🧩 FILE 1: `python_lexer.py` — *Lexical Analysis (Tokenization)*

🧠 What it does:

Breaks Python-like code into **tokens** (the smallest building blocks, like words in English).

Example:
`x = 5 + 2` → Tokens: `NAME(x)`, `ASSIGN(=)`, `NUMBER(5)`, `PLUS(+)`, `NUMBER(2)`

🔍 Main parts:

Part	Explanation
`import ply.lex as lex`	Imports the PLY lexer tool.
`tokens = (...)`	Lists all token types (keywords, operators, etc.)
`reserved = {...}`	Maps Python keywords (`def`, `class`, etc.) to token names.
`t_ASSIGN = r'='` etc.	Defines patterns for symbols like `=`, `+`, `==`, etc.
`t_STRING`	Regex for `'string'` or `"string"` literals.
`t_NUMBER`	Matches integers only (`123`), and converts text to `int`.
`t_NAME`	Matches variable names like `x`, `_value`. If it's a keyword, marks it accordingly.
`t_NEWLINE`	Tracks new lines and updates the line number.
`t_ignore = ' \t'`	Ignores spaces and tabs (so indentation isn't handled).
`t_error`	Called if an illegal character appears (prints message & skips it).
`lexer = lex.lex()`	Builds the lexer object.
`if __name__ == "__main__"`	Test code — tokenizes a short sample input.

💬 Key things to mention if asked:

- **Main job**: Split code into tokens.
- **PLY** uses regex rules to match tokens.
- **INDENT/DEDENT** tokens are *declared* but not implemented (so no real indentation handling yet).
- **Limitations**: only integers, no floats, no triple-quoted strings, no comments.

⚙️ FILE 2: `python_parser.py` — *Syntax Analysis (Parsing)*

🧠 What it does:

Takes the **tokens from the lexer** and checks if they follow valid Python grammar rules.

Example:
It checks if something like

```
def add(a, b):
    return a + b
```

follows the pattern `DEF NAME (parameters) : suite`.

🔍 Main parts:

Part	Explanation
`import ply.yacc as yacc`	Imports the PLY parser tool.
`from python_lexer import tokens, lexer`	Uses the lexer from the previous file.
`precedence`	Defines order for operators like `+`, `-`, `*`, `/`.
Grammar functions `p_*`	Define syntax rules. Example:

	`p_func_decl : DEF NAME LPAREN parameters RPAREN COLON suite`
`p_statements`	Allows multiple or single statements.
`p_statement`	Recognizes valid statements (assignment, class, func, return, pass).
`p_lambda_expr`	Recognizes `lambda` expressions.
`p_list_comprehension`	Recognizes `[x for x in ... if ...]`.
`p_class_decl`, `p_func_decl`	Recognize class and function declarations.
`p_dict_method_call`	Recognizes object method calls like `a.get(...)`.
`p_suite`	Simplified — only one statement after `:`. (No indentation logic.)
`p_error(p)`	Handles syntax errors and prints the line number.
`validate_syntax(code)`	Main function called by `validator.py` — returns `"Valid Syntax"` or `"Syntax Error"`.

💬 Key things to remember:

- **PLY yacc** uses grammar rules (`p_*` functions) to check valid syntax.
- The parser **does not build a full AST**, just validates structure.
- The grammar supports **a limited subset of Python**:
    - `def`, `class`, `lambda`, `list comprehension`, assignments, expressions.
- **No indentation parsing**, so multi-line functions/classes only work superficially.
- Returns `"Valid Syntax"` or `"Syntax Error"`.

⚠️ Common questions & answers:

Q: What happens if you write invalid code?
A: `p_error()` catches it, prints the message, and sets `syntax_error_occurred = True`.

Q: Why add a newline at the end of the code in `validate_syntax`?
A: Some grammar rules expect a newline to mark the end of a statement, so it ensures the parser doesn't stop early.

Q: Why is indentation ignored?
A: Because lexer ignores spaces/tabs and INDENT/DEDENT rules are not implemented.

🧪 FILE 3: `validator.py` — *Test & Interactive Checker*

🧠 What it does:

Uses the `validate_syntax()` function from `python_parser.py` to:

1. Test multiple code examples automatically.
2. Let the user type code to check syntax interactively.

🔍 Main parts:

Part	Explanation
`from python_parser import validate_syntax`	Imports the syntax checker.
`test_cases = [...]`	Predefined valid/invalid code snippets to test.
`run_tests()`	Runs all test cases, prints pass/fail results.
`interactive_validation()`	Lets user type one line of code and tells if syntax is valid.
`if __name__ == "__main__":`	Runs tests first, then starts interactive mode.

💬 Key points:

- The tests check **lambda**, **list comprehension**, **class**, **function**, and **dictionary method** syntax.
- Example valid: `my_func = lambda a, b: a + b`
- Example invalid: `lambda a, b a + b` (missing `:`)
- Interactive mode warns: *"single line only"* because no proper indentation support for multi-line blocks.

🧩 HOW THE 3 FILES WORK TOGETHER

Step	File	Role
1	`python_lexer.py`	Breaks input into tokens.
2	`python_parser.py`	Checks if token sequence follows correct syntax.
3	`validator.py`	Tests or interacts with user to validate code.

⚡ Common Teacher Questions + Answers

Question	Answer
1. What is the purpose of the lexer?	To convert source code into tokens using regex patterns.
2. What does the parser do?	Checks if tokens follow grammar rules (syntax validation).
3. What tool is used here?	**PLY (Python Lex-Yacc)**.
4. What happens on invalid syntax?	The parser calls `p_error`, sets a flag, and `validate_syntax` returns `"Syntax Error"`.

5. What are tokens like `NAME`, `NUMBER`, `STRING`?	Categories of words the lexer recognizes.
6. Why declare INDENT/DEDENT tokens?	To plan for indentation handling like Python — but not implemented yet.
7. What's a lambda expression?	A short anonymous function, e.g. `lambda x: x+1`.
8. What's a list comprehension?	A compact way to build lists, e.g. `[x for x in range(5)]`.
9. What's the biggest limitation?	No indentation (multi-line) or advanced syntax support (imports, booleans, etc.).
10. What does `validate_syntax("code")` return?	`"Valid Syntax"` or `"Syntax Error"`.

🧭 Quick Revision Summary

```
python_lexer.py  → tokenizes the code
python_parser.py → checks grammar validity
validator.py     → runs tests & interactive checker
```

- Lexer uses regex to detect keywords, numbers, strings, etc.
- Parser defines grammar with functions `p_*`.
- Together, they can validate small Python-like code structures.
- Main missing feature: indentation handling (INDENT/DEDENT).