

Architektura i Implementacja interpretera w Pythonie

Aleksander Szarek

III Liceum Ogólnokształcące im. dr Władysława Biegańskiego w Częstochowie

Abstrakt: W tym artykule przedstawiony został proces tworzenia interpretowanego języka programowania w Pythonie na przykładzie autorskiego języka OLang. Głównymi aspektami artykułu jest opis działania interpretowanego języka programowania, preprocesora oraz bibliotek. Opisany poniżej język umożliwia tworzenie nie tylko prostych programów, ale też złożonych algorytmów dzięki bibliotekom takim jak: string, time czy convert oraz możliwości pisania własnych bibliotek w Pythonie.

Słowa kluczowe: język programowania, interpreter, preprocesor, Python

Wprowadzenie

Język programowania [1] to algorytm ułatwiający pisanie innych algorytmów. Wyróżniamy języki kompilowane, np. C, C++, Java oraz interpretowane, np. Python i JavaScript. Ten artykuł skupia się na tej drugiej grupie, gdyż są one łatwiejsze do stworzenia. Przykładowy język programowania, dalej nazywany "język" lub "OLang" składa się z trzech modułów, które umożliwiają jego pracę: preprocesor, interpreter oraz biblioteki. Preprocesor [2] jest najważniejszym elementem języka, przygotowuje on kod do dalszej interpretacji, oraz tworzy obiekty [3], które mogą być potem wykorzystywane przez odpowiednie instrukcje. OLang wykorzystuje preprocesor do tworzenia obiektów takich jak funkcje oraz pętle, tworzenia instrukcji umożliwiających wykonanie ich oraz dołączania bibliotek za pomocą instrukcji preprocesora "\$use". Interpreter, natomiast, to część języka, która umożliwia wykonywanie kodu na bieżąco, czyli linijka po linijce. W tym przypadku interpreter usuwa komentarze oznaczone poprzez znak '//', dzieli kod na części, gdzie pierwsza część zawsze odpowiada poleceniu z biblioteki; oraz wykonuje owe polecenie poprzez przekazanie odpowiednich argumentów do odpowiedniej biblioteki. Kod interpretera przedstawiono na rysunku 1.

```
def run_object(line: str):
    if "//" in line:
        line = line[:line.index("//")]
    try:
        return eval(line.strip())
    except Exception as e:
        ...
    try:
        return float(line.strip())
    except ValueError:
        ...
    if line.strip() == "":
        return
    for lib in librariesVars:
        try:
            insts = lib.instructions
            instsVars = lib.variables
            ins = line.strip().split('(')[0].split()[0]
        except Exception as e:
            throw_error(f"Failed to access library specification for {libraries[librariesVars.index(lib)]}", True, line)
            return
        if ins in insts:
            try:
                return instsVars[insts.index(ins)](line.strip(), run_object, throw_error)
            except Exception as e:
                throw_error(f"Unknown error while executing {ins}", True, line)
            return
    throw_error(f"Could not find instruction in current scope!", True, line)
```

Rysunek 1. Interpreter

Wszystkie biblioteki używane przez interpreter zapisane są w folderze "libraries" w kodzie języka. Tam też można dopisywać nowe biblioteki, które potem można dodawać do programu za pomocą instrukcji preprocesora "\$use", bądź poprzez konfigurację preprocesora - dodając wpisy to listy

“defaultLibraries” (pol. domyślne biblioteki). Domyślny kod pliku konfiguracyjnego preprocesora przedstawiono na rysunku 2.

```
{-} config.json > ...
1  {
2      "interpreterVer": "0.1.0",
3      "preprocessorDebug": false,
4      "returnOnFatal": true,
5      "generalLibraryLocation": "./libraries",
6      "includeLibraries": ["convert", "mainlib", "string", "arrays", "varlib"]
7  }
```

Rysunek 2. Plik konfiguracyjny preprocesora

Materiały i metody

W celu stworzenia języka wykorzystano język programowania Python w wersji 3.11 oraz wiele bibliotek domyślnie dołączonych do instalacji Pythona, tj:

