**Topic: Lexer & Scanner**

**Course: Formal Languages & Finite Automata**

**Author: Obrijan Filip**

**Theory:**

 The term lexer comes from lexical analysis which, in turn, represents the process of extracting lexical tokens from a string of characters. There are several alternative names for the mechanism called lexer, for example tokenizer or scanner. The lexical analysis is one of the first stages used in a compiler/interpreter when dealing with programming, markup or other types of languages.

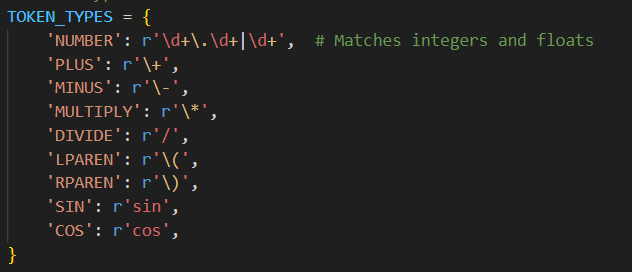
The tokens are identified based on some rules of the language and the products that the lexer gives are called lexemes. So basically the lexer is a stream of lexemes. Now in case it is not clear what's the difference between lexemes and tokens, there is a big one. The lexeme is just the byproduct of splitting based on delimiters, for example spaces, but the tokens give names or categories to each lexeme. So the tokens don't retain necessarily the actual value of the lexeme, but rather the type of it and maybe some metadata.

**Objectives:**

1. Understand what lexical analysis [1] is.
2. Get familiar with the inner workings of a lexer/scanner/tokenizer.
3. Implement a sample lexer and show how it works.

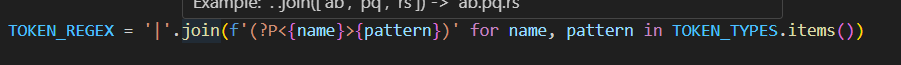
**Note:** Just because too many students were showing me the same idea of lexer for a calculator, I've decided to specify requirements for such case. Try to make it at least a little more complex. Like, being able to pass integers and floats, also to be able to perform trigonometric operations (cos and sin). **But it does not mean that you need to do the calculator, you can pick anything interesting you want**

**Implementation description:**



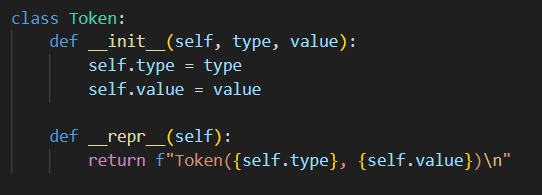
What It Does:

* This dictionary maps **token names** (like NUMBER, PLUS, SIN) to their **regular expression patterns**.
* Example:
  + NUMBER: Matches integers (5) and floats (5.5).
  + SIN: Matches "sin" in expressions like "sin(30)".



What It Does:

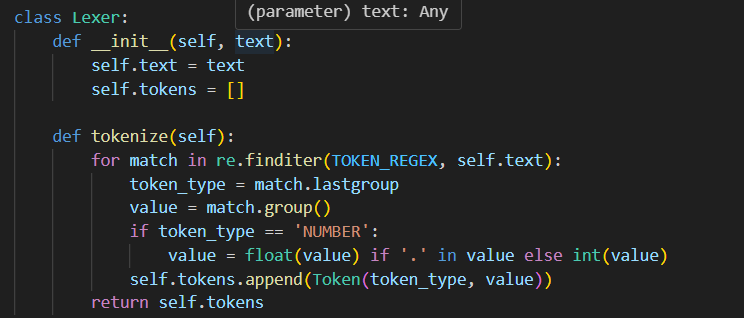
* Constructs a single **regular expression** from all token patterns.
* Uses **named capturing groups** (?P<NAME>) to extract matched tokens.

  
What It Does:

Represents **a single token** with:

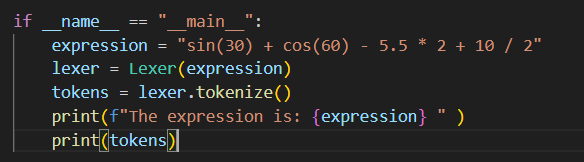
* type: The category of the token (NUMBER, PLUS, etc.).
* value: The actual value from the expression (5.5, sin, +, etc.).

\_\_repr\_\_: Defines how tokens are printed.



What It Does:

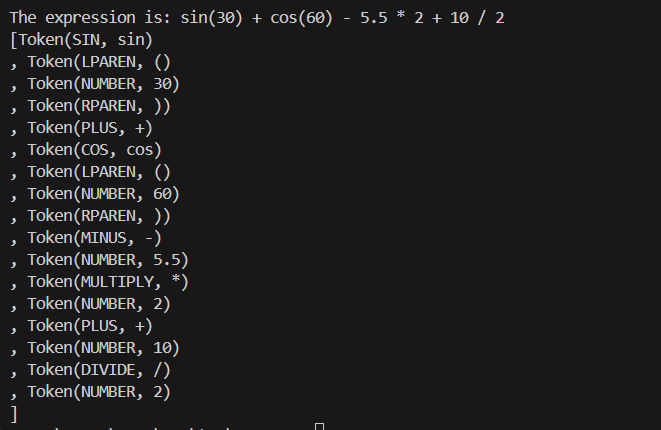
* Stores:
  + text: The input expression ("sin(30) + cos(60) - 5.5 \* 2 + 10 / 2").
  + tokens: A list to store extracted tokens.
* Uses **regular expressions** to find tokens in the input text.
* Extracts:
  + token\_type: The token category (NUMBER, PLUS, etc.).
  + value: The matched text ("30", "+", etc.).
* Converts numbers (30, 5.5) to **int** or **float**.
* Stores tokens in self.tokens and returns them.



What It Does:

* **Defines an expression**: "sin(30) + cos(60) - 5.5 \* 2 + 10 / 2".
* **Creates a Lexer object** and tokenizes the expression.

**Output example:**

****

**Conclusion:**

This laboratory work provided an in-depth understanding of lexical analysis and its role in language processing. By implementing a lexer in Python, we successfully identified and categorized different components of a mathematical expression, including numbers, arithmetic operators, and trigonometric functions. The process of tokenization, achieved through the use of regular expressions, demonstrated how raw text can be converted into meaningful tokens that can later be processed by a parser or an interpreter.

One of the key takeaways from this implementation was the importance of defining clear tokenization rules. By specifying distinct token types for numbers, operators, and mathematical functions, we ensured accurate identification and categorization of expression components. Additionally, using Python's re module allowed for efficient pattern matching, making the lexer both powerful and flexible.

Beyond this basic implementation, there is significant potential for expansion. Future improvements could include support for additional mathematical functions, variables, and logical operators. Furthermore, integrating this lexer into a complete interpreter or compiler would be a valuable next step in understanding language processing pipelines.

In conclusion, this project reinforced the foundational principles of lexical analysis while providing practical experience in handling token streams. The ability to break down complex expressions into structured elements is essential for building programming languages, domain-specific languages (DSLs), and various computational tools.

**References:**

1.Else Course FAF.LFA21.1

2. [1] [A sample of a lexer implementation](https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html)

3. [2] [Lexical analysis](https://en.wikipedia.org/wiki/Lexical_analysis)