
# Python Lex and Yacc

*Prepared by*

*Zeyad moneer abo El-Zahab*

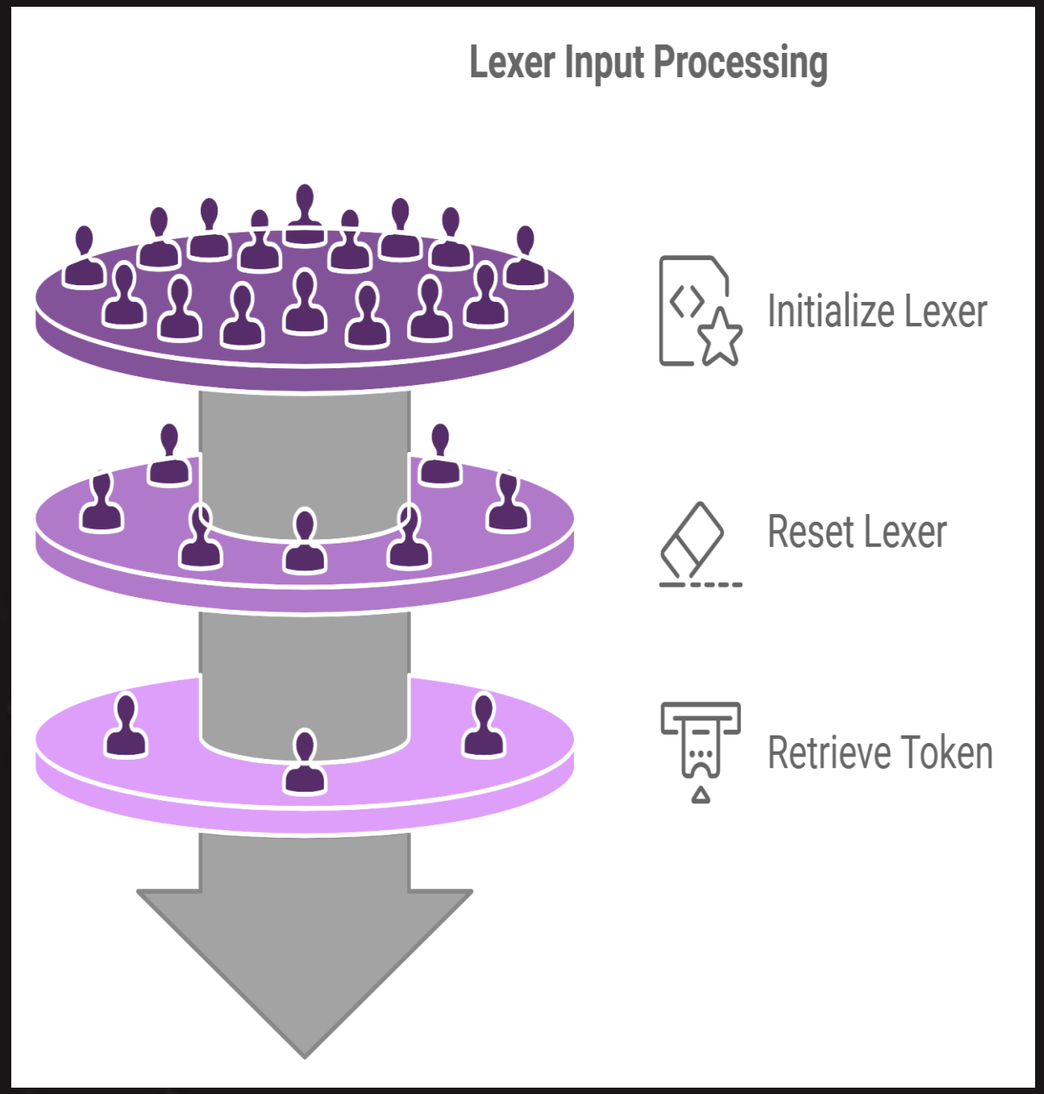
# Agenda

- ❑ Building and using the lexer
- ❑ The @TOKEN decorator
- ❑ Optimized mode
- ❑ Debugging
- ❑ Alternative specification of lexers
- ❑ Lexer cloning
- ❑ Internal lexer state
- ❑ Conditional lexing and start conditions
- ❑ How to use conditional analysis ?