- **json**: Biblioteka wykorzystana do odczytu informacji z pliku konfiguracyjnego preprocesora (config.json).
- **os**: Biblioteka wykorzystana do sprawdzania, czy biblioteka oczekiwana przez preprocesor istnieje.
- **importlib.util**: Biblioteka wykorzystana do dołączania bibliotek oczekiwanych przez preprocesor do programu.
- **colorama**: Biblioteka wykorzystana do zmiany stylu oraz koloru tekstu, np. w przypadku błędów.
- **re**: Biblioteka wykorzystana w niektórych bibliotekach z poleceniami pozwalającymi na dołączanie argumentów. Dzięki niej możliwa jest analiza argumentów polecenia, szczególnie jeśli jest więcej niż jeden argument.
- **datetime**: Biblioteka umożliwiająca stworzenie poleceń bazujących na czasie, jak. now(), czy dconv().
- **time**: Biblioteka umożliwiająca wykonywanie operacji związanych z czasem jak wait().
- **random**: Biblioteka umożliwiająca działanie funkcji random() z biblioteki time.

Poza bibliotekami wykorzystano również klasy do tworzenia poszczególnych obiektów, np. funkcji, pętli, zmiennych, czy list. Wykorzystanie klas umożliwiło odgórne przypisanie operacji dostępnych dla danego obiektu. Przykładem może być klasa zmiennych, której kod przedstawiono na rysunku 3.

```
class Variable:
    def __init__(self, name: str, type: int, value: object):
        self.name = name
        self.type = type
        self.type_name = "int" if type == 1 else "float" if type == 2 else "str" if type == 3 else "bool" if type == 4 else "object"
        self.value = value
    def get_value(self) -> object:
        return self.value
    def set_value(self, value, run_object, throw_error) -> bool:
        evaluated = eval_value(value, run_object, throw_error)
        if self.type == 1: # int...

        elif self.type == 2: # float...

        elif self.type == 3: # str...

        elif self.type == 4: # bool...

        return False
```

Rysunek 3. Klasa zmiennej

Warto również wyróżnić system działania bibliotek na przykładzie biblioteki “mainlib” zawierającej instrukcje: write(), writeline() oraz read(). Każda domyślnie dołączona biblioteka zawiera komentarz pokrótce opisujący wszystkie instrukcje w języku angielskim, funkcje pomocnicze (w tym przypadku eval_value() oraz split_args()), klasę funkcji, gdzie każda funkcja jest innym poleceniem, oraz dwie wymagane listy: instructions typu string oraz variables typu obiektowego. Lista instructions posiada tekstowy opis wszystkich instrukcji, co umożliwia wykrycie ich przez interpreter. Lista variables,

natomiast, zawiera wskaźniki do funkcji, które odpowiadają za owe polecenia. Kod biblioteki "mainlib" został przedstawiony na rysunku 4.

```
libraries > mainlib.py > ...
1  ...
2  This module provides main functionality for printing strings to the screen or getting input values.
3  Instructions:
4  - write(arg1, arg2, ...): Prints the given arguments to the console.
5  - writeline(arg1, arg2, ...): Prints the given arguments to the console followed by a newline.
6  - read(): Reads input from the user and returns it as a string.
7  ...
8  import re
9  > def eval_value(value: str, run_object, throw_error) -> object: ...
77 > def split_args(content: str) -> list[str]: ...
109
110 class MainLib:
111     def write(self, arg: str, run_object, throw_error):
112         content = arg.strip()[arg.find('(') + 1:arg.rfind(')')].replace("\\n", "\n").replace("\\t", "\t")
113         items = split_args(content)
114
115         for item in items:
116             item = item.strip()
117             if item.startswith('') and item.endswith(''):
118                 print(item[1:-1], end='')
119             elif item.startswith('') and item.endswith(''):
120                 print(item[1:-1], end='')
121             else:
122                 try:
123                     print(float(item), end='')
124                 except ValueError:
125                     print(eval_value(item, run_object, throw_error), end='')
126     def writeline(self, arg: str, run_object, throw_error):
127         self.write(arg, run_object, throw_error)
128         print()
129     def read(self, arg: str, run_object, throw_error) -> str:
130         if arg.strip() != "read()":
131             throw_error("Invalid read function syntax. Expected 'read()'.", True, arg)
132             return ""
133         try:
134             return input()
135         except EOFError:
136             throw_error("Input was interrupted or EOF reached.", False, arg)
137             return ""
138
139 l = MainLib()
140 instructions: list[str] = ["write", "writeline", "read"]
141 variables: list[object] = [l.write, l.writeline, l.read]
```

