

10. 完整的语法规范

这是完整的 Python 语法规范，直接提取自用于生成 CPython 解析器的语法 (参见 [Grammar/python.gram](#))。这里显示的版本省略了有关代码生成和错误恢复的细节。

此处使用的标记法与前述文档一致，其说明详见 [notation](#) 节，但存在一个额外复杂情况：

- ~ ("cut"): 提交到当前替代项并且即使此项解析失败也不适用规则

```
# PEG grammar for Python

# ===== START OF THE GRAMMAR =====

# General grammatical elements and rules:
#
# * Strings with double quotes ("") denote SOFT KEYWORDS
# * Strings with single quotes ('') denote KEYWORDS
# * Upper case names (NAME) denote tokens in the Grammar/Tokens file
# * Rule names starting with "invalid_" are used for specialized syntax errors
#   - These rules are NOT used in the first pass of the parser.
#   - Only if the first pass fails to parse, a second pass including the invalid
#     rules will be executed.
#   - If the parser fails in the second phase with a generic syntax error, the
#     location of the generic failure of the first pass will be used (this avoids
#     reporting incorrect locations due to the invalid rules).
#   - The order of the alternatives involving invalid rules matter
#     (like any rule in PEG).
#
# Grammar Syntax (see PEP 617 for more information):
#
# rule_name: expression
#   Optionally, a type can be included right after the rule name, which
#   specifies the return type of the C or Python function corresponding to the
#   rule:
# rule_name[return_type]: expression
#   If the return type is omitted, then a void * is returned in C and an Any in
#   Python.
# e1 e2
#   Match e1, then match e2.
# e1 | e2
#   Match e1 or e2.
#   The first alternative can also appear on the line after the rule name for
#   formatting purposes. In that case, a | must be used before the first
#   alternative, like so:
#     rule_name[return_type]:
#       | first_alt
#       | second_alt
# ( e )
#   Match e (allows also to use other operators in the group like '(e)*')
# [ e ] or e?
#   Optionally match e.
# e*
```

```

#   Match zero or more occurrences of e.
# e+
#   Match one or more occurrences of e.
# s.e+
#   Match one or more occurrences of e, separated by s. The generated parse tree
#   does not include the separator. This is otherwise identical to (e (s e)*).
# &e
#   Succeed if e can be parsed, without consuming any input.
# !e
#   Fail if e can be parsed, without consuming any input.
# ~
#   Commit to the current alternative, even if it fails to parse.
# &&e
#   Eager parse e. The parser will not backtrack and will immediately
#   fail with SyntaxError if e cannot be parsed.
# 

# STARTING RULES
# =====

file: [statements] ENDMARKER
interactive: statement_newline
eval: expressions NEWLINE* ENDMARKER
func_type: '(' [type_expressions] ')' '->' expression NEWLINE* ENDMARKER

# GENERAL STATEMENTS
# =====

statements: statement+

statement:
| compound_stmt
| simple_stmts

single_compound_stmt:
| compound_stmt

statement_newline:
| single_compound_stmt NEWLINE
| simple_stmts
| NEWLINE
| ENDMARKER

simple_stmts:
| simple_stmt ';' NEWLINE # Not needed, there for speedup
| ';' simple_stmt+ ';' NEWLINE

# NOTE: assignment MUST precede expression, else parsing a simple assignment
# will throw a SyntaxError.
simple_stmt:
| assignment
| type_alias
| star_expressions
| return_stmt
| import_stmt
| raise_stmt
| pass_stmt
| del_stmt
| yield_stmt
| assert_stmt

