2. 词法分析

A Python program is read by a *parser*. Input to the parser is a stream of tokens, generated by the *lexical analyzer* (also known as the *tokenizer*). This chapter describes how the lexical analyzer produces these tokens.

The lexical analyzer determines the program text's encoding (UTF-8 by default), and decodes the text into source characters. If the text cannot be decoded, a `SyntaxError` is raised.

Next, the lexical analyzer uses the source characters to generate a stream of tokens. The type of a generated token generally depends on the next source character to be processed. Similarly, other special behavior of the analyzer depends on the first source character that hasn't yet been processed. The following table gives a quick summary of these source characters, with links to sections that contain more information.

字符	下一词元（或其他相关文档）
• space • tab • formfeed	• 空格
• CR, LF	• 换行 • 缩进
• backslash (`\`)	• Explicit line joining • (Also significant in string escape sequences)
• hash (`#`)	• Comment
• quote (`'`, `"`)	• String literal
• ASCII letter (`a`-`z`, `A`-`Z`) • non-ASCII character	• Name • Prefixed string or bytes literal
• underscore (`_`)	• Name • (Can also be part of numeric literals)
• number (`0`-`9`)	• Numeric literal
• dot (`.`)	• Numeric literal • Operator
• question mark (`?`) • dollar (`$`) • backquote (`` ` ``) • control character	• Error (outside string literals and comments)

字符	下一词元（或其他相关文档）
• other printing character	• Operator or delimiter
• end of file	• End marker

2.1. 行结构

Python 程序可以拆分为多个 *逻辑行*。

2.1.1. 逻辑行

逻辑行的结束由词元 `NEWLINE` 表示。 除非语法允许 `NEWLINE` (举例来说，在复合语句中的多个语句之前) 否则语句不能跨越逻辑行边界。 逻辑行由一条或多条 *物理行* 根据 显示 或 隐式 的 *行连接* 规则构造而成。

2.1.2. 物理行

物理行是由使用下列行结束符序列中的一个作为终结的字符序列：

- 使用 ASCII LF (linefeed) 的 Unix 形式，
- 使用 ASCII 序列 CR LF (return 加 linefeed) 的 Windows 形式，
- 使用 ASCII CR (return) 字符的 'Classic Mac OS' 形式。

无论平台如何，这些序列中的每一个都会被替换为单个ASCII LF（换行）字符。（即使在 字符串字面量 中也是如此。）每一行可以使用这些序列中的任何一个；它们在文件内部不需要保持一致。

输入的结束也作为最后一个物理行的隐式终止符。

形式上：

```
newline: <ASCII LF> | <ASCII CR> <ASCII LF> | <ASCII CR>
```

2.1.3. 注释

注释以井号 （`#`） 开头，在物理行末尾截止。注意，井号不是字符串字面量。除非应用隐式行拼接规则，否则，注释代表逻辑行结束。句法不解析注释。

2.1.4. 编码声明

Python 脚本第一或第二行的注释匹配正则表达式 `coding[=:]\s*([-\w.]+)` 时，该注释会被当作编码声明；这个表达式的第一组指定了源码文件的编码。编码声明必须独占一行，在第二行时，则第一行必须也是注释。编码表达式的形式如下：

```
# -*- coding: <encoding-name> -*-
```

这也是 GNU Emacs 认可的形式，此外，还支持如下形式：

```
# vim:fileencoding=<encoding-name>
```

这是 Bram Moolenaar 的 VIM 认可的形式。

如果没有找到编码格式声明，则默认编码格式为 UTF-8。 如果文件的隐式或显式编码格式为 UTF-8，则初始的 UTF-8 字节顺序标记（`b'\xef\xbb\xbf'`）会被忽略，而不是成为语法错误。

如果声明了编码格式，该编码格式的名称必须是 Python 可识别的 (参见 标准编码)。 编码格式会被用于所有的词法分析，包括字符串字面量、注释和标识符等。

所有词法分析，包括字符串字面量、注释和标识符，都作用于使用源编码解码的Unicode文本。除了 NUL 控制字符以外的任何Unicode码点都可以出现在Python源代码中。

```
source_character:  <any Unicode code point, except NUL>
```

2.1.5. 显式拼接行

两个及两个以上的物理行可用反斜杠（`\`）拼接为一个逻辑行，规则如下：以不在字符串或注释内的反斜杠结尾时，物理行将与下一行拼接成一个逻辑行，并删除反斜杠及其后的换行符。例如：

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # 看来是个有效的日期
        return 1
```

以反斜杠结尾的行不能包含注释。反斜杠不能用于延续注释内容。除字符串字面量之外，反斜杠不能用于延续标记（即非字符串字面量的标记不能通过反斜杠拆分到物理行的下一行）。在字符串字面量之外的行中，反斜杠出现在其他位置是非法的。

2.1.6. 隐式拼接行

圆括号、方括号、花括号内的表达式可以分成多个物理行，不必使用反斜杠。例如：

