TKOM - Python++

Dokumentacja

Michał Sobiech 318722

05.06.2024

1 Temat

- Temat nr 8: Język z wbudowanym typem słownika, którego zawartość można sortować. Możliwe są podstawowe operacje na słowniku (dodawanie, usuwanie, wyszukiwanie elementów wg klucza, sprawdzanie, czy dany klucz znajduje się w słowniku itd.), iterowanie po elementach oraz wykonywanie na słowniku zapytań w stylu LINQ. Metoda sortująca elementy słownika przyjmuje jako argument funkcję określającą kolejność elementów.
- Typowanie dynamiczne
- Język silnie typowany
- Język realizacji projektu: Python

2 Sposób uruchamiania

```
$ python++ main.ppp
```

3 Zakładana funkcjonalność

- Obsługiwane typy danych
  - Liczby
    - Int
    - Float
  - Wartość logiczna - bool
  - Ciąg znaków - string
  - Słownik - dict
    - Dodawanie elementów

        * Odejmowanie elementów
        * Wyszukiwanie elementów po kluczu
        * Wyszukiwanie elementów po indeksie
        * Sprawdzanie obecności klucza w słowniku
        * Iterowanie po elementach
        * Sortowanie słownika przy użyciu funkcji określającej kolejność elementów
        * Operacje w stylu LINQ
    - Lista - list
        * Dodawanie i odejmowanie elementów
        * Wyszukiwanie elementów po indeksie
    - Para klucz-wartość - pair
        * dostęp do klucza
        * dostęp do wartości
- Obsługa komentarzy
- Możliwość tworzenia zmiennych, przypisywania im wartości i odczytywania ich
- Wszystko jest mutowalne
- Instrukcje warunkowe
    - if
    - elif
    - else
- Instrukcje pętli
    - for
    - while
- Możliwość definiowania i wywoływania własnych funkcji
- Przekazywanie argumentów do funkcji przez wartość
- Zmienne dostępne jedynie w swoim zakresie (brak rozwiązania w stylu pythonowego ”global”)
- Rekursywne wywołania funkcji
- Obsługa błędów przez interpreter
- Brak niejawnej konwersji
- Przeciążanie funkcji niedozwolone
- Przykrywanie zmiennych dozwolone

4 Struktura

4.1 Komponenty

- Źródło kodu
  - Leniwie czyta kolejne znaki kodu źródłowego
  - Klasa
    - CodeSource(src_file_path: str) - ładuje znaki ze wskazanego pliku. Implementuje funkcje iteratora, więc znaki pobiera się wywołaniem next().
- Analizator leksykalny
  - Przetwarza znaki na tokeny
  - Leniwie podbiera znaki od źródła kodu
  - Klasa
    - Lexer(code_source: CodeSource, error_handler: ErrorHandler)
      - get_next_token() -> Optional[Token] - zwraca kolejny token. Jeśli lexer już wyczerpał znaki, to metoda zwraca None.
      - get_current_token() -> Token - zwraca ostatni ułożony token
      - get_current_token_pos() -> tuple[int, int] - zwraca pozycję obecnego tokenu w postaci krotki zawierającej kolejno rząd i kolumnę liczone od 0.
- Analizator składniowy
  - Generuje drzewo składniowe na podstawie tokenów
  - Klasa
    - Parser(lexer: Lexer, error_handler: ErrorHandler)
      - get_ast() -> Code - zwraca drzewo składniowe
- Interpreter
  - Wykonuje instrukcje z drzewa składniowego
  - Klasa
    - Interpreter(ast: Code, output_stream: IOBase, error_handler: ErrorHandler)
      - interpret() -> None - wykonuje instrukcje z AST.