```

```
    | break_stmt
    | continue_stmt
    | global_stmt
    | nonlocal_stmt

compound_stmt:
    | function_def
    | if_stmt
    | class_def
    | with_stmt
    | for_stmt
    | try_stmt
    | while_stmt
    | match_stmt

# SIMPLE STATEMENTS
# =====

# NOTE: annotated_rhs may start with 'yield'; yield_expr must start with 'yield'
assignment:
    | NAME ':' expression ['=' annotated_rhs ]
    | ('(' single_target ')'
        | single_subscript_attribute_target) ':' expression ['=' annotated_rhs ]
    | (star_targets '=' )+ annotated_rhs '!=' [TYPE_COMMENT]
    | single_target augassign ~ annotated_rhs

annotated_rhs: yield_expr | star_expressions

augassign:
    | '+='
    | '-='
    | '*='
    | '@='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '<<='
    | '>>='
    | '**='
    | '//='

return_stmt:
    | 'return' [star_expressions]

raise_stmt:
    | 'raise' expression ['from' expression ]
    | 'raise'

pass_stmt:
    | 'pass'

break_stmt:
    | 'break'

continue_stmt:
    | 'continue'

global_stmt: 'global' ',' .NAME+
```

```

nonlocal_stmt: 'nonlocal' ' '.NAME+
del_stmt:
| 'del' del_targets &(';' | NEWLINE)

yield_stmt: yield_expr

assert_stmt: 'assert' expression [',' expression ]

import_stmt:
| import_name
| import_from

# Import statements
# =====

import_name: 'import' dotted_as_names
# note below: the ('.' / '...') is necessary because '...' is tokenized as ELLIPSIS
import_from:
| 'from' ('.' | '...')* dotted_name 'import' import_from_targets
| 'from' ('.' | '...')+ 'import' import_from_targets
import_from_targets:
| '(' import_from_as_names [','] ')'
| import_from_as_names ','
| '*'

import_from_as_names:
| ',' .import_from_as_name+
import_from_as_name:
| NAME ['as' NAME]

dotted_as_names:
| ',' .dotted_as_name+
dotted_as_name:
| dotted_name ['as' NAME]

dotted_name:
| dotted_name '.' NAME
| NAME

# COMPOUND STATEMENTS
# =====

# Common elements
# =====

block:
| NEWLINE INDENT statements DEDENT
| simple_stmts

decorators: ('@' named_expression NEWLINE )+

# Class definitions
# =====

class_def:
| decorators class_def_raw
| class_def_raw

class_def_raw:

```

```

| 'class' NAME [type_params] ['(' [arguments] ')' ] '::' block

# Function definitions
# ----

function_def:
| decorators function_def_raw
| function_def_raw

function_def_raw:
| 'def' NAME [type_params] '(' [params] ')' ['->' expression] ':' [func_type_]
| 'async' 'def' NAME [type_params] '(' [params] ')' ['->' expression] ':' [fu

# Function parameters
# ----

params:
| parameters

parameters:
| slash_no_default param_no_default* param_with_default* [star_etc]
| slash_with_default param_with_default* [star_etc]
| param_no_default+ param_with_default* [star_etc]
| param_with_default+ [star_etc]
| star_etc

# Some duplication here because we can't write (', ' / &'),
# which is because we don't support empty alternatives (yet).

slash_no_default:
| param_no_default+ '/' ',' 
| param_no_default+ '/' '&)'
slash_with_default:
| param_no_default* param_with_default+ '/' ',' 
| param_no_default* param_with_default+ '/' '&)' 

star_etc:
| '*' param_no_default param_maybe_default* [kwds]
| '*' param_no_default_star_annotation param_maybe_default* [kwds]
| '*' ',' param_maybe_default+ [kwds]
| kwds

kwds:
| '**' param_no_default

# One parameter. This *includes* a following comma and type comment.
#
# There are three styles:
# - No default
# - With default
# - Maybe with default
#
# There are two alternative forms of each, to deal with type comments:
# - Ends in a comma followed by an optional type comment
# - No comma, optional type comment, must be followed by close paren
# The latter form is for a final parameter without trailing comma.
#
param_no_default:
| param ',' TYPE_COMMENT?

```

```
| param TYPE_COMMENT? &)')
param_no_default_star_annotation:
| param_star_annotation ',' TYPE_COMMENT?
| param_star_annotation TYPE_COMMENT? &)')
param_with_default:
| param default ',' TYPE_COMMENT?
| param default TYPE_COMMENT? &)')
param_maybe_default:
| param default? ',' TYPE_COMMENT?
| param default? TYPE_COMMENT? &)')
param: NAME annotation?
param_star_annotation: NAME star_annotation
annotation: ';' expression
star_annotation: ';' star_expression
default: '=' expression | invalid_default

# If statement
# -------

if_stmt:
| 'if' named_expression ':' block elif_stmt
| 'if' named_expression ':' block [else_block]
elif_stmt:
| 'elif' named_expression ':' block elif_stmt
| 'elif' named_expression ':' block [else_block]
else_block:
| 'else' ':' block

# While statement
# -------

while_stmt:
| 'while' named_expression ':' block [else_block]

# For statement
# -------

for_stmt:
| 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block [else_b
| 'async' 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block

# With statement
# -------

with_stmt:
| 'with' '(' ',' .with_item+ ','? ')' ':' [TYPE_COMMENT] block
| 'with' ',' .with_item+ ':' [TYPE_COMMENT] block
| 'async' 'with' '(' ',' .with_item+ ','? ')' ':' block
| 'async' 'with' ',' .with_item+ ':' [TYPE_COMMENT] block

with_item:
| expression 'as' star_target &(',' | ')') | ':')

# Try statement
# -------

try_stmt:
| 'try' ':' block finally_block
| 'try' ':' block except_block+ [else_block] [finally_block]
```

```

| 'try' ':' block except_star_block+ [else_block] [finally_block]

# Except statement
# -------

except_block:
    | 'except' expression ':' block
    | 'except' expression 'as' NAME ':' block
    | 'except' expressions ':' block
    | 'except' ':' block
except_star_block:
    | 'except' '*' expression ':' block
    | 'except' '*' expression 'as' NAME ':' block
    | 'except' '*' expressions ':' block
finally_block:
    | 'finally' ':' block

# Match statement
# -------

match_stmt:
    | "match" subject_expr ':' NEWLINE INDENT case_block+ DEDENT

subject_expr:
    | star_named_expression ',' star_named_expressions?
    | named_expression

case_block:
    | "case" patterns guard? ':' block

guard: 'if' named_expression

patterns:
    | open_sequence_pattern
    | pattern

pattern:
    | as_pattern
    | or_pattern

as_pattern:
    | or_pattern 'as' pattern_capture_target

or_pattern:
    | '|'.closed_pattern+

closed_pattern:
    | literal_pattern
    | capture_pattern
    | wildcard_pattern
    | value_pattern
    | group_pattern
    | sequence_pattern
    | mapping_pattern
    | class_pattern

# Literal patterns are used for equality and identity constraints
literal_pattern:
    | signed_number !( '+' | '-' )

