software Foundation, *Lexical analysis*, Czerwiec 2016 https://docs.python.org/release/3.5.2/reference/lexical_analysis.html

[5] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Chris S. Serra, *The Zephyr Abstract Syntax Description Language.*, Wrzesień 1997

[6] Python Software Foundation, *Kod źrodłowy języka Python w wersji 3.5.2*, 2016 https://github.com/python/cpython/blob/3.5/Python/compile.c

[7] Python Software Foundation, *dis — Disassembler for Python bytecode*, 2016 https://docs.python.org/3.5/library/dis.html

[8] Jackson Lee Salling, *Control flow graph visualization and its application to coverage and fault localization in Python*, Maj 2015 https://repositories.lib.utexas.edu/bitstream/handle/2152/32311/SALLING-MASTERSREPORT-2015.pdf

[9] Python Software Foundation, *Memory Management*, 2016 https://docs.python.org/3.5/c-api/memory.html

[10] Nadejda Ismailova, *Development of a Data Provenance Analysis Tool for Python Bytecode*, 1 Maj 2015

[11] Guido van Rossum, *What's New In Python 3.0*, 14 Luty 2009 https://docs.python.org/3.0/whatsnew/3.0.html

[12] Joshua Landau, *PEP 448 – Additional Unpacking Generalizations*, 29 Czerwiec 2013 https://www.python.org/dev/peps/pep-0448/

[13] Micha Gorelick, Ian Ozsvald *High Performance Python*, O'Reilly, 21 Wrzesień 2014

[14] Python Software Foundation *Extending Python with C or C++*, 2016 https://docs.python.org/3.5/extending/extending.htmlreference-counts

[15] Python Software Foundation *gc — Garbage Collector interface*, 2016 https://docs.python.org/3.5/library/gc.htmlmodule-gc

[16] Richard Jones, Rafael Lins *Collection: Algorithms For Automatic Dynamic Memory Mamagement*, John Wiley Sons, 1996 ISBN 0-471-94148-4

[17] Gergö Barany *Python Interpreter Performance Deconstructed*, Vienna University of Technology, 11 Czerwiec 2014

[18] Gergö Barany *pylibjit: A JIT Compiler Library for Python*, Vienna University of Technology, Marzec 2014

[19] Stefan Brunthaler *Speculative Staging for Interpreter Optimization*, University of California, Irvine, 8 Październik 2013

[20] The Scipy community *NumPy C Code Explanations*, The Scipy community, Czerwiec 2017 https://docs.scipy.org/doc/numpy-1.13.0/reference/internals.code-explanations.htmlmemory-model

[21] Raymond Hettinger *PEP 289 – Generator Expressions*, Python Software Foundation, Czerwiec 2017 https://www.python.org/dev/peps/pep-0289/

[22] Stefan Brunthaler *Efficient Interpretation using Quickening*, Vienna University of Technology, Marzec 2014 https://www.sba-research.org/wp-content/uploads/publications/dls10.pdf

[23] C. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, A. Rigo *Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages*, Heinrich-Heine-Universität Düsseldorf, STUPS Group, Styczeń 2015

[24] R. Meier, A. Rigo *A Way Forward in Parallelising Dynamic Languages*, Department of Computer Science ETH Zürich, pypy.org, Wrzesień 2013

[25] Y.B. Asher, N. Rotem *The Effect of Unrolling and Inlining for Python Bytecode Optimizations*, CS department, Haifa University, Maj 2009

[26] R. Power, A. Rubinsteyn *How fast can we make interpreted Python?*, New York University , 13 Sierpień 2013

[27] Społeczność Pypy *Pypy documentation*, Pypy.org , Listopad 2015 http://doc.pypy.org/en/latest/

[28] S.K. Lam, A. Pitrou, S. Seibert *Numba: A LLVM-based Python JIT Compiler*, Continuum Analytics, Styczeń 2014

[29] S. Behnel, R. Bradshaw , C. Citro *Cython: The Best of Both Worlds*, Computing in Science and Engineering, vol. 13.2 (2011), pp. 31-39

[30] Cython Community *Building Cython code*, Cython 0.28a0 documentation, 24 Kwiecień 2017 https://cython.readthedocs.io/en/latest/src/quickstart/build.html

[31] J. Dundas, T. Mudge *Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss*, Int. Conf. on Supercomputing, Lipiec 1997

[32] Anaconda Developers *Numba architecture*, Anaconda Inc., 2012 https://numba.pydata.org/numba-doc/dev/developer/architecture.html

[33] PyPy community *Bytecode Interpreter*, The PyPy Project, 2017 http://doc.pypy.org/en/latest/interpreter.html

[33] NoamYoravRaphael *byteplay Module Documentation*, Byteplay, Luty 2014 https://wiki.python.org/moin/ByteplayDoc

[34] Gerald Kaszuba *Python Call Graph*, 2013 http://pycallgraph.slowchop.com/en/master/

[35] Jingqiu Shao, Yingxu Wang *A new measure of software complexity based on cognitive weights* Can. J. Elect. Comput., Kwiecień 2013

Spis rysunków

Spis listingów