**Модуль 10. Практическая работа  (4 ак. ч.)**

**Вариант 1 Создание простого интерпретатора арифметических выражений**

Выражения содержат только целые числа, операторы и круглые скобки.

Допустимые операторы в выражении:

* +: сложение
* -: вычитание
* \*: умножение
* /: деление
* ^: возведение в степень

Операторы не имеют приоритетов, приоритеты задаются круглыми скобками. Для удобства единое выражение тоже заключено в скобки.

Таким образом, результатом выражения (1+((2+3)\*(4\*5))) будет 101, а выражения (2+((2\*3)/(4^5))) будет 2.

**Основное задание**

**import** **abc**

**class** **InterpreterAbstract**(abc.ABC):

*'''Интерпретатор кода'''*

**def** \_\_init\_\_(self, code):

*'''Принимает код'''*

self.code = code

**def** execute(self):

*'''Запускает механизм исполнения кода*

*Возвращает результат исполнения кода'''*

**return** self.\_parse()

@abc.abstractmethod

**def** \_parse(self):

*'''Осуществляет парсинг кода.*

*Вызывает \_evaluate для исполнения выражений*

*Возвращает результат исполнения кода в excecute'''*

**pass**

@abc.abstractmethod

**def** \_evaluate(self, code):

*'''Осуществляет вычисление выражения*

*Возвращает результат выражения в \_parse'''*

**pass**

Используя абстрактный класс InterpreterAbstract создайте и опишите класс Interpreter

* конструктор класса должен принимать код в виде строки.
* метод \_parse должен разобрать строку кода посимвольно и передать её на исполнение методу \_evaluate
* метод \_evaluate должен вычислить выражение и вернуть его результат.

interpreter = Interpreter('(1+((2+3)\*(4\*5)))')

**print**(interpreter.execute()) *# 101*

interpreter = Interpreter('(2+((2\*3)/(4^5)))')

**print**(interpreter.execute()) *# 2*

**Подсказка**

* вам надо создать два стека: для чисел и операторов
* продвигаясь по коду:
  + при нахождении числа добавляйте его в стек для чисел
  + при нахождении опрератора добавляйте его в стек для операторов
  + при нахождении символа ):
    - извлеките последний оператор из стека операторов
    - извлеките два последних числа из стека чисел
    - произведите над числами соответствующую операцию
    - добавьте результат операции в стек для чисел
  + когда символы в коде закончатся, извлеките значение из стека чисел - это и будет результат всего выражения

**Дополнительно 1**

До сих пор мы игнорировали пробелы в коде, однако они могут присутствовать.

Например: (1 + ( ( 2 + 3 )\*( 4 \*5) ) )

Сделайте так, чтобы код работал при любом количестве пробелов.

Заодно избавьтесь от первой и последней круглых скобок, что бы конечный пользователь о них не думал:

interpreter = Interpreter('1 + ( ( 2 + 3 ) \* ( 4 \* 5 ) )')

**Дополнительно 2**

Измените конструктор, чтобы он мог принимать не только строку кода, но и файл с кодом.

В файле каждое выражение должно располагаться на отдельной строке.

В этом случае результатом выполнения кода должен быть список с результатами каждого выражения.

Например, если содержимое файла выглядит так:

1 + ( ( 2 + 3 ) \* ( 4 \* 5 ) )

2 + ( ( 2 \* 3 ) / ( 4 ^ 5 ) )

то

interpreter = Interpreter(file='code.txt')

**print**(interpreter.execute()) *# [101, 2]*

**Дополнительно 3**

До этого момента, мы считали, что все скобки открыты, закрыты и вложены друг в друга правильно, однако это может быть не так.

Добавьте метод \_validate, который будет перед исполнением кода проверять его сбалансированность скобок.

В случае нарушения баланса скобок, метод должен выбросить исключение об ошибке и завершить работу нашего интерпретатора.

Решение

1 этап

import abc  
from stack import Stack  
  
  
class InterpreterAbstract(abc.ABC):  
 *'''Интерпретатор кода'''* def \_\_init\_\_(self, code):  
 *'''Принимает код'''* self.code = code  
  
 def execute(self):  
 *'''Запускает механизм исполнения кода  
 Возвращает результат исполнения кода'''* return self.\_parse()  
  
 @abc.abstractmethod  
 def \_parse(self):  
 *'''Осуществляет парсинг кода.  
 Вызывает \_evaluate для исполнения выражений  
 Возвращает результат исполнения кода в excecute'''* pass  
  