```

```

    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'

# Literal expressions are used to restrict permitted mapping pattern keys
literal_expr:
    | signed_number !( '+' | '-' )
    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'

complex_number:
    | signed_real_number '+' imaginary_number
    | signed_real_number '-' imaginary_number

signed_number:
    | NUMBER
    | '-' NUMBER

signed_real_number:
    | real_number
    | '-' real_number

real_number:
    | NUMBER

imaginary_number:
    | NUMBER

capture_pattern:
    | pattern_capture_target

pattern_capture_target:
    | !'"' NAME !( '.' | '(' | '=' )

wildcard_pattern:
    | " _"

value_pattern:
    | attr !( '.' | '(' | '=' )

attr:
    | name_or_attr '.' NAME

name_or_attr:
    | attr
    | NAME

group_pattern:
    | '(' pattern ')'

sequence_pattern:
    | '[' maybe_sequence_pattern? ']'
    | '(' open_sequence_pattern? ')'

open_sequence_pattern:

```

```

    | maybe_star_pattern ',' maybe_sequence_pattern?

maybe_sequence_pattern:
    | ',' . maybe_star_pattern+ ',' ?

maybe_star_pattern:
    | star_pattern
    | pattern

star_pattern:
    | '*' pattern_capture_target
    | '*' wildcard_pattern

mapping_pattern:
    | '{' '}'
    | '{' double_star_pattern ',' '?' '}'
    | '{' items_pattern ',' double_star_pattern ',' '?' '}'
    | '{' items_pattern ',' '?' '}'

items_pattern:
    | ',' . key_value_pattern+

key_value_pattern:
    | (literal_expr | attr) ':' pattern

double_star_pattern:
    | '**' pattern_capture_target

class_pattern:
    | name_or_attr '(' ')'
    | name_or_attr '(' positional_patterns ',' '?' ')'
    | name_or_attr '(' keyword_patterns ',' '?' ')'
    | name_or_attr '(' positional_patterns ',' keyword_patterns ',' '?' ')'

positional_patterns:
    | ',' . pattern+

keyword_patterns:
    | ',' . keyword_pattern+

keyword_pattern:
    | NAME '=' pattern

# Type statement
# -------

type_alias:
    | "type" NAME [type_params] '=' expression

# Type parameter declaration
# -------

type_params:
    | '[' type_param_seq ']'

type_param_seq: ',' . type_param+ [',']

type_param:
    | NAME [type_param_bound] [type_param_default]
    | '*' NAME [type_param_starred_default]

