Python
Lex and Yacc

Prepared by

Zeyad moneer abo El-Zahab

Agenda

- What is PLY ?
- Components of Python lex
- Tokens Specification
- Token values
- Discarding tokens
- Line numbers and positional information
- Ignored characters
- Literal characters
- Error handling
- EOF Handling

What is PLY (Python Lex –Yacc)

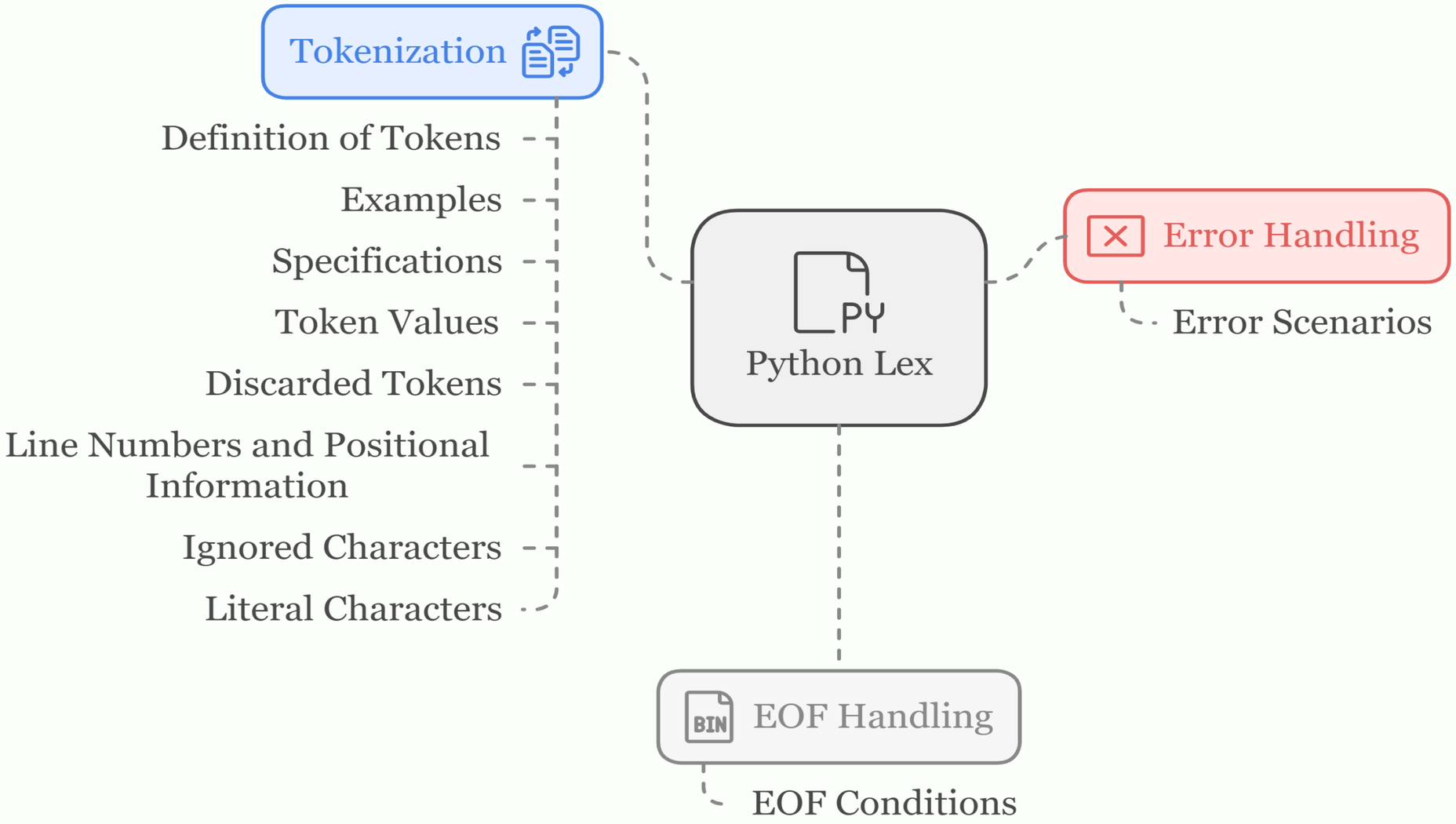
- ✓ **PLY consists of two separate modules; lex.py and yacc.py, both of which are found in a Python package called ply.**
- ✓ **It is a pure Python implementation of the traditional Lex and Yacc tools, used for lexical analysis and parsing.**
    - **lex.py** is used to break input text into a collection of tokens specified by a collection of regular expression rules.
    - **yacc.py** is used to recognize language syntax that has been specified in the form of a context free grammar.

