Hello World pod lupą

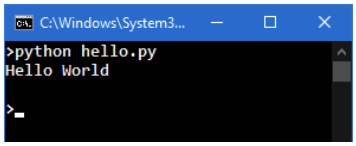
Pierwszym krokiem w klasycznej ścieżce edukacji przyszłych programistów jest stworzenie programu wypisującego – najczęściej w konsoli – tekst „Hello, World!”. Sam program jest z definicji banalny, ale to, co dzieje po jego uruchomieniu – już nie do końca. W tym artykule prześledzimy ścieżkę wykonania mini-programu „Hello World” napisanego w Pythonie, zaczynając od pojedynczego wywołania wysokopoziomowej funkcji print, poprzez kolejne poziomy abstrakcji interpretera, systemu operacyjnego i sterowników graficznych, a kończąc na wyświetleniu odpowiednich pikseli na ekranie. Skupimy się przy tym na systemie Windows. Jak się okaże, ścieżka ta sama w sobie nie jest ani prosta, ani krótka, ale zdecydowanie bardzo ciekawa.

KOD W PYTHONIE

Kod, od którego zaczniemy, jest banalny:

```
print("Hello World")
```

Efekt jego działania jest zarówno przewidywalny, jak i oczywisty:



Co jednak sprawia, że nasz komputer w efekcie wykonania powyższego programu uznaje za stosowne zmienić kolor kilkuset wybranych pikseli na ekranie?

Pierwszym krokiem okazuje się być kompilacja wskazanego pliku zawierającego nasz kod źródłowy (*hello.py*). Niektórzy czytelnicy mogą czuć się zaskoczeni już w tym momencie: „Ale chwila, czy Python – w przeciwieństwie do C czy C++ – nie jest czasem językiem interpretowanym?”. I faktycznie, Python często jest nazywany językiem skryptowym, a te, z definicji, nie powinny być kompilowane, czyż nie?

W praktyce wiele popularnych języków skryptowych, jak na przykład PHP, Ruby, Lua, JavaScript, Perl czy właśnie Python, są kompilowane do swoich własnych wariantów kodu bajtowego (ang. *bytecode*), czyli formy binarnej, która – mimo iż jest niekompatybilna z językiem maszynowym prawdziwych procesorów[1] – jest dużo łatwiejsza do szybkiej interpretacji i wykonania niż czysty kod źródłowy. Python w tym przypadku jest językiem o tyle wdzięcznym, że udostępnia moduły pozwalające na wgląd w poszczególne części tego procesu z poziomu samego języka.

Dla przypomnienia, proces kompilacji – w sporym uproszczeniu – można sprowadzić do trzech kroków:

- **analizy leksykalnej** (wykonywanej przez *lexer*), której wynikiem jest lista tokenów,
- **analizy składniowej** (wykonywanej przez *parser*), której wynikiem jest drzewo wyrażeń (AST, *Abstract Syntax Tree*),
- oraz **generowania kodu** – w naszym przypadku bajtowego.

1. Z drobnym wyjątkiem w postaci Jazelle DBX, czyli dodatkowego trybu w niektórych starszych procesorach z rodziny ARM, który umożliwiał wykonanie kodu bajtowego Java.

Efekt analizy leksykalnej możemy obejrzeć, korzystając z uruchomionego z linii poleceń modułu `tokenize`, czego wynikiem będzie wypisanie listy tokenów, której poszczególne wiersze zawierają pozycję danego tokena w pliku, jego typ oraz ewentualną zawartość tekstową.

```
>python -m tokenize hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,5:            NAME           'print'
1,5-1,6:            OP             '('
1,6-1,19:           STRING         '"Hello World"'
1,19-1,20:          OP             ')'
1,20-1,21:          NEWLINE        '\n'
2,0-2,0:            ENDMARKER      ''
```