Rysunek 4. Struktura biblioteki – biblioteka mainlib

Przeglądając kod wyżej opisanej biblioteki można zauważyć funkcje split_args() oraz eval_value() będące funkcjami pomocniczymi. Owe funkcje umożliwiają działanie na obiektach, działaniach matematycznych oraz wartościach. Funkcja split_args() (stworzona na podstawie artykułu "Parameter estimation for text analysis.") [4] odpowiada za rozdzielanie argumentów polecenia. Rozdzielana ona argumenty po znaku ',' ignorując te znaki wchodzące w skład innych argumentów, np.

"write("Wartość zmiennej g, której typ to ", type(g), " wynosi: ", get(g));" - funkcja split_args() zignoruje , zawarty w wartości tekstowej "Wartość zmiennej g, której typ to ", gdyż jest on częścią obiektu typu string. Drugim przykładem niech będzie "declare(a, int, random(3, 10));", gdzie funkcja split_args() zignoruje znak ',' w poleceniu random(). Kod funkcji split_args() przedstawiono na rysunku 5.

```

def split_args(content: str) -> list[str]:
    args = []
    current = ""
    depth = 0
    in_string = False
    string_char = ""

    for c in content:
        if in_string:
            current += c
            if c == string_char:
                in_string = False
                continue

        if c in "\"'":
            in_string = True
            string_char = c
            current += c
        elif c == "," and depth == 0:
            args.append(current.strip())
            current = ""
        else:
            if c == "(":
                depth += 1
            elif c == ")":
                depth -= 1
            current += c

    if current:
        args.append(current.strip())

    return args

```

Rysunek 5. Funkcja `split_args()`

Z drugiej strony, funkcja `eval_value()` odpowiada za wyliczenie pojedynczego argumentu. Jej zadaniem jest wyliczenie każdego elementu działania (elementem działania jest każdy obiekt oddzielony od innego operatorami arytmetycznymi) poprzez przypisanie mu odpowiedniego typu (liczbowego, bądź tekstowego) lub przekazanie argumentu do interpretera jako osobnej instrukcji i zastąpienie owego wartością zwróconą przez interpreter. Kod funkcji `eval_value()` przedstawiono na rysunku 6.

```

9 def eval_value(value: str, run_object, throw_error) -> object:
10     value = str(value)
11     raw_value = value.replace(" ", "").replace(",", "$")
12     values = raw_value.replace("+", ",").replace("-", ",").replace("**", ",").replace("/", ",").replace("^", ",").split(",")
13
14     vals = []
15     valchars = []
16     bracketstarts = []
17     bracketends = []
18     v = []
19     for i in value:
20         if i in '+-*/%^':
21             valchars.append(i)
22     if value.count("(") != value.count(")"):
23         throw_error("Mismatched brackets in the expression.", True, value)
24         return False
25     if len(values) > 1:
26         for i, val in enumerate(values):
27             if val.startswith("("):
28                 bracketstarts.append(i)
29                 val = val[1:]
30             if val.endswith(")"):
31                 val = val[:-1]
32                 bracketends.append(i+1)
33             try:
34                 val = float(val)
35             except ValueError:
36                 if re.match(r"^[a-zA-Z_]\w*(\.(.*)$)", val.replace("$", ",")):
37                     val = run_object(val.replace("$", ","))
38                 else:
39                     val = run_object(val.replace("$", ","))
40             vals.append(val)
41
42     for i, val in enumerate(vals):
43         while i in bracketstarts:
44             v.append('(')
45             bracketstarts.remove(i)
46         while i in bracketends:
47             v.append(')')
48             bracketends.remove(i)
49         v.append(str(val))
50         if i < len(vals) - 1:
51             v.append(str(valchars[i]))
52     if bracketends:
53         v.append(')')
54     else:
55         try:
56             return float(value)
57         except ValueError:
58             return run_object(value)
59     evaluated = "".join(v).replace('^', '**').replace('[', '(').replace(']', ')')
60     try:
61         return eval(evaluated)
62     except Exception as e:
63         throw_error(f"Evaluation failed for '{evaluated}'", True, evaluated)