 @abc.abstractmethod  
 def \_evaluate(self, code):  
 *'''Осуществляет вычисление выражения  
 Возвращает результат выражения в \_parse'''* pass  
  
  
class Interpreter(InterpreterAbstract):  
 def \_\_init\_\_(self, code):  
 *'''Принимает код'''* self.\_\_code = code  
 self.\_\_ops = Stack()  
 self.\_\_vals = Stack()  
  
 def execute(self):  
 return self.\_parse()  
  
 def \_parse(self):  
 return self.\_evaluate(self.\_\_code)  
  
 def \_evaluate(self, code):  
 operators = ['+', '-', '\*', '/', '^']  
 for char in code:  
 if char == '(':  
 continue  
 elif char in operators:  
 self.\_\_ops.push(char)  
 elif char == ')':  
 op = self.\_\_ops.pop()  
 v = int(self.\_\_vals.pop())  
 if op == '+':  
 v = int(self.\_\_vals.pop()) + v  
 elif op == '-':  
 v = int(self.\_\_vals.pop()) - v  
 elif op == '\*':  
 v = int(self.\_\_vals.pop()) \* v  
 elif op == '/':  
 v = int(self.\_\_vals.pop()) / v  
 elif op == '^':  
 v = int(self.\_\_vals.pop()) \*\* v  
 self.\_\_vals.push(v)  
 continue  
 else:  
 self.\_\_vals.push(char)  
 return self.\_\_vals.pop()  
  
  
interpreter = Interpreter('(1+((2+3)\*(4\*5)))')  
print(interpreter.execute()) *# 101*interpreter = Interpreter('(2+((2\*3)/(4^5)))')  
print(interpreter.execute()) *# 2*

2 этап

Далее, если всё работает, избавимся от скобок

def \_evaluate(self, code):  
 operators = ['+', '-', '\*', '/', '^']  
 code = f'({code})'  
 for char in code:

Потом мы решили брать из файла

Меняем конструктор

class Interpreter(InterpreterAbstract):  
 def \_\_init\_\_(self, file=None, string=None):  
 if string is not None:  
 self.\_\_code = string  
 else:  
 if file is None:  
 raise AttributeError('Строка или файл')  
 with open(file) as f:  
 self.\_\_code = f.read()  
 self.\_\_ops = Stack()  
 self.\_\_vals = Stack()

и модифицируем метод pars

def \_parse(self): *# непонятно пока* lines = self.\_\_code.strip().split('\n')  
 ret = []  
 for line in lines:  
 line = ''.join(line.split())  
 ret.append(self.\_evaluate(line))  
 return ret

1+((2+3)\*(4\*5))

2+((2\*3)/(4^5))

[101, 2]

Но наш парсер работает пока только с цифрами, но нам же нужны числа, двух, трех, четырех-значные.

Изменяем

def \_evaluate(self, code):  
 operators = ['+', '-', '\*', '/', '^']  
 code = f'({code})'  
 prev\_char = ''  
 for char in code:  
 if char.isdigit():  
 prev\_char += char  
 continue  
 elif prev\_char != '':  
 print(prev\_char)  
 self.\_\_vals.push(prev\_char)  
 prev\_char = ''

дописываем валидатор

def \_validate(self, code):  
 stack = Stack()  
 for char in code:  
 if char == '(':  
 stack.push(char)  
 elif char == ')':  
 stack.push()  
 if stack.is\_empty():  
 return True  
 else:  
 return False

и парсер

def \_parse(self): *# непонятно пока* lines = self.\_\_code.strip().split('\n')  
 ret = []  
 for line in lines:  
 if not self.\_validate(line):  
 raise Exception('Ошибка со скобками')  
 for line in lines:  
 line = ''.join(line.split())  
 ret.append(self.\_evaluate(line))  
 return ret

получили полноценный парсер-валидатор и на скобки, и на числа.

Есть небольшой недостаток, связанный со скобками и со знаками операций.

Архитектурно лучше вынести операнды в отдельный класс.

Вдруг они еще добавиться. Принцип персональной ответственности.

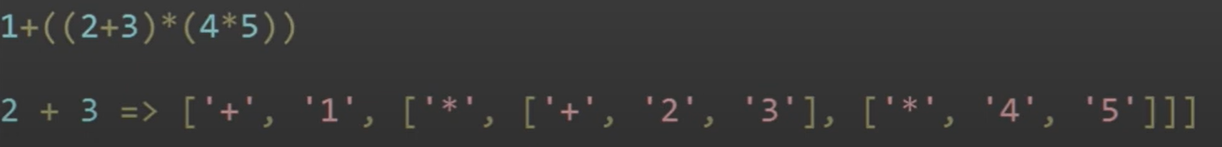
Класс – надстройку.

import enum  
class Token(enum.Enum):  
 PLUS = '+'  
 MINUS = '-'  
 ASTERISK = '\*'  
 SLASH = '/'  
 CARET = '^'  
 EXP\_OPEN = '('  
 EXP\_CLESE = ')'  
  
 def \_\_eq\_\_(self, other):  
 if type(self, other):  
 if type(other) is str:  
 return self.value[0] == other