4.2 Gramatyka

4.2.1 Symbole obsługiwane przez parser

```
code             =  { fun_definition | statement };

fun_definition   = "def", id, "(", fun_params, ")", block;
```

```
fun_params       = [ id { ",", id } ];

statement        = assign_or_expr_or_incrdecr, ";"
                 | return, ";"
                 | cond_statement,
                 | for_loop
                 | while_loop;

id_or_fun_call   = id, [ "(", [ fun_call_args ], ")" ];

assign_or_expr_or_incrdecr = expression, [ '=', expression
                           | ( "++" | "--" ) ];

cond_statement   = if_statement, { elif_statement }, [ else_statement ];
if_statement     = "if", expression, block;
elif_statement   = "elif", expression, block;
else_statement   = "else", block;
block            = "{", { statement }, "}";

fun_call_args    = expression, { ",", expression };

for_loop         = "for", id, "in", expression, block;

while_loop       = "while", expression, block;

return           = "return", [ expression ];

expression        = and_term, { "or", and_term };
and_term          = not_term, { "and", not_term };
not_term          = [ "not" ], comp_term;
comp_term         = add_or_sub_term, [ comp_operator, add_or_sub_term ];
add_or_sub_term   = mul_or_div_term, { ("+" | "-"), mul_or_div_term };
mul_or_div_term   = unary_minus_term, { ("*" | "/"), unary_minus_term };
unary_minus_term  = [ "-" ], dot_term;
dot_term          = term, { ".", term };
nested_expression = "(", expression, ")";
term              = literal
                  | list
                  | dict_or_pair
                  | id_or_fun_call
                  | nested_expr
                  | linq_query;


linq_query       = "from", id, "in", expression, "where", expression,
                 [ "orderby", expression ],
                 "select", expression, { ",", expression };
```

```
dict_or_pair     = "{", ( "}"
                 | (expression, (",", expression
                 | ":", expression, [ dict_elems ] ), "}" ) );

dict_elems       = dict_element, { ",", dict_element };
dict_element     = expression, ":", expression;

list             = "[", [ list_elems ], "]";
list_elems       = expression, { ",", expression };
```

4.2.2 Symbole obsługiwane przez lekser

```
comp_operator    = "<" | ">" | "<=" | ">=" | "==";

variable         = id, { ".", id };

id               = ( letter | "_" ) { alphanumeric | "_" };

literal          = string | int | float | bool | none;

none             = "None";

bool             = "True" | "False";

string           = "'", { ascii_char }, "'";

float            = int, ".", positive_int;
int              = "0" | nonzero_digit, { digit };

alphanumeric     = letter | digit;
letter           = wszystkie małe i duże litery
digit            = "0" | nonzero_digit;
nonzero_digit    = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
ascii_char       = dowolny znak ascii
```

4.3 Priorytety operatorów

Asocjacyjność operatorów. Im wyżej w tabeli, tym wyższy priorytet

Operator	Asocjacyjność
() .	-
not - (unarny)	-
* /	Lewostronna
+ - (odejmowanie)	Lewostronna
> >= < <= ==	-
and	Lewostronna
or	Lewostronna

Operatory ”++” i ”–” są poza hierarchią.

4.4 Rodzaje tokenów

Tokeny to obiekty posiadające 3 pola:

- Rodzaj
- Wartość (pusta dla niektórych rodzajów, np. Plus)
- Pozycja w kodzie

Pogrupowane rodzaje:

- Literały wbudowanych typów
  - IntLiteral
  - FloatLiteral
  - StringLiteral
- Operatory do obliczeń
  - Plus
  - PlusPlus
  - Minus
  - MinusMinus

    - Multiply
    - Divide
- Operatory do porównywania
    - Less
    - LessEquals
    - Greater
    - GreaterEquals
    - EqualsEquals
- Operatory logiczne
    - Or
    - And
    - Not
- Słowa-klucze
    - Def
    - Return
    - For
    - In
    - If
    - Elif
    - Else
    - From
    - Where
    - Select
    - OrderBy
    - True
    - False
    - None
- Nawiasy
    - ParenthesesLeft
    - ParenthesesRight
    - BracketsLeft
    - BracketsRight
    - BracesLeft
    - BracesRight

- Pozostałe
  - Identifier
  - Comment
  - Comma
  - Colon
  - Dot
  - Whitespace

5 Testowanie

Będą 2 rodzaje testów:

- Jednostkowe, przeprowadzane dla każdego modułu oddzielnie
  - Testy kodu źródła
  - Testy leksera
  - Testy parsera
  - Testy interpretera
- Integracyjne

6 Przykładowy kod

6.1 Definiowanie zmiennych

```
bool_var = True;
int_var = 1;
float var = 0.1;
str_var = 'sample text';
dict_var = {'a': 1, 'b': 2};
list_var = [1, 2, 3];
pair_var = {'a', 1};
```

6.2 Definiowanie i wywoływanie funkcji

```
def increment(number) {
    return number + 1;
}

a = increment(1);
```

6.3 Globalne funkcje wbudowane

```
print(1);        # prints 1
bool(1);         # True
int('123') ;     # 123
float('123');    # 123.0
string(123);     # '123'
```

6.4 Komentarze

```
# This is a comment
```

6.5 Operatory

```
a + b;
a - b;
a * b;
a / b;
(a);
a.b();
not a;
-a;
a > b;
a >= b;
a < b;
a <= b;
a == b;
a and b;
a or b;
```

6.6 Operacje na stringach

```
a = 'sample';

a + 'text'      # 'sampletext'
```

6.7 Operacje na listach

```
list = [1, 2, 3];

list.length();       # 3

for e in list {
    print(e);
}
# 1
```

```
# 2
# 3

list.append(4);      # list = [1, 2, 3, 4]
list.pop();          # list = [1, 2, 3]
list.at(0);          # 1
```

6.8 Operacje na słownikach

```
dict = {
    'a': 1,
    'b': 2,
    'c': 3
};
dict.append({'d', 4});  # dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

dict.by_key('c');       # 3
dict.by_index(1);       # {'a', 1}
dict.pop();             # dict = {'a': 1, 'b': 2, 'c': 3}
dict.length()           # 3
dist.has_key('a')       # True

# Sortowanie
dict = {'d': 4, 'a': 1, 'c': 3, 'b': 2};
def sort_func(tuple_a, tuple_b) {
    return tuple_a.value() > tuple_b.value();
}
dict.sort(sort_func);   # dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

6.9 Operacje na krotkach

```
a = {'a', 1};

a.key();     # 'a'
a.value()    # 1
```

6.10 Operacje w stylu LINQ

```
dict = {
    'd': 4,
    'a': 1,
    'b': 2,
    'c': 3
};

def sorting_fun(a, b) {
```

```
    return a.value() < b.value();
}

result = from tuple in dict
         where tuple.value() > 2
         orderby sorting_fun
         select tuple.key(), tuple.value();
# [['c', 3], ['d', 4]]
```

7 Obsługa błędów

7.1 Błędy leksera - TokenError

Format błędu:

```
[Rodzaj błędu], line [numer linii], column [numer kolumny]: [Opis błędu]
```

- Kody
  1. STR_EOF_INSIDE - znak końca pliku pojawił się w trakcie definicji stringa
  2. CHAR_AFTER_STR_END - Znak pojawił się zaraz po definicji stringa
  3. FLOAT_INVALID_CHAR - Niewłaściwy znak pojawił się w środku definicji floata
  4. INT_INVALID_CHAR - Niewłaściwy znak pojawił się w trakcie definicji liczby całkowitej
  5. INT_DIGIT_AFTER_ZERO - Po zerze pojawiła się liczba
- Przykłady
  - `1a = 1;`

    ```
    Lexer error at line 1, column 2: Invalid char inside a number (int/float)
    definition.
    ```

7.2 Błędy parsera - SyntaxError

- Format błędu:

  ```
  [Rodzaj błędu], line [numer linii], column [numer kolumny]: [Opis błędu]
  ```

- Kody
  1. NEITHER_FUN_DEF_NOR_STATEMENT
  2. FUN_INVALID_NAME

3. FUN_NO_LEFT_PARENTHESES
4. FUN_NO_RIGHT_PARENTHESES
5. FUN_DEF_INVALID_PARAM
6. FUN_DEF_INVALID_BODY
7. FUN_CALL_INVALID_ARG
8. BLOCK_NO_RIGHT_BRACES
9. IF_INVALID_CONDITION
10. IF_INVALID_BODY
11. ELIF_INVALID_CONDITION
12. ELIF_INVALID_BODY
13. ELSE_INVALID_BODY
14. ASSIGN_INVALID_RIGHT_SIDE
15. NO_SEMICOLON
16. OR_TERM_INVALID_RIGHT_SIDE
17. AND_TERM_INVALID_RIGHT_SIDE
18. NOT_TERM_INVALID_RIGHT_SIDE
19. COMP_TERM_INVALID_RIGHT_SIDE
20. ADD_OR_SUB_TERM_INVALID_RIGHT_SIDE
21. MUL_OR_DIV_TERM_INVALID_RIGHT_SIDE
22. MINUS_TERM_INVALID_RIGHT_SIDE
23. DOT_TERM_INVALID_TERM
24. LINQ_INVALID_ITERATOR_VAR
25. LINQ_NO_IN
26. LINQ_INVALID_SEQUENCE
27. LINQ_NO_WHERE
28. LINQ_INVALID_CONDITION
29. LINQ_NO_ORDERBY_FUN
30. LINQ_NO_SELECT
31. LINQ_INVALID_SELECTED_VALUES
32. LINQ_SELECTED_VALUES_END_WITH_COMMA
33. FOR_LOOP_INVALID_ITERATOR_VAR
34. FOR_LOOP_NO_IN
35. FOR_LOOP_INVALID_SEQUENCE
36. FOR_LOOP_INVALID_BODY
37. WHILE_LOOP_INVALID_CONDITION
38. WHILE_LOOP_INVALID_BODY

39. DICT_NO_RIGHT_BRACES
40. DICT_ELEM_INVALID_VALUE
41. DICT_ELEM_NO_COLON
42. INVALID_DICT_ELEM
43. PAIR_INVALID_VALUE
44. PAIR_NO_RIGHT_BRACES
45. LIST_INVALID_ELEMENT
46. LIST_NO_RIGHT_BRACKETS
47. NESTED_EXPR_INVALID_EXPR
48. NESTED_EXPR_NO_RIGHT_PARENTHESES

- Przykłady
  - `print(a)`

    ```
    Syntax error at line 1, column 9: Expected a semicolon.
    ```

  - `def a() {};`

    ```
    Syntax error at line 1, column 11: Neither a function definition nor a statement.
    ```

7.3 Błędy interpretera - InterpreterError

- Format błędu:

```
[Rodzaj błędu], line [numer linii], column [numer kolumny]: [Opis błędu]
Context stack: [...]
```

- Kody

1. STACK_OVERFLOW
2. FUNCTION_ALREADY_EXISTS
3. NO_SUCH_FUNCTION
4. INVALID_ARG_COUNT
5. INVALID_TYPE
6. INVALID_VALUE
7. NAME_TAKEN_BY_FUNCTION
8. NAME_TAKEN_BY_VAR
9. NO_SUCH_OBJECT
10. NO_SUCH_RVALUE

11. ARG_INVALID_TYPE
12. DIVISION_BY_ZERO
13. DICT_KEY_EXISTS
14. NO_SUCH_KEY_IN_DICT
15. DICT_IS_EMPTY
16. LIST_IS_EMPTY
17. INVALID_INDEX
18. UNITERABLE_OBJECT
19. NO_SUCH_VARIABLE

- Przykłady
  - ```
    print(a);
    ```

    ```
    InterpreterError at line 1, column 7: This variable/function does not exist.
    Call stack:
    print at line 1
    ```

  - ```
    int('aaa');
    ```

    ```
    InterpreterError at line 1, column 5: Invalid value(s).
    Call stack:
    int at line 1
    ```

  - ```
    for element in 1 {
        print(e);
    }
    ```

    ```
    InterpreterError at line 1, column 16: Cannot iterate over this object.
    Call stack:
    (Empty)
    ```
    ```