# Building and using the lexer

- This function uses Python reflection to read the  regular expression rules out of the calling context and build the lexer. Once the lexer has been built, two methods can be used to control the lexer.

  - lexer.input(data). Reset the lexer and store a new input string.
  - lexer.token(). Return the next token. Returns special  LexToken instance on success or None if the end of the input text has been reached.



# Building and using the lexer

```
LexToken(WORD,'hello',1,0)
LexToken(NUMBER,'123',1,6)
LexToken(WORD,'world',1,10)
LexToken(NUMBER,'456',1,16)
```

lex.py > ...

```python
import ply.lex as lex

tokens = (
    'NUMBER',
    'WORD',
)

t_WORD = r'[a-zA-Z]+'
t_NUMBER = r'\d+'

t_ignore = ' \t'

def t_error(t):
    print(f"illegal char :{t.value[0]}")
    t.lexer.skip(1)

lexer = lex.lex()

data = "hello 123 world 456"

lexer.input(data)

while True:
    tok = lexer.token()
    if not tok:
        break
    print(tok)

```

# The @TOKEN decorator

- in some applications, you may want to define build tokens from as a series of more complex regular expression rules.

The `@TOKEN` decorator provides a more convenient way to define tokens compared to traditional methods.

**Improved Readability**

Clearer and more concise code, making it easier to understand the lexer's structure.