Components of Python LEX

✓ **Python Lex (Lexer):**

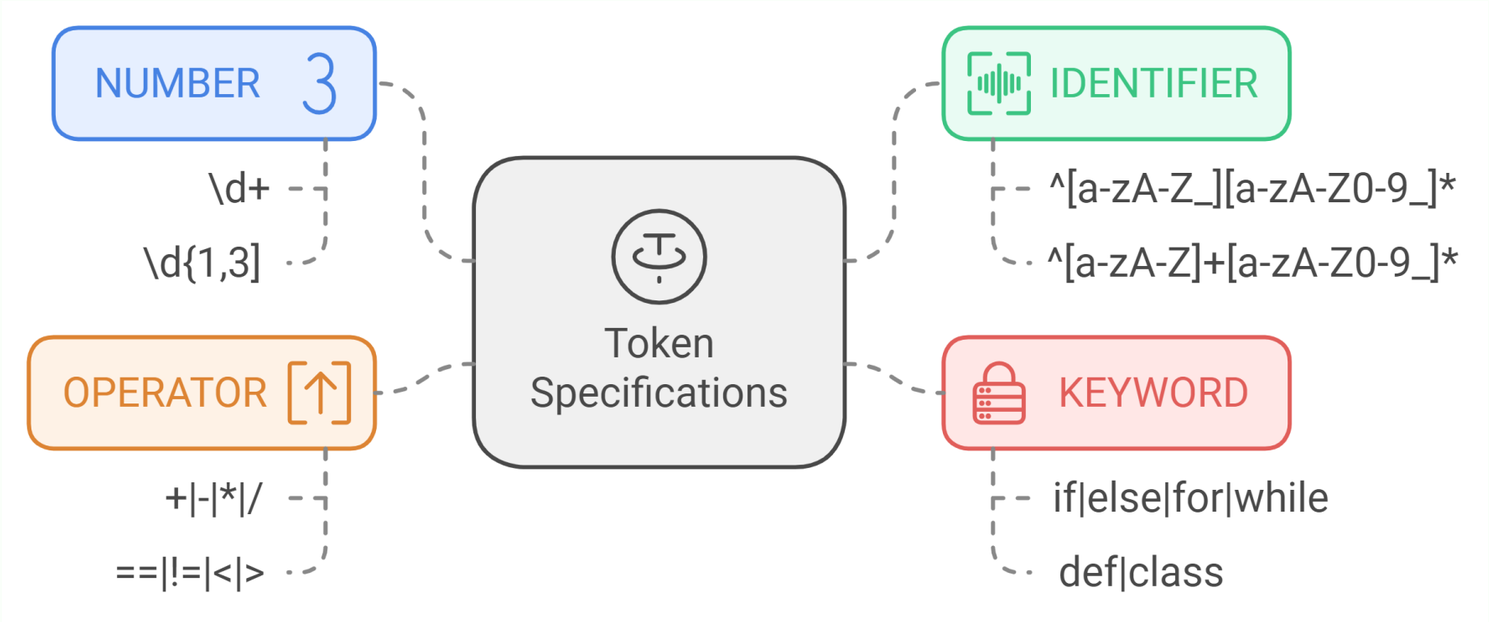
- Responsible for breaking the input text into a sequence of tokens.
- Tokens represent atomic units such as keywords, operators, identifiers, etc.
- Defined using regular expressions.