```

```
| '**' NAME [type_param_default]

type_param_bound: ':' expression
type_param_default: '=' expression
type_param_starred_default: '=' star_expression

# EXPRESSIONS
# -------

expressions:
| expression (',' expression )+ [',']

expression:
| disjunction
| conjunction
| lambdef

yield_expr:
| 'yield' 'from' expression
| 'yield' [star_expressions]

star_expressions:
| star_expression (',' star_expression )+ [',']

star_expression:
| '*' bitwise_or
| expression

star_named_expressions: ',' . star_named_expression+ [',']

star_named_expression:
| '*' bitwise_or
| named_expression

assignment_expression:
| NAME ':=' ~ expression

named_expression:
| assignment_expression
| expression '!':='

disjunction:
| conjunction ('or' conjunction )+
| conjunction

conjunction:
| inversion ('and' inversion )+
| inversion

inversion:
| 'not' inversion
| comparison

# Comparison operators
# -----
```

```

comparison:
| bitwise_or compare_op_bitwise_or_pair+
| bitwise_or

compare_op_bitwise_or_pair:
| eq_bitwise_or
| noteq_bitwise_or
| lte_bitwise_or
| lt_bitwise_or
| gte_bitwise_or
| gt_bitwise_or
| notin_bitwise_or
| in_bitwise_or
| isnot_bitwise_or
| is_bitwise_or

eq_bitwise_or: '==' bitwise_or
noteq_bitwise_or:
| ('!=' ) bitwise_or
lte_bitwise_or: '<=' bitwise_or
lt_bitwise_or: '<' bitwise_or
gte_bitwise_or: '>=' bitwise_or
gt_bitwise_or: '>' bitwise_or
notin_bitwise_or: 'not' 'in' bitwise_or
in_bitwise_or: 'in' bitwise_or
isnot_bitwise_or: 'is' 'not' bitwise_or
is_bitwise_or: 'is' bitwise_or

# Bitwise operators
# -------

bitwise_or:
| bitwise_or '||' bitwise_xor
| bitwise_xor

bitwise_xor:
| bitwise_xor '^' bitwise_and
| bitwise_and

bitwise_and:
| bitwise_and '&' shift_expr
| shift_expr

shift_expr:
| shift_expr '<<' sum
| shift_expr '>>' sum
| sum

# Arithmetic operators
# -------

sum:
| sum '+' term
| sum '-' term
| term

term:
| term '*' factor
| term '/' factor
| term '//' factor

```

```

    | term '%' factor
    | term '@' factor
    | factor

factor:
    | '+' factor
    | '-' factor
    | '~' factor
    | power

power:
    | await_primary '**' factor
    | await_primary

# Primary elements
# -----

# Primary elements are things like "obj.something.something", "obj[something]", "o

await_primary:
    | 'await' primary
    | primary

primary:
    | primary '.' NAME
    | primary genexp
    | primary '(' [arguments] ')'
    | primary '[' slices ']'
    | atom

slices:
    | slice ','
    | ','.(slice | starred_expression)+ [',']

slice:
    | [expression] ':' [expression] [':' [expression] ]
    | named_expression

atom:
    | NAME
    | 'True'
    | 'False'
    | 'None'
    | strings
    | NUMBER
    | (tuple | group | genexp)
    | (list | listcomp)
    | (dict | set | dictcomp | setcomp)
    | '...'

group:
    | '(' (yield_expr | named_expression) ')'

# Lambda functions
# -----

lambdef:
    | 'lambda' [lambda_params] ':' expression

lambda_params:

```

```

| lambda_parameters

# Lambda_parameters etc. duplicates parameters but without annotations
# or type comments, and if there's no comma after a parameter, we expect
# a colon, not a close parenthesis. (For more, see parameters above.)
#
lambda_parameters:
| lambda_slash_no_default lambda_param_no_default* lambda_param_with_default*
| lambda_slash_with_default lambda_param_with_default* [lambda_star_etc]
| lambda_param_no_default+ lambda_param_with_default* [lambda_star_etc]
| lambda_param_with_default+ [lambda_star_etc]
| lambda_star_etc

lambda_slash_no_default:
| lambda_param_no_default+ '/' ','
| lambda_param_no_default+ '/' &':'

lambda_slash_with_default:
| lambda_param_no_default* lambda_param_with_default+ '/' ','
| lambda_param_no_default* lambda_param_with_default+ '/' &':'

lambda_star_etc:
| '*' lambda_param_no_default lambda_param_maybe_default* [lambda_kwds]
| '*' ',' lambda_param_maybe_default+ [lambda_kwds]
| lambda_kwds

lambda_kwds:
| '**' lambda_param_no_default

lambda_param_no_default:
| lambda_param ','
| lambda_param &':'

lambda_param_with_default:
| lambda_param default ','
| lambda_param default &':'

lambda_param_maybe_default:
| lambda_param default? ','
| lambda_param default? &':'

lambda_param: NAME

# LITERALS
# =====

fstring_middle:
| fstring_replacement_field
| FSTRING_MIDDLE

fstring_replacement_field:
| '{' annotated_rhs '='? [fstring_conversion] [fstring_full_format_spec] '}'

fstring_conversion:
| "!" NAME

fstring_full_format_spec:
| ':' fstring_format_spec*

fstring_format_spec:
| FSTRING_MIDDLE
| fstring_replacement_field

fstring:
| FSTRING_START fstring_middle* FSTRING_END

tstring_format_spec_replacement_field:
| '{' annotated_rhs '='? [fstring_conversion] [tstring_full_format_spec] '}'