```

Rysunek 6. Funkcja eval_value()

Dzięki zastosowaniu bibliotek, klas oraz specjalistycznych funkcji, OLang jest w stanie bezbłędnie rozumieć kod, który analizuje i wykonywać go w czasie rzeczywistym.

Rezultaty

Gotowy program języka OLang możemy uruchomić na dowolnej maszynie z zainstalowanym Pythonem 3.11 lub wyższym i uruchomić dowolny kod, który napisaliśmy w owym języku. Przykładem niech będzie program wyliczający kolejne liczby ciągu Fibonacciego [5], którego kod wraz z wynikiem przedstawiono na rysunku 7.

```
test.olang
1  $use time;
2  declare(a, int, 1);
3  declare(b, int, 1);
4  declare(tmp, int);
5  for(declare(i, int, 0), get(i) < 7, set(i, get(i) + 1)){
6      write(get(a), "\n");
7      set(tmp, get(a));
8      set(a, get(b) + get(a));
9      set(b, get(tmp));
10     wait 1;
11 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS ESP-IDF

Preprocessing complete. Objects and main code are ready for execution.

2.0
3.0
5.0
13.0
21.0
34.0
55.0

Interpreter finished code execution with exit code 0

Rysunek 7. Ciąg Fibonacciego

Kolejnym przykładem może być prosty program z wykorzystaniem biblioteki "time" a dokładnie instrukcji now() umożliwiającej pozyskanie aktualnej daty i godziny w połączeniu z instrukcją dconv() umożliwiającą pozyskanie poszczególnych elementów zarówno daty jak i godziny. Kod programu wraz z rezultatem przedstawiono na rysunku 8.

```
test.olang
1  $use time;
2  declare(a, str);
3  set(a, now());
4  write("Today's date is: ", dconv(d, get(a)), ".", dconv(m, get(a)), ".", dconv(y, get(a)));
5
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS ESP-IDF

x64\bundled\libs\debuggy\launcher' '50639' '--' 'F:\Projects\Python\2025 Projects\OLang2\run.py'

=== LIBRARIES ===

[ID 0] convert
[ID 1] mainlib
[ID 2] string
[ID 3] arrays
[ID 4] varlib
[ID 5] time

=== OBJECTS ===

=== MAIN CODE ===
declare(a, str)
set(a, now())
write("Today's date is: ", dconv(d, get(a)), ".", dconv(m, get(a)), ".", dconv(y, get(a)))