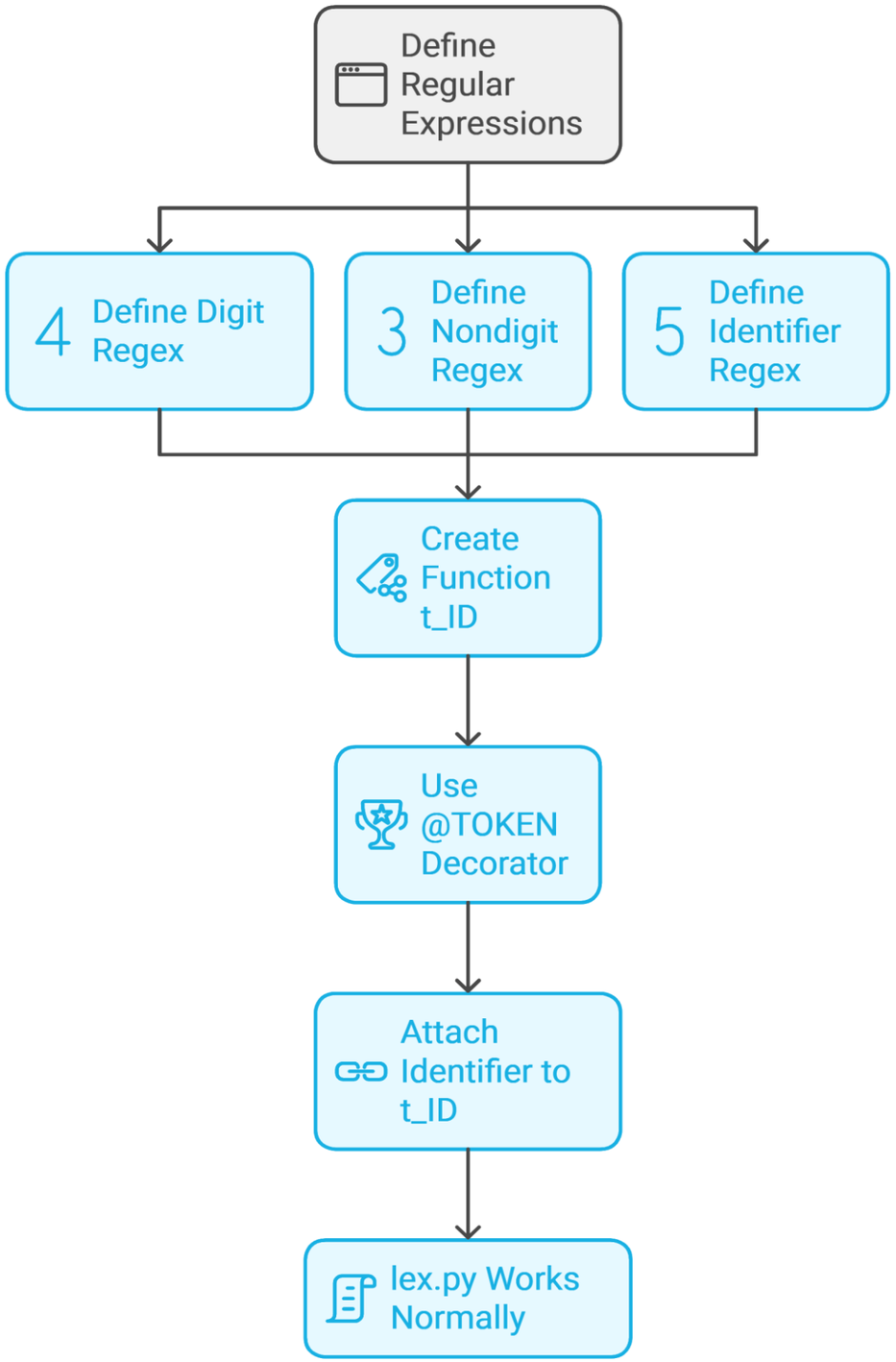
**Reduced Boilerplate**

The decorator takes care of common lexer setup, reducing the amount of code you need to write.

# The @TOKEN decorator

@token.py > ...

```python
from ply.lex import TOKEN

digit = r'([0-9])'
nondigit = r'([_A-Za-z])'
identifier = r'(' + nondigit + r'(' + digit + r'|' + nondigit + r')*)'

@TOKEN(identifier)
def t_ID(t):
    print("it's identifier : ", t.value)
    return t

# ------------------------------------------------------------
# Explain the difference

import ply.lex as lex

def t_ID(t):
    r'([_A-Za-z])([_A-Za-z0-9])*'
    print("it's identifier : ", t.value)
    return t
```



# Optimized mode

- Purpose of Optimizations:

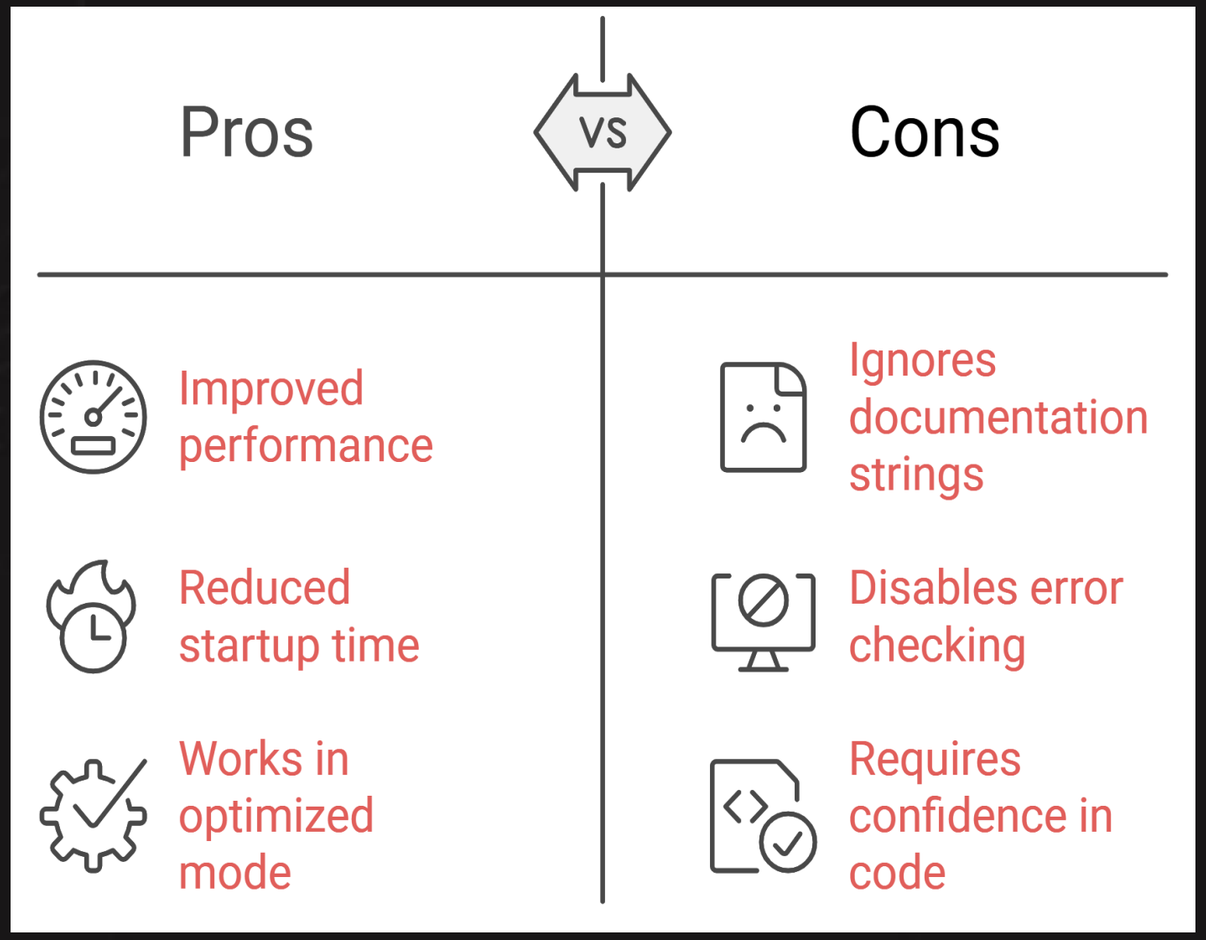
  Reducing memory usage and processing time during lexing.