```

```

tstring_format_spec:
| TSTRING_MIDDLE
| tstring_format_spec_replacement_field

tstring_full_format_spec:
| ':' tstring_format_spec*
tstring_replacement_field:
| '{' annotated_rhs '='? [fstring_conversion] [tstring_full_format_spec] '}'

tstring_middle:
| tstring_replacement_field
| TSTRING_MIDDLE

tstring:
| TSTRING_START tstring_middle* TSTRING_END

string: STRING
strings:
| (fstring|string)+

list:
| '[' [star_named_expressions] ']'

tuple:
| '(' [star_named_expression ',' [star_named_expressions]] ')'

set: '{' star_named_expressions '}'

# Dicts
# -----

dict:
| '{' [double_starred_kvpairs] '}' 

double_starred_kvpairs: ',' .double_starred_kvpair+ [',']

double_starred_kvpair:
| '**' bitwise_or
| kvpair

kvpair: expression ':' expression

# Comprehensions & Generators
# -----

for_if_clauses:
| for_if_clause+

for_if_clause:
| 'async' 'for' star_targets 'in' ~ disjunction ('if' disjunction )*
| 'for' star_targets 'in' ~ disjunction ('if' disjunction )*

listcomp:
| '[' named_expression for_if_clauses ']'

setcomp:
| '{' named_expression for_if_clauses '}' 

genexp:
| '(' ( assignment_expression | expression '!:=') for_if_clauses ')'

dictcomp:

```

```

| '{' kvpair for_if_clauses '}'

# FUNCTION CALL ARGUMENTS
# =====

arguments:
| args [','] &')

args:
| ','.(starred_expression | ( assignment_expression | expression ':=') !'=')+

kwargs:
| ','.kwarg_or_starred+ ',' ','.kwarg_or_double_starred+
| ','.kwarg_or_starred+
| ','.kwarg_or_double_starred+

starred_expression:
| '*' expression

kwarg_or_starred:
| NAME '=' expression
| starred_expression

kwarg_or_double_starred:
| NAME '=' expression
| '**' expression

# ASSIGNMENT TARGETS
# =====

# Generic targets
# ----

# NOTE: star_targets may contain *bitwise_or, targets may not.
star_targets:
| star_target ','
| star_target (',' star_target )* [,]

star_targets_list_seq: ','.star_target+ [,]

star_targets_tuple_seq:
| star_target (',' star_target )+ [,]
| star_target ','

star_target:
| '*' (!'*' star_target)
| target_with_star_atom

target_with_star_atom:
| t_primary '.' NAME !t_lookahead
| t_primary '[' slices ']' !t_lookahead
| star_atom

star_atom:
| NAME
| '(' target_with_star_atom ')'
| '(' [star_targets_tuple_seq] ')'
| '[' [star_targets_list_seq] ']'

```

```

single_target:
| single_subscript_attribute_target
| NAME
| '(' single_target ')'

single_subscript_attribute_target:
| t_primary '..' NAME !t_lookahead
| t_primary '[' 'slices ']' !t_lookahead

t_primary:
| t_primary '..' NAME &t_lookahead
| t_primary '[' 'slices ']' &t_lookahead
| t_primary genexp &t_lookahead
| t_primary '(' '[arguments]' ')' &t_lookahead
| atom &t_lookahead

t_lookahead: '(' | '[' | '..'

# Targets for del statements
# -----

del_targets: ',' .del_target+ [',']

del_target:
| t_primary '..' NAME !t_lookahead
| t_primary '[' 'slices ']' !t_lookahead
| del_t_atom

del_t_atom:
| NAME
| '(' del_target ')'
| '(' '[del_targets]' ')'
| '[' '[del_targets]' ']'

# TYPING ELEMENTS
# -----

# type_expressions allow */** but ignore them
type_expressions:
| ',' .expression+ ',' '*' expression ',' '**' expression
| ',' .expression+ ',' '*' expression
| ',' .expression+ ',' '**' expression
| '*' expression ',' '**' expression
| '*' expression
| '**' expression
| ',' .expression+

func_type_comment:
| NEWLINE TYPE_COMMENT &(NEWLINE INDENT)    # Must be followed by indented block
| TYPE_COMMENT

# ===== END OF THE GRAMMAR =====

# ===== START OF INVALID RULES =====

```