Python Lex


Tokenization
Definition of Tokens
Examples
Specifications
Token Values
Discarded Tokens
Line Numbers and Positional Information
Ignored Characters
Literal Characters
PY
Python Lex
Error Handling
Error Scenarios
BIN
EOF Handling
EOF Conditions

Token Specification

- ✓ **Tokens are defined using regular expressions.**
- ✓ **Rules are declared with the prefix (t_) (e.g. , t_PLUS = r'\+').**
- ✓ **Token functions can also perform actions, like converting matched text into other data types.**

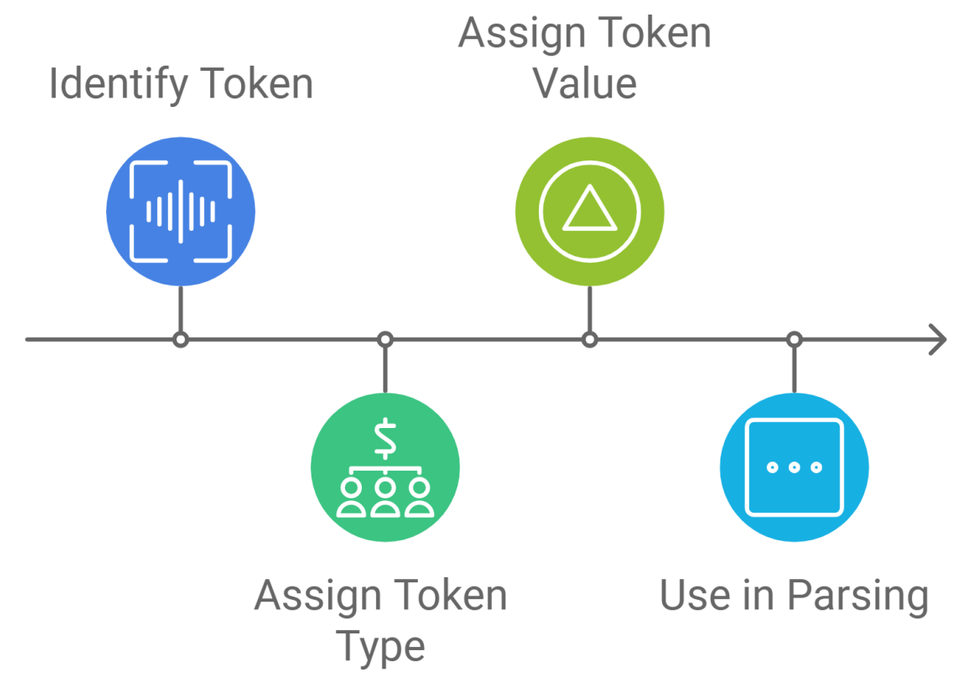


Token Specification

```
# List of token names.   This is always required
tokens = (
    'NUMBER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
)
```

Token Values

- **When the lexer identifies a token, it assigns a value to it. For example, if the input string contains the number 3, the lexer will create a token of type NUMBER with the value 3. This value can be used later in parsing or evaluation.**
- **Values can be modified to store additional data.**



Token Values

```
# Regular expression rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'\-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'
```

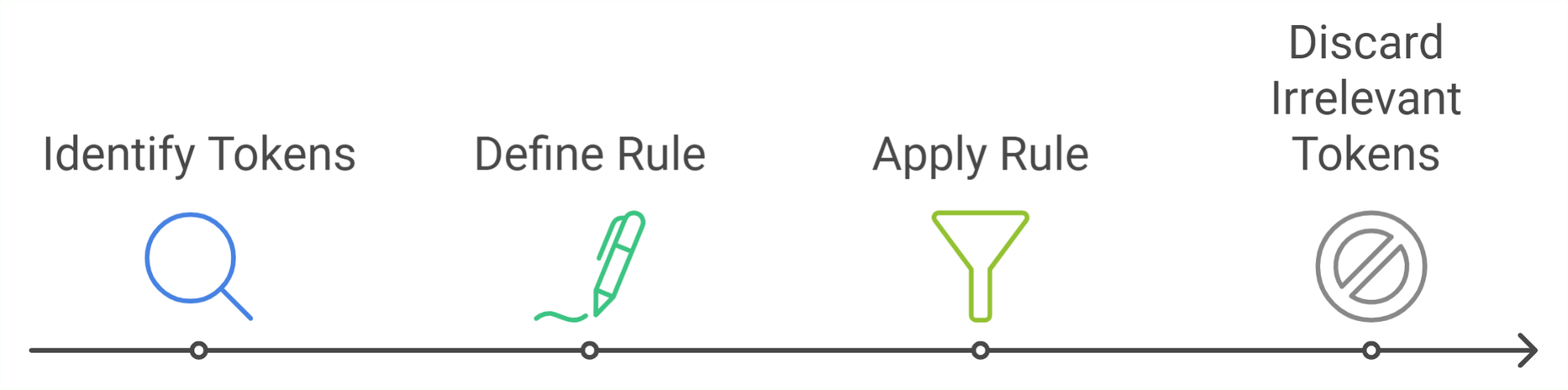
```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    t.datatype = 'integer'    # additional attribute
    return t
```