- Enabling Optimized Mode:

  Setting optimize=1 in the lex() function.

- Implications:

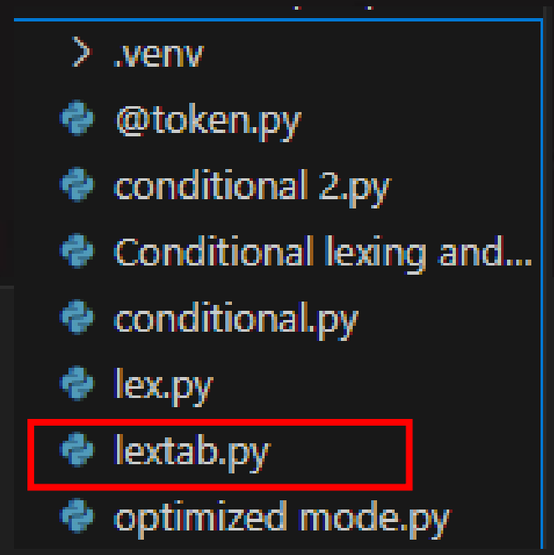
  Trade-off between detailed error reporting and performance gains in optimized mode.

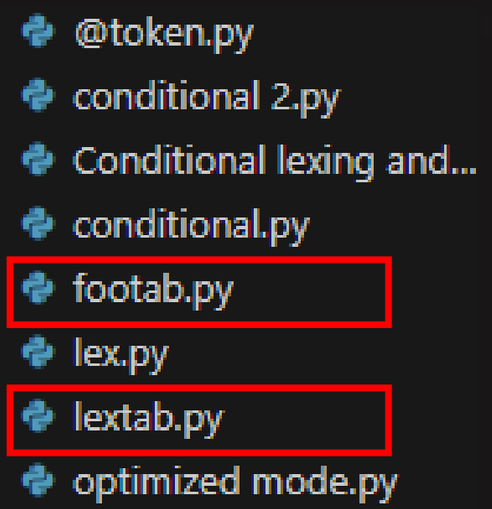


# Optimized mode

optimized mode.py > ...

```python
import ply.lex as lex

tokens = ('PRINT', 'IF', 'WHILE')

t_PRINT = r'print'
t_IF = r'if'
t_WHILE = r'while'

# optimize mode
lexer = lex.lex(optimize=1)

# change the file name, It is automatically imported on subsequent runs.
lexer = lex.lex(optimize=1, lextab="footab")
```



# Debugging

1

## Enabling Debugging

Set (debug=1) to track lexer behavior.