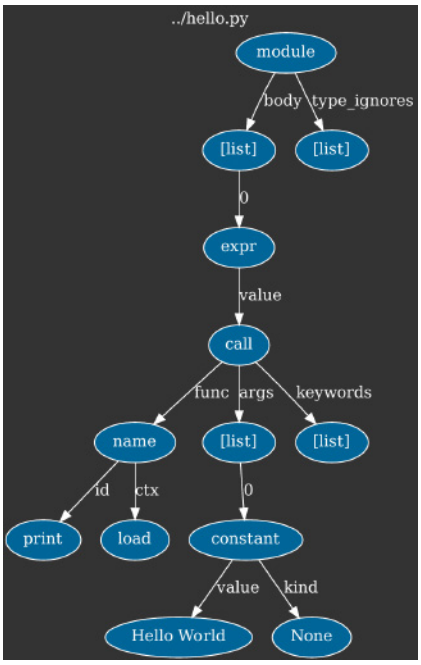
Nasz prosty Hello World składa się z niewielu tokenów: nazwy `print`, operatorów `(` i `)` oraz literału tekstowego `"Hello World"`. Oprócz nich jest jeszcze nieistotny w tym przypadku znak nowej linii, a także tokeny zawierające metadane, takie jak użyte w pliku źródłowym kodowanie czy znacznik końca danych.

Tokeny są następnie przekazywane do parsera, który, korzystając z zasad gramatycznych, generuje drzewo AST. Wynik działania parsera możemy obejrzeć, korzystając z modułu `ast`, który, podobnie jak wcześniej `tokenize`, działa również bezpośrednio z linii poleceń.

```
>python -m ast hello.py
Module(
  body=[
    Expr(
      value=Call(
        func=Name(id='print', ctx=Load()),
        args=[
          Constant(value='Hello World')],
        keywords=[]))],
  type_ignores=[])
```

Podobnie jak w przypadku listy tokenów, samo drzewo AST ogranicza się jedynie do kilku węzłów: do korzenia `Module` podczepione jest tylko jedno wyrażenie – typu `Call` (wywołanie funkcji), które z kolei połączone jest jedynie z węzłem zawierającym nazwę (`Name`) funkcji oraz jednym argumentem będącym stałą (`Constant`) o wartości `'Hello World'`.

Warto dodać, że powstało kilka prostych narzędzi umożliwiających wyświetlenie drzewa AST programów napisanych w Pythonie w postaci faktycznego grafu. Efekt działania jednego z nich – AST visualizer autorstwa *quantifiedcode* [1] – znajduje się na Rysunku 1.



Rysunek 1. Drzewo AST programu „Hello World"

Listing 1. Widok heksadecymalny na plik wynikowy

```
>hexdump hello.cpython-39.pyc
00000000: 61 0D 0D 0A 00 00 00 00 - 83 F2 49 61 15 00 00 00 |a         Ia    |
00000010: E3 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 |                  |
00000020: 00 02 00 00 00 40 00 00 - 00 73 0C 00 00 00 65 00 |     @   s    e |
00000030: 64 00 83 01 01 00 64 01 - 53 00 29 02 7A 0B 48 65 |d     d S ) z He|
00000040: 6C 6C 6F 20 57 6F 72 6C - 64 4E 29 01 DA 05 70 72 |llo WorldN)   pr|
00000050: 69 6E 74 A9 00 72 02 00 - 00 00 72 02 00 00 00 FA |int  r    r     |
00000060: 08 68 65 6C 6C 6F 2E 70 - 79 DA 08 3C 6D 6F 64 75 | hello.py  <modu|
00000070: 6C 65 3E 01 00 00 00 F3 - 00 00 00 00             |le>         |
0000007c;
```

W kolejnym kroku na podstawie drzewa AST generowany jest kod bajtowy, który również możemy podejrzeć, tym razem korzystając z modułu `dis`, który wyświetli wszystkie instrukcje w formie tekstowej.

```
>python -m dis hello.py
1   0 LOAD_NAME      0 (print)
    2 LOAD_CONST     0 ('Hello World')
    4 CALL_FUNCTION  1
    6 POP_TOP
    8 LOAD_CONST     1 (None)
   10 RETURN_VALUE
```