الرمز	المعنى الخاص	مثال	التخليص (للمطابقة الحرفية)	مثال التخليص
.	مطابقة أي حرف مفرد (ما عدا السطر الجديد)	a.b يطابق "a" متبوعة بأي حرف ثم "b" مثل "acb" أو "a1b"	.\	r'a\.b' يطابق "a.b" فقط
+	واحد أو أكثر من الحرف الذي قبله	a+ يطابق "a" واحدة أو أكثر (مثل "a" أو "aaa")	+\	r'a\+b' يطابق "a+b" فقط
*	صفر أو أكثر من الحرف الذي قبله	a* يطابق "a" صفر أو أكثر (مثل "" أو "aaaa")	*\	r'a*b' يطابق "a*b" فقط
?	صفر أو مرة واحدة من الحرف الذي قبله	a?b يطابق "b" أو "ab"	?\	r'a\?b' يطابق "a?b" فقط
{ }	تحديد عدد التكرار	a{3} يطابق "aaa" (ثلاث مرات فقط)	{\ }\	r'a\{3\}' يطابق "a{3}" فقط
\	تخليص الرموز الخاصة (Escape character)	\d يطابق أي رقم	\\	r'\\' يطابق "\" فقط
[ ]	مجموعة من الأحرف	[abc] يطابق "a" أو "b" أو "c"	[\]\	r'\[abc\]' يطابق "[abc]" فقط
`	`	"أو" (OR) بين الأنماط	`a	a" أو "b" يطابق`b
$	نهاية السطر	abc $ يطابق "abc" فقط إذا كانت في نهاية السطر	$\	r'\$100' يطابق "100$" فقط
^	بداية السطر	^ abc يطابق "abc" فقط إذا كانت في بداية السطر	^\	r'\^abc' يطابق "^abc" فقط
()	إنشاء مجموعة (Group)	(abc) يطابق "abc" ككتلة واحدة	(\)\	r'\(abc\)' يطابق "(abc)" فقط

Discarding Tokens

- **Some tokens may not be relevant for further processing and can be discarded.**

  For instance, comments or whitespace can be ignored by defining a rule to skip them. In the example, whitespace is ignored using `t_ignore = '\t'`