2

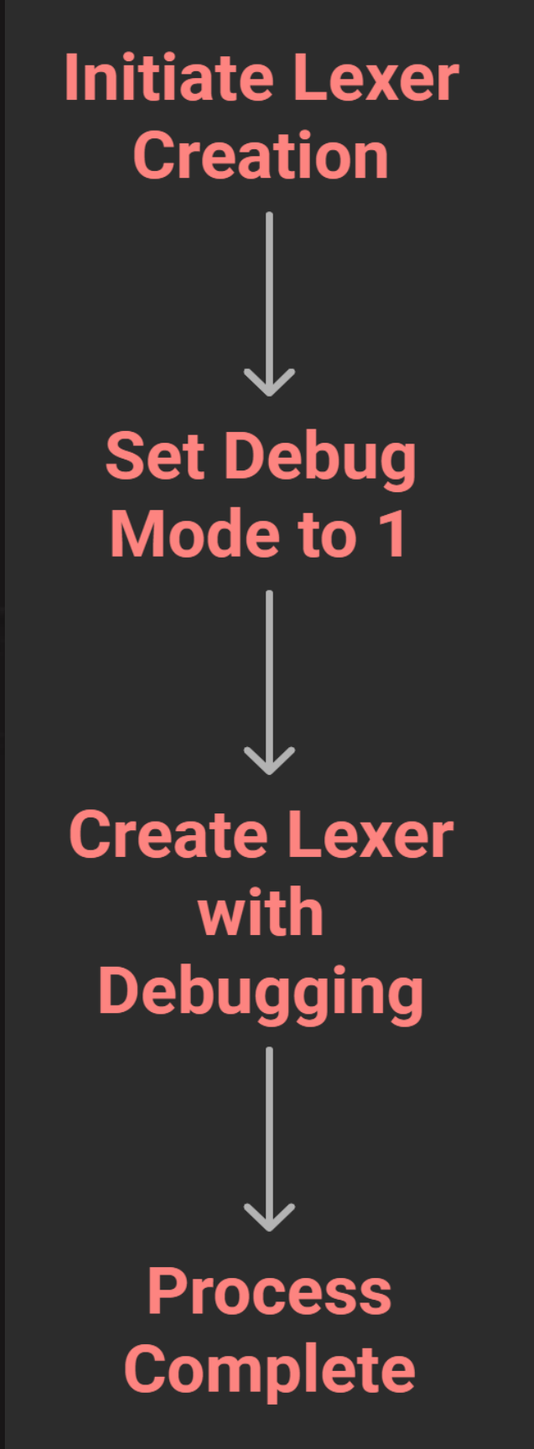
## Print Statements

Insert print statements in your lexer rules to track the tokenization process.

3

## Logging

Utilize a logging framework to record token information for later analysis.



# Debugging

lexer.log

```
DEBUG:root:Token NUMBER: 3 at position 0
INFO:root:Token: LexToken(NUMBER,3,1,0)
ERROR:root:Illegal character ' ' at position 1
INFO:root:Token: LexToken(PLUS,'+',1,2)
ERROR:root:Illegal character ' ' at position 3
DEBUG:root:Token NUMBER: 5 at position 4
INFO:root:Token: LexToken(NUMBER,5,1,4)
ERROR:root:Illegal character ' ' at position 5
INFO:root:Token: LexToken(TIMES,'*',1,6)
ERROR:root:Illegal character ' ' at position 7
DEBUG:root:Token NUMBER: 10 at position 8
INFO:root:Token: LexToken(NUMBER,10,1,8)
ERROR:root:Illegal character ' ' at position 10
INFO:root:Token: LexToken(MINUS,'-',1,11)
ERROR:root:Illegal character ' ' at position 12
DEBUG:root:Token NUMBER: 4 at position 13
INFO:root:Token: LexToken(NUMBER,4,1,13)
ERROR:root:Illegal character ' ' at position 14
INFO:root:Token: LexToken(DIVIDE,'/',1,15)
ERROR:root:Illegal character ' ' at position 16
DEBUG:root:Token NUMBER: 2 at position 17
INFO:root:Token: LexToken(NUMBER,2,1,17)
```

debug.py > ...