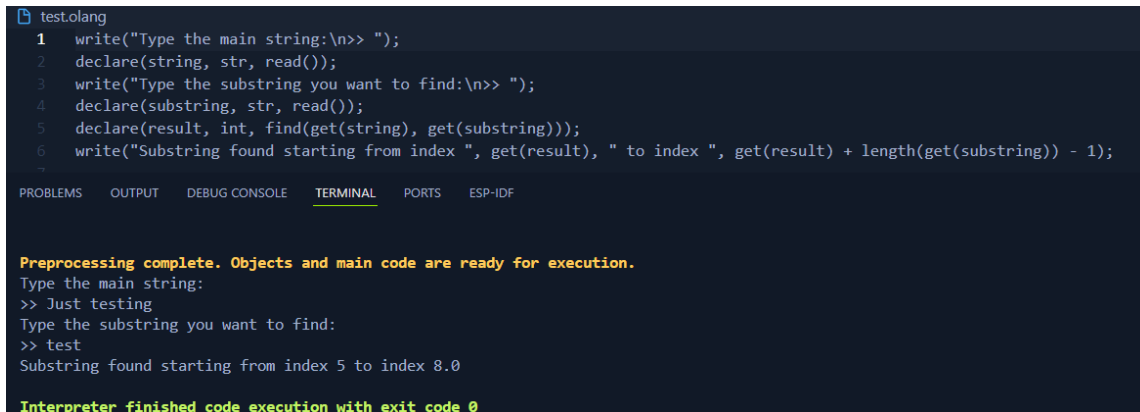
Preprocessing complete. Objects and main code are ready for execution.
Today's date is: 31.5.2025

Interpreter finished code execution with exit code 0

Rysunek 8. Dzisiejsza data

Ostatnim przykładem programu napisanym w OLang jest program znajdujący podciąg w ciągu znaków - w tym przypadku są to dwie zmienne typu tekstowego. Dzięki wykorzystaniu instrukcji

find() z biblioteki string wraz z funkcją length() pochodzącej z tej samej biblioteki, program z łatwością znajduje poszukiwany podciąg ("test"). Kod programu wraz z rezultatem przedstawiono na rysunku 9.



```
testolang
1  write("Type the main string:\n>> ");
2  declare(string, str, read());
3  write("Type the substring you want to find:\n>> ");
4  declare(substring, str, read());
5  declare(result, int, find(get(string), get(substring)));
6  write("Substring found starting from index ", get(result), " to index ", get(result) + length(get(substring)) - 1);

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  ESP-IDF

Preprocessing complete. Objects and main code are ready for execution.
Type the main string:
>> Just testing
Type the substring you want to find:
>> test
Substring found starting from index 5 to index 8.0

Interpreter finished code execution with exit code 0
```

Rysunek 8. Wyszukiwanie pociągu w ciągu znaków

Podsumowanie

Stworzono język programowania w Pythonie o nazwie OLang w celu zaprezentowania etapów tworzenia takowego oraz metodyki działania owego. OLang składa się z trzech głównych komponentów, tj. preprocesor, interpreter oraz z bibliotek, które można samodzielnie programować w Pythonie rozszerzając możliwości języka o nowe instrukcje. Artykuł ma na celu przybliżenie również składni języka OLang oraz przedstawienie kilka przykładów programów, które można w nim napisać. Należy tutaj jednak podkreślić, że nie są to jedyne możliwości, a tylko przykłady oparte na dostępnych domyślnie bibliotekach.

Literatura

1. Kenneth E. Iverson. 1962. A programming language. In Proceedings of the May 1-3, 1962, spring joint computer conference (AIEE-IRE '62 (Spring)). Association for Computing Machinery, New York, NY, USA, 345–351.
2. STALLMAN, Richard M.; WEINBERG, Zachary. The C preprocessor. Free Software Foundation, 1987, 16.
3. MACLENNAN, Bruce J. Values and objects in programming languages. ACM SIGPLAN Notices, 1982, 17.12: 70-79.
4. HEINRICH, Gregor. Parameter estimation for text analysis. Darmstadt, Germany: Technical report, 2005.
5. YAYENIE, Omer. A note on generalized Fibonacci sequences. Applied Mathematics and computation, 2011, 217.12: 5603-5611.

Architecture and implementation of the interpreter in Python

Abstract: In this article is presented the process of creating an interpreted programming language in Python based on an example of my own programming language called OLang. The main aspects of the article are: the description of how the interpreted programming language, preprocessor and libraries work. Language described here allows the creation of not only simple programs, but also of the more complicated ones thanks to libraries like: string, time or convert and the ability to write other libraries in Python.

Keywords: programming language, interpreter, preprocessor, Python