Line Numbers and Positional Information

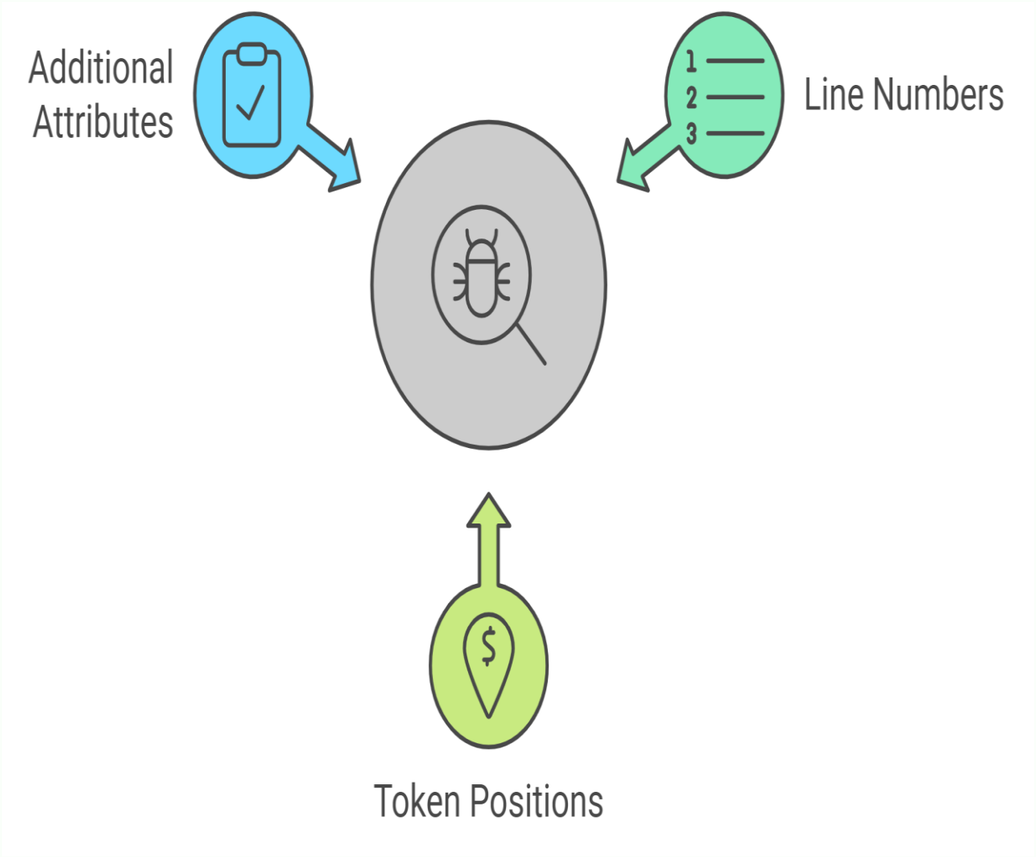
- ✓ **lex.py doesn't know anything about what constitutes a "line" of input. To update this information, you need to write a special rule. In the example, the t_newline() rule shows how to do this.**
- ✓ **Within the rule, the lineno attribute of the underlying lexer t.lexer is updated. After the line number is updated, the token is simply discarded since nothing is returned.**
- ✓ **lex.py does not perform any kind of automatic column tracking. However, it does record positional information related to each token in the lexpos attribute.**

Line Numbers and Positional Information (cont.)

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    t.datatype = 'integer'    # additional attribute
    return t
```

```
data = '3 + 5'
```

```
Token: NUMBER, Value: 3, Line: 1, Position: 0, Datatype: integer
Illegal character ' ' at line 1, position 1
Token: PLUS, Value: +, Line: 1, Position: 2, Datatype: N/A
Illegal character ' ' at line 1, position 3
Token: NUMBER, Value: 5, Line: 1, Position: 4, Datatype: integer
```



Ignored characters

- ✓ **Characters that are not part of any defined token can be ignored. This is typically done for whitespace and comments. The lexer can be configured to skip these characters to streamline the tokenization process.**

- ✓ **Improves performance.**

```
# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'
```

Literal Characters

- **A literal character is simply a single character that is returned "as is" when encountered by the lexer.**

- **Literals are checked after all of the defined regular expression rules.**

- **When a literal token is returned, both its type and value attributes are set to the character itself. For example, '+'.**

- **In the arithmetic example, the operators +, -, *, and / are treated as literal characters and are defined as tokens.**

```
literals = ['+', '-', '*', '/', '(', ')']
```

Error handling

- **Definition of t_error() Function**
  - **Purpose: Handles lexing errors when illegal characters are found.**
  - **t.value: Contains the rest of the input that hasn't been tokenized.**

```
# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)
```

- **Skipping Invalid Input**
  - **t.lexer.skip(1): Moves the lexer forward by 1 character to avoid infinite loops.**

EOF Handling

- **Handles End-of-File (EOF): Called when the input reaches the end.**
- **t.lexer.input(more): Accepts more input to continue lexing.**
- **t.lexer.token(): Returns the next available token after getting new input.**

```
# EOF handling
def t_eof(t):
    more = input('More input: ')
    if more:
        t.lexer.input(more)
        return t.lexer.token()
    return None
```

Thank you

Any questions?