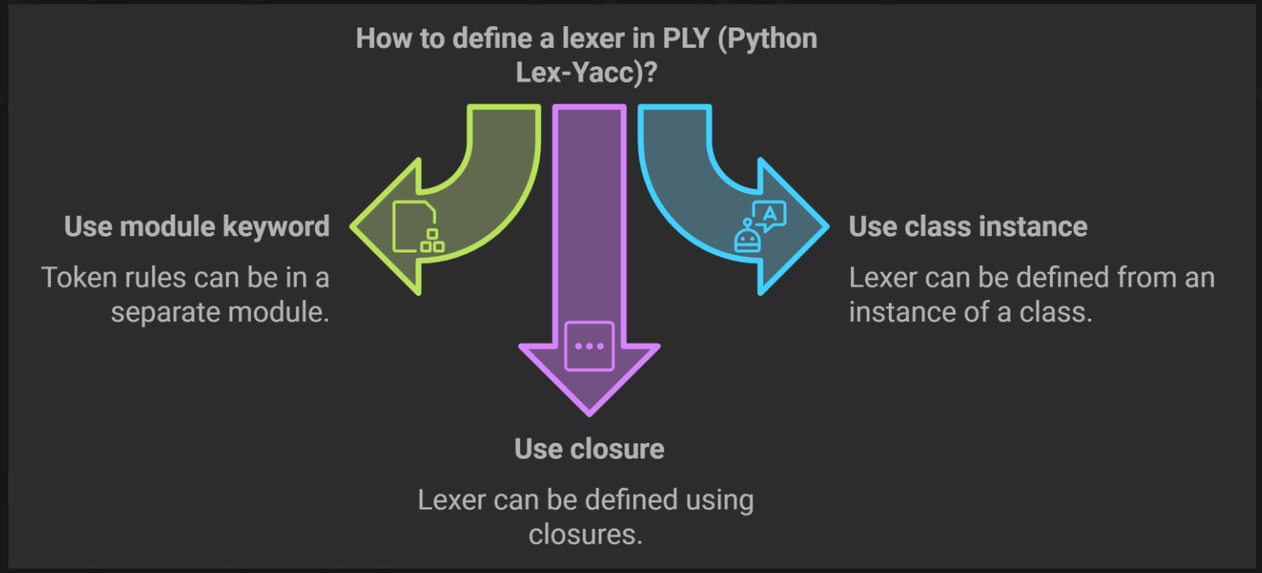
```python
import ply.lex as lex
import logging

# logging
logging.basicConfig(filename='lexer.log', level=logging.DEBUG)

tokens = ('NUMBER', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE')

t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    logging.debug(f"Token NUMBER: {t.value} at position {t.lexpos}")
    return t

def t_error(t):
    logging.error(f"Illegal character '{t.value[0]}' at position {t.lexpos}")
    t.lexer.skip(1)

# Debug mode
lexer = lex.lex(debug=1)

data = "3 + 5 * 10 - 4 / 2"
lexer.input(data)

for token in lexer:
    logging.info(f"Token: {token}")
```

- lex.py
- lexer.log
- optimized mode.py

# Alternative Specification of Lexers

- Alternative specification of lexers refers to different methods or approaches used to define and implement lexers, Unlike traditional lexer generators that use regular expressions, alternative specifications may utilize different frameworks ( class , module , closure, etc. )

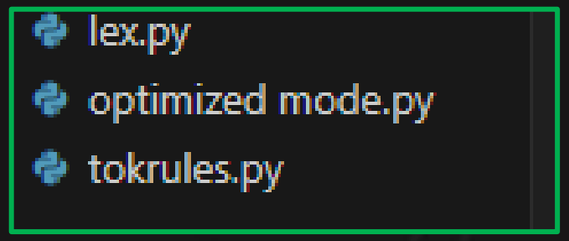
- **purpose :** These alternatives can improve flexibility, error handling, and performance depending on the specific needs of the programming language or application."



# Alternative Specification of Lexers

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

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




ALT.py > ...

```
1   import ply.lex as lex
2   import tokrules
3   lexer = lex.lex(module=tokrules)
4   lexer.input("3 + 4 * 1")
5   while True:
6       tok = lexer.token()
7       if not tok:
8           break
9       print(tok)
```

# Lexer cloning

- Duplicate Lexers
  - ✓ Create multiple copies of the original lexical analyzer object so that these copies can process multiple text streams independently.
- Independent State
  - ✓ Duplicate copies of the lexical analyzer maintain their own internal state, preventing information or data from interfering between different copies.
- Parallel Processing
  - ✓ The cloning feature can be used to perform parallel processing operations, so that multiple inputs are analyzed simultaneously using cloned copies.