Skrótowy opis wszystkich instrukcji można znaleźć w dokumentacji samego modułu `dis` [2], więc na potrzeby tego artykułu ograniczymy się do omówienia jedynie operacji użytych w naszym Hello World:

- **`LOAD_NAME`** ***nazwa*** – umieszcza na stosie wartość zmiennej globalnej o podanej nazwie[2],
- **`LOAD_CONST`** ***stała*** – umieszcza na stosie podaną stałą[3],
- **`CALL_FUNCTION`** ***liczba_parametrów*** – wywołuje zdjętą ze stosu funkcję oraz przekazuje jej podaną liczbę zdjętych ze stosu parametrów,
- **`POP_TOP`** – usuwa jeden element ze szczytu stosu,
- **`RETURN_VALUE`** – wychodzi z funkcji, zwracając element z wierzchu stosu.

Jak można się domyślić z opisu powyższych instrukcji, wzorcowa implementacja Pythona korzysta z maszyny stosowej i – w przeciwieństwie do np. procesorów ARM czy x86 – nie ma rejestrów, a więc kod jest wykonywany poprzez umieszczanie wartości na stosie, a następnie użycie instrukcji, które zdejmują argumenty ze stosu i umieszczają na nim wynik operacji[4].

2. Posłużyliśmy się tu pewnym uproszczeniem, jako że faktycznie w samym kodzie bajtowym w tym miejscu znajdziemy nie ciąg „print", a indeks w tablicy nazw (`co_names`), pod którym można znaleźć ten string.
3. Podobnie jak w przypadku `LOAD_NAME`, w samym kodzie bajtowym znajdziemy jedynie indeks w tablicy stałych (`co_consts`).
4. Pomijając skoki, kod bajtowy Pythona mógłby być w zasadzie porównany do Odwrotnej Notacji Polskiej.

Nasz program sprowadza się do sześciu instrukcji, które kolejno:

- umieszczają na stosie funkcję `print`,
- umieszczają na stosie string „Hello World",
- wywołują zdjętą ze stosu funkcję z jednym argumentem (czyli `print('Hello World')`),
- usuwają ze stosu zwróconą przez `print` wartość `None`,
- umieszczają na stosie wartość `None`,
- pobierają wartość z góry stosu i ją zwracają (czyli istniejące niejawnie w kodzie `return None`), co powoduje zakończenie programu.

OBIEKT CODE

Efektem kompilacji kodu źródłowego nie jest jednak jedynie kod bajtowy, lecz również cała seria metadanych zapisanych w obiekcie klasy `code`. W trzeciej wersji Pythona klasa `code` nie należy do wąskiego grona podstawowych typów (takich jak `str` czy `int`), ale można się do niej odwołać, korzystając z modułu `types`:

```
>>> import types
>>> types.CodeType
<class 'code'>
```

Inspekcja obiektu `code` naszego programu będzie jednak odrobinę bardziej skomplikowana. Otóż najprostszym sposobem, żeby móc wejść w interakcję z tym obiektem, jest wymuszenie skompilowania naszego skryptu do pliku (`python -m compileall hello.py`), a następnie przyjrzenie się otrzymanemu w ten sposób plikowi wynikowemu *__pycache__/hello.cpython-39.pyc* (Listing 1).

Na szczęście nie musimy opierać naszej analizy bezpośrednio o dane binarne. Zamiast tego skorzystamy ze standardowej biblioteki Pythona o nazwie `marshal` w celu zdeserializowania powyższego pliku (uprzednio pomijając nieistotny dla nas 16-bajtowy nagłówek) i wypiszemy informacje o otrzymanym obiekcie typu `code` (`types.CodeType`). W tym celu posłużymy się następującym krótkim programem pomocniczym:

```
import marshal
with open("__pycache__/hello.cpython-39.pyc", "rb") as f:
  marshaled_obj = f.read()[16:]  # Zignoruj 16 bajtów nagłówka.
obj = marshal.loads(marshaled_obj)  # Zwróci obiekt typu code.

print("-=- Code object")
for field in dir(obj):
  if not field.startswith("co_"):
    continue
  print("  %-20s: %s" % (field, getattr(obj, field)))
```