



**VISHWAKARMA**  
**UNIVERSITY**  
*Maximising Human Potential*

**Activity based**  
**Project Report on**  
**System programming**  
**Submitted to Vishwakarma University, Pune**  
**Under the Initiative of**  
**Contemporary Curriculum, Pedagogy, and Practice (C2P2)**

**By**

**Merwin Pinto**

**SRN No : 202100102**

**Roll No : 01**

**Diya Oswal**

**SRN No : 202101718**

**Roll No : 40**

**Div : E**

**Third Year Engineering**

**Department of Computer Engineering**

**Faculty of Science and Technology**

**Academic Year**

## Project Statement :

Implementing an Expression calculator based Interpreter in Python and Demonstrating its  
3 Phases: - Lexical , Parser and Execution.

## Project Description :

An Expression interpreter is a computer program that directly executes instructions written in a mathematical Expression . Our main objective is to Demonstrate the Working of our Interpreter using the 3 Phases with Explanation

## Interpreter is further divided into 3 parts

1. Lexical Analysis ( tokens )
2. Parser ( Syntax tree )
3. Execution( executes the program )



## PROJECT MODULE 1 : LEXICAL ANALYSER

**Lexical analysis** is the process of converting a sequence of characters in a source code file into a sequence of tokens that can be more easily processed by our interpreter. It is often the first phase of the compilation process and is followed by Parser / Syntax Analysis

Tokens here mean eg : identifiers , integers, numbers , parentheses etc...

Lexemes are smallest unit of a word

Tokens contain pattern of characters and are defined with set of rules by CFG

**Tokenization in Lexical Analysis** : A token is a sequence of characters that is treated as a single unit by the interpreter. Tokens are the basic building blocks of a program, and they can be classified into different types, such as keywords, identifiers, operators, and punctuation marks. So basically the Entire Program is Divided into tokens and this is Analyzed by the Parser for Syntax and creates Syntax tree to be executed at last

### **examples of tokens:**

**Identifiers:** *a , xyz , main\_prog*

**Operators:** *+, -, \*, /, ==, !=*

**Punctuation marks:** *(, ), {, }, ;,*

### Implementation :

#### Implementation procedure

**Initialize the lexical analyzer:** providing a list of regular expression patterns and their corresponding token types.

These patterns define the types of tokens that the lexical analyzer can identify.

**Defining the `lexer()` method:** This method takes a string of code as input and returns a list of tokens. Each token is a tuple containing the token type and the corresponding lexeme (the portion of the code that matches the token's pattern).

**Iterate through the input code:** Use a loop to iterate through the input code character by character. For each character, determine if it matches any of the regular expression patterns in the token list.

**Check for matching patterns:** For each pattern in the token list, try to match the current character and the following characters against the pattern using the `match()` method of the regular expression object.

**Add matching tokens:** If a pattern matches, create a token tuple containing the token type and the matched lexeme. Add this token to the list of tokens and advance the input code pointer to the position after the matched lexeme.

**Raise error for invalid characters:** If no pattern matches the current character, raise a `SyntaxError` exception indicating an invalid character.