# Lexer cloning

```
Number of numbers in the expression: 3
The expression has an odd number of numbers.
```

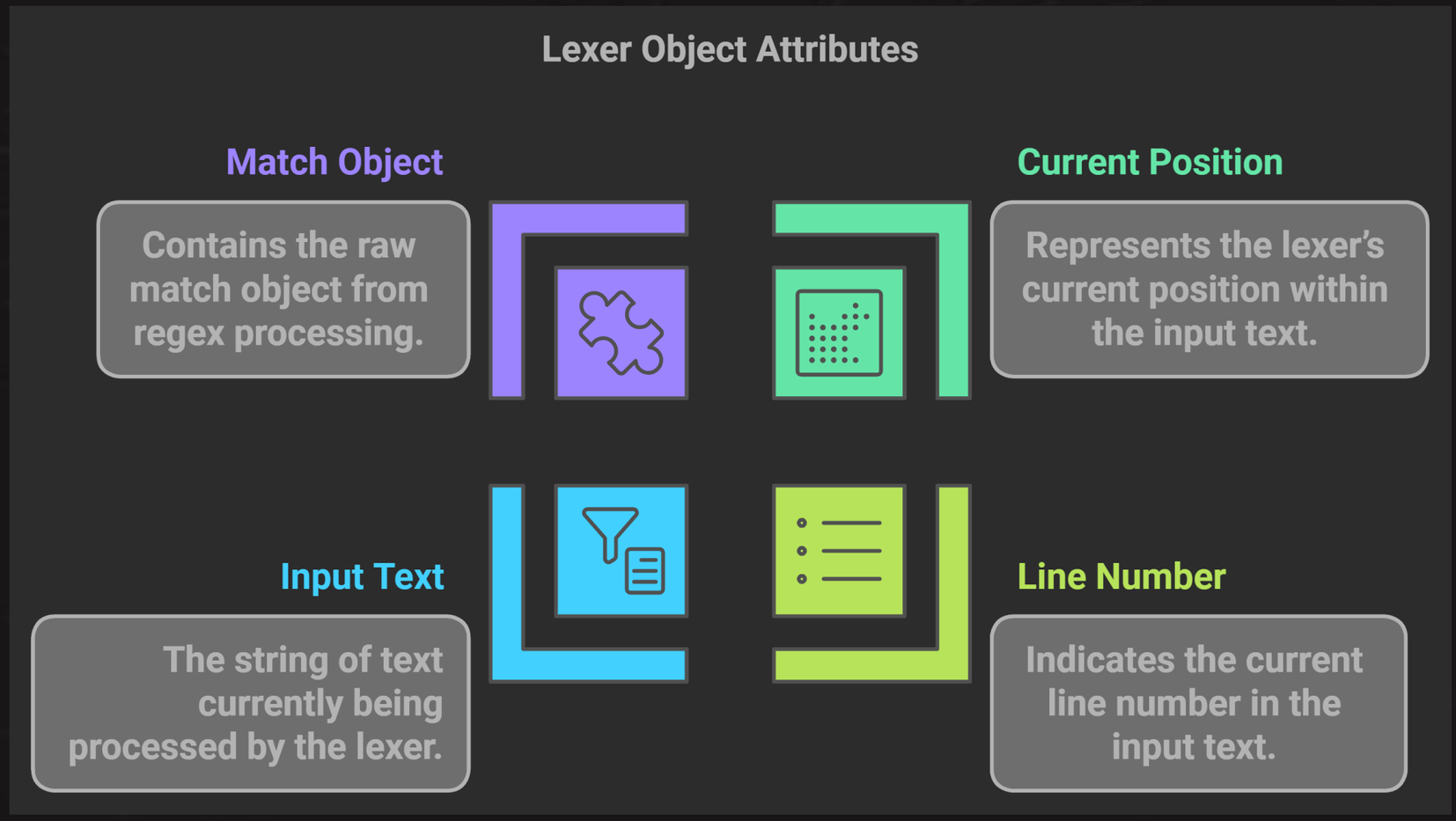
clone.py > ...

```python
28
29
30  lexer.input("3 + 4 * 5")
31
32  # lexer clone
33  lookahead_lexer = lexer.clone()
34
35  # Analyze a portion of text using cloning
36  # Read all numbers using the cloned version
37  count_numbers = 0
38  while True:
39      tok = lookahead_lexer.token()
40      if not tok:
41          break
42      if tok.type == 'NUMBER':
43          count_numbers += 1
44
45  print("Number of numbers in the expression:", count_numbers)
46  if count_numbers % 2 == 1:
47      print("The expression has an odd number of numbers.")
48  else:
49      print("The expression has an even number of numbers.")
50
```

# Internal lexer state

- A Lexer object lexer has a number of internal attributes that may be useful in certain situations.

