PolyLangInterpreter - Full Code Explanation

Overview

This Python class 'PolyLangInterpreter' is a custom-made interpreter for a small language called 'Poly'. It mimics the behavior of simple programming languages, supporting variables, functions, closures, recursion, and if-else conditions. The goal of this interpreter is to read code written in the 'Poly' syntax and execute it as if it were a scripting language.

1. Initialization (__init__)

The constructor initializes an empty dictionary 'self.context'. This dictionary stores all variables and functions defined by the user during code execution. It acts like memory or a symbol table that keeps track of names and their values.

2. eval_expr Method

This method evaluates expressions. It handles both normal mathematical/logical expressions and inline if-then-else conditions. It uses Python’s eval() to execute expressions safely within a controlled context. For example, 'if x > 5 then 10 else 0' becomes '(10) if (x > 5) else (0)' before evaluation.

3. make_func Method

This method builds functions dynamically. It allows Poly to define its own functions, even recursive or nested (closure) ones. When a function is defined using 'func add(a,b)=a+b', this method creates a callable Python function object. It captures the outer context so closures work correctly. Recursion is handled by referencing the function by name.

4. parse_line Method

This is the line-by-line parser for Poly code. It identifies what each line means and acts accordingly: - If it starts with 'print(', it outputs values. - If it starts with 'let', it defines a variable. - If it starts with 'func', it defines a function. - If it's a function call, it executes it. It also handles string interpolation (like print('Hello {name}')).

5. run Method

This method runs an entire Poly program. It processes the input code line by line, managing if-else blocks and nested structures. It uses an output buffer to capture results. The interpreter can run complete scripts, not just single lines.

6. Why We Need These Parts

• 'context' keeps track of program state (variables/functions). • 'eval_expr' executes logic and math. • 'make_func' enables function creation and recursion. • 'parse_line' interprets commands. • 'run' controls the program flow and executes blocks. Together, they make a fully functional mini-language interpreter.