и главную функцию evaluate

def \_ (self, code):  
 operators = ['+', '-', '\*', '/', '^']  
 code = f'({code})'  
 prev\_char = ''  
 for char in code:  
 if char.isdigit():  
 prev\_char += char  
 continue  
 elif prev\_char != '':  
 self.\_\_vals.push(prev\_char)  
 prev\_char = ''  
  
 for char in code:  
 if char == Token.EXP\_OPEN:  
 continue  
 elif char in operators:  
 self.\_\_ops.push(char)  
 elif char == Token.EXP\_CLOSE:  
 op = self.\_\_ops.pop()  
 v = int(self.\_\_vals.pop())  
 if op == '+':  
 v = int(self.\_\_vals.pop()) + v  
 elif op == Token.MINUS:  
 v = int(self.\_\_vals.pop()) - v  
 elif op == Token.ASTERISK:  
 v = int(self.\_\_vals.pop()) \* v  
 elif op == Token.SLASH:  
 v = int(self.\_\_vals.pop()) / v  
 elif op == Token.CARET:  
 v = int(self.\_\_vals.pop()) \*\* v  
 self.\_\_vals.push(v)  
 continue  
 else:  
 self.\_\_vals.push(char)  
 return self.\_\_vals.pop()

Абстрактное синтаксическое дерево



Код с багом

import abc  
from stack import Stack  
  
  
class InterpreterAbstract(abc.ABC):  
 *'''Интерпретатор кода'''* def \_\_init\_\_(self, code):  
 *'''Принимает код'''* self.code = code  
  
 def execute(self):  
 *'''Запускает механизм исполнения кода  
 Возвращает результат исполнения кода'''* return self.\_parse()  
  
 @abc.abstractmethod  
 def \_parse(self):  
 *'''Осуществляет парсинг кода.  
 Вызывает \_evaluate для исполнения выражений  
 Возвращает результат исполнения кода в excecute'''* pass  
  
 @abc.abstractmethod  
 def \_evaluate(self, code):  
 *'''Осуществляет вычисление выражения  
 Возвращает результат выражения в \_parse'''* pass  
  
  
import enum  
  
  
class Token(enum.Enum):  
 PLUS = '+'  
 MINUS = '-'  
 ASTERISK = '\*'  
 SLASH = '/'  
 CARET = '^'  
 EXP\_OPEN = '('  
 EXP\_CLOSE = ')'  
  
 def \_\_eq\_\_(self, other):  
 if type(other) is str:  
 return self.value[0] == other  
  
  
class Interpreter(InterpreterAbstract):  
 def \_\_init\_\_(self, file=None, string=None):  
 if string is not None:  
 self.\_\_code = string  
 else:  
 if file is None:  
 raise AttributeError('Строка или файл')  
 with open(file) as f:  
 self.\_\_code = f.read()  
 self.\_\_ops = Stack()  
 self.\_\_vals = Stack()  
  
 def execute(self):  
 return self.\_parse()  
  
 def \_parse(self): *# непонятно пока* lines = self.\_\_code.strip().split('\n')  
 ret = []  
 for line in lines:  
 if not self.\_validate(line):  
 raise Exception('Ошибка со скобками')  
 for line in lines:  
 line = ''.join(line.split())  
 ret.append(self.\_evaluate(line))  
 return ret  
  
 def \_validate(self, code):  
 stack = Stack()  
 for char in code:  
 if char == Token.EXP\_OPEN:  
 stack.push(char)  
 elif char == Token.EXP\_CLOSE:  
 stack.push()  
 if stack.is\_empty():  
 return True  
 else:  
 return False  
  
 def \_evaluate(self, code):  
 operators = ['+', '-', '\*', '/', '^']  
 code = f'({code})'  
 prev\_char = ''  
 for char in code:  
 if char.isdigit():  
 prev\_char += char  
 continue  
 elif prev\_char != '':  
 print(prev\_char)  
 self.\_\_vals.push(prev\_char)  
 prev\_char = ''  
  
 for char in code:  
 if char == '(':  
 continue  
 elif char in operators:  
 self.\_\_ops.push(char)  
 elif char == ')':  
 op = self.\_\_ops.pop()  
 v = int(self.\_\_vals.pop())  
 if op == '+':  
 v = int(self.\_\_vals.pop()) + v  
 elif op == '-':  
 v = int(self.\_\_vals.pop()) - v  
 elif op == '\*':  
 v = int(self.\_\_vals.pop()) \* v  
 elif op == '/':  
 v = int(self.\_\_vals.pop()) / v  
 elif op == '^':  
 v = int(self.\_\_vals.pop()) \*\* v  
 self.\_\_vals.push(v)  
 continue  
 else:  
 self.\_\_vals.push(char)  
 return self.\_\_vals.pop()  
  
  
  
interpreter = Interpreter(string = '11+22')  
print(interpreter.execute()) *# 1*interpreter = Interpreter(file='txt1.txt')  
print(interpreter.execute()) *# 101  
  
# interpreter = Interpreter('2+((2\*3)/(4^5))')  
# print(interpreter.execute()) # 2*