  - ✓ lexer.lexpos
  - ✓ lexer.lineno
  - ✓ lexer.lexdata
  - ✓ lexer.lexmatch



# Conditional lexing and start conditions

- In advanced parsing, you may encounter situations where you need to deal with multiple grammar rules within the same text. For example, In a script that contains comments or special sections enclosed in parentheses, you need dedicated parsing rules to deal with these sections. To facilitate this, PLY provides the ability to define different parsing cases, so that you can switch between them as needed.

- Parsing cases can be of two types:

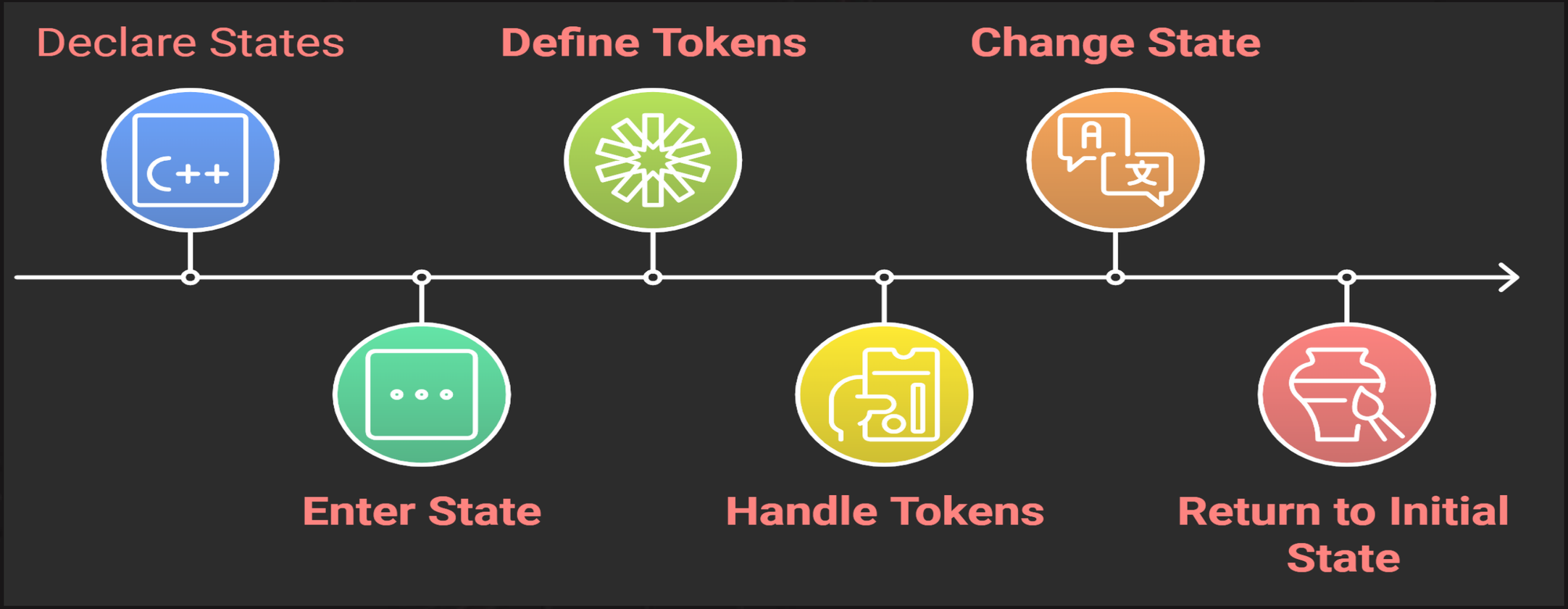
  ✓ Exclusive case: If the case is exclusive, the default parsing rules will be disabled, and parsing will be limited to the rules defined for that case only.

  ✓ Inclusive case: In this case, the rules defined for that case will be combined with the default rules.

# How to use conditional analysis ?

- In order to implement conditional analysis in a symbol parser using the PLY library, states are defined that are switched between based on specific symbols encountered in the text. The default state is **INITIAL**, which is where the parser starts, and can then be switched to other states based on the needs of the analysis.
- **Working steps:**
  - ✓ Define states: Different states are defined in the parser using the states list.
  - ✓ Switch between states: The begin(), push_state(), and pop_state() functions are used to change the state during the analysis.
  - ✓ Return state: You can return to the default state (or any other state) based on the analysis

# Conditional lexing and start conditions



# Thank you

*Any questions?*