**Return the list of tokens:** Once the entire input code has been processed, return the list of tokens.

### Code :

```
import re
```

```
# LEXICAL ANALYSER
```

```
class lexical_analyzer:
```

```
    def __init__(self):
```

```
        self.tokens = [
```

```
            (re.compile(r'\d+\.\d+|\d+'), "NUMBER"),
```

```
            (re.compile(r'[a-zA-Z]+'), "VARIABLE"),
```

```
            (re.compile(r'[+ \- * % / ^]'), "OPERATOR"),
```

```
            (re.compile(r'\('), "PARENTHESIS"),
```

```
            (re.compile(r'\)'), "PARENTHESIS"),
```

```
            (re.compile(r'\['), "BRACKET"),
```

```
            (re.compile(r'\]'), "BRACKET"),
```

```
            (re.compile(r'\{'), "CBRACKET"),
```

```
            (re.compile(r'\}'), "CBRACKET")
```

```
        ]
```

```
    def lexer(self, code):
```

```
        tokens = []
```

```
        while code:
```

```
    if code[0].isspace():
        code = code[1:]
    else:
        matched = False
        for pattern, token_Type in self.tokens:
            match = pattern.match(code)
            if match:
                tokens.append((token_Type,
match.group()))

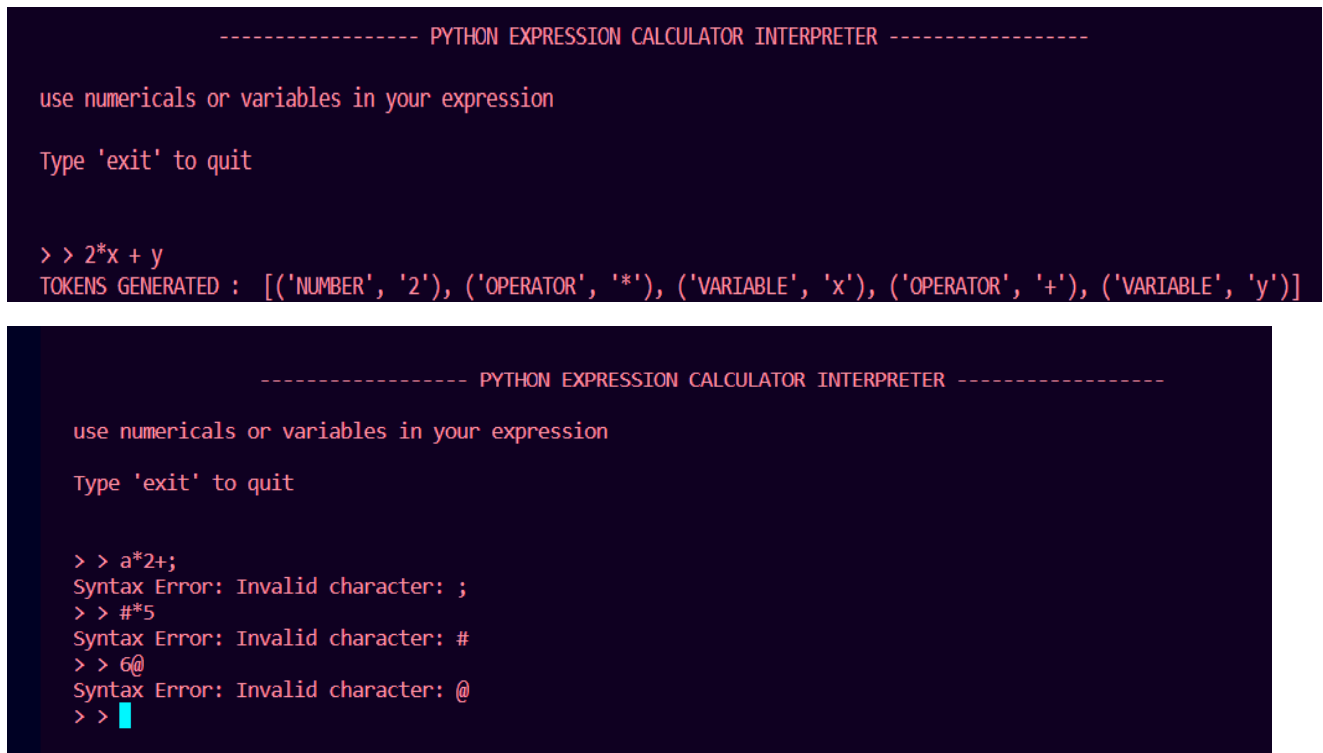
                code = code[len(match.group()):]
                matched = True
                break

        if not matched:
            raise SyntaxError("Invalid character: " +
code[0])

    return tokens
```

### Output:

#### Take a screen shot of project output



```
----- PYTHON EXPRESSION CALCULATOR INTERPRETER -----  
  
use numericals or variables in your expression  
  
Type 'exit' to quit  
  
> > 2*x + y  
TOKENS GENERATED : [('NUMBER', '2'), ('OPERATOR', '*'), ('VARIABLE', 'x'), ('OPERATOR', '+'), ('VARIABLE', 'y')]  
  
----- PYTHON EXPRESSION CALCULATOR INTERPRETER -----  
  
use numericals or variables in your expression  
  
Type 'exit' to quit  
  
> > a*2+;  
Syntax Error: Invalid character: ;  
> > #*5  
Syntax Error: Invalid character: #  
> > 6@  
Syntax Error: Invalid character: @  
> > █
```

### Conclusion :

In this Analysis we are checking the validity of the String whether its correct as per expression to be calculated.

The expression is valid because it has the following structure:

operand operator operand operator  
operand

where "operand" is a number or a variable ( in our expression ) , and "operator" is an arithmetic operator (+, -, \*, /). The expression type is "arithmetic".

In case if there are ([ ]) or { } then parenthesis or bracket or Cbrackets are recognised

**Procedure was followed to check validity oif our expression**

**Check for valid tokens**

**Analyze the expression structure**

**Check for valid operands**

**Check for valid operator**

In case if there is some special character which isnt recognised throws error for that and says invalid character.