```
month_names = ['Januari', 'Februari', 'Maart',      # 这些是
               'April',   'Mei',      'Juni',       # 一年之中
               'Juli',    'Augustus', 'September',  # 各个月份的
               'Oktober', 'November', 'December']   # 荷兰语名称
```

隐式行拼接可含注释；后续行的缩进并不重要；还支持空的后续行。隐式拼接行之间没有 NEWLINE 标记。三引号字符串支持隐式拼接行（见下文），但不支持注释。

2.1.7. 空白行

仅包含空格、制表符、换页符以及可能的注释的逻辑行将被忽略（即不会生成 NEWLINE 标记）。在交互式输入语句时，空行的处理方式可能因读取-求值-打印循环的实现而异。在标准交互式解释器中，完全空白的逻辑行（即不包含任何空白符或注释）会终止多行语句。

2.1.8. 缩进

逻辑行开头的空白符（空格符和制表符）用于计算该行的缩进层级，决定语句组块。

制表符（从左至右）被替换为一至八个空格，缩进空格的总数是八的倍数（与 Unix 的规则保持一致）。首个非空字符前的空格数决定了该行的缩进层次。缩进不能用反斜杠进行多行拼接；首个反斜杠之前的空白符决定了缩进的层次。

源文件混用制表符和空格符缩进时，因空格数量与制表符相关，由此产生的不一致将导致不能正常识别缩进层次，从而触发 `TabError`。

跨平台兼容性说明： 鉴于非 UNIX 平台文本编辑器本身的特性，请勿在源文件中混用制表符和空格符。另外也请注意，不同平台有可能会显式限制最大缩进层级。

行首含换页符时，缩进计算将忽略该换页符。换页符在行首空白符内其他位置的效果未定义（例如，可能导致空格计数重置为零）。

连续行的缩进级别使用一个栈来生成 `INDENT` 和 `DEDENT` 标记，规则如下。

Before the first line of the file is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one `INDENT` token is generated. If it is smaller, it *must* be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a `DEDENT` token is generated. At the end of the file, a `DEDENT` token is generated for each number remaining on the stack that is larger than zero.

下面的 Python 代码缩进示例虽然正确，但含混不清：

```
def perm(l):
        # 计算由 l 的所有排列组成的列表
    if len(l) <= 1:
                  return [l]
    r = []
    for i in range(len(l)):
             s = l[:i] + l[i+1:]
             p = perm(s)
             for x in p:
              r.append(l[i:i+1] + x)
    return r
```

下例展示了多种缩进错误：

```
 def perm(l):                        # 错误：第一行有缩进
for i in range(len(l)):              # 错误：没有缩进
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # 错误：非预期的缩进
        for x in p:
                r.append(l[i:i+1] + x)
            return r                 # 错误：不一致的缩进
```

（实际上，解析器可以识别前三个错误；只有最后一个错误由词法分析器识别 --- `return r` 的缩进无法匹配从栈里移除的缩进层级。）

2.1.9. 标记间的空白字符

除了在逻辑行的开头或字符串字面量中，空白符（空格、制表符和换页符）可以互换使用以分隔标记。仅当两个标记的拼接可能被解释为不同的标记时，才需要在它们之间使用空白符。例如，`ab` 是一个标记，而 `a b` 是两个标记。然而，`+a` 和 `+ a` 都产生两个标记，即 `+` 和 `a`，因为 `+a` 不是有效的标记。

2.1.10. 结束标记

在非交互输入结束时，词法分析器将生成一个 `ENDMARKER` 词元。

2.2. 其他标记

除了 `NEWLINE`、`INDENT` 和 `DEDENT` 之外，还存在以下几类词元: *标识符* 和 *关键字* (`NAME`)、*字面量* (如 `NUMBER` 和 `STRING`)，以及其他符号 (*运算符* 和 *分隔符*, `OP`)。 空白字符（除了前面讨论的逻辑行终止符）不是词元，而是用于分隔词元。 在有歧义的情况下，词元由从左到右读取时能形成合法词元的最长可能字符串组成。

2.3. 名称（标识符和关键字）

`NAME` 标记表示 *标识符*、*关键字* 和 *软关键字*。

在 ASCII 范围（U+0001..U+007F）内，名称的有效字符包括大小写字母（`A-Z` 和 `a-z`）、下划线 `_`，并且除了首字符外，还可以包含数字 `0` 到 `9`。

名称必须至少包含一个字符，但没有长度上限。大小写敏感。

除了 `A-Z`、`a-z`、`_` 和 `0-9` 之外，名称还可以使用 ASCII 范围之外的"类字母"和"类数字"字符，具体如下所述。

所有标识符在解析时都会转换为 规范化形式 NFKC；标识符的比较基于 NFKC。

形式上，规范化标识符的首字符必须属于集合 `id_start`，该集合是以下各项的并集：

- Unicode 类别 `<Lu>` - 大写字母 (包括 `A` 到 `Z`)
- Unicode 类别 `<Ll>` - 小写字母 (包括 `a` 到 `z`)
- Unicode 类别 `<Lt>` - 标题大小写字母
- Unicode 类别 `<Lm>` - 修饰字母
- Unicode category `<Lo>` - 其他字母
- Unicode 类别 `<Nl>` - 数字字母
- {`"_"`} - 下划线
- `<Other_ID_Start>` - 在 PropList.txt 中显式定义的用于支持向下兼容的字符集合

其余字母必须归属于 `id_continue` 集合，它是以下字符的并集：

- `id_start` 中的所有字符
- Unicode 类别 `<Nd>` - 十进制数字（包括 `0` 到 `9`）
- Unicode 类别 `<Pc>` - 连接标点符号
- Unicode 类别 `<Mn>` - 非间距标记
- Unicode 类别 `<Mc>` - 间距组合标记
- `<Other_ID_Continue>` - 为支持向下兼容性在 PropList.txt 中另一组显式列出的字符集合

Unicode 类别使用的是 `unicodedata` 模块中所包含的 Unicode 字符数据库版本。

这些集合基于 Unicode 标准附录 UAX-31。有关更多详细信息，另请参阅 **PEP 3131**。

更形式化地说，名称由以下词法定义描述：

```
NAME:          xid_start xid_continue*
id_start:      <Lu> | <Ll> | <Lt> | <Lm> | <Lo> | <Nl> | "_" | <Other_ID_Start>
id_continue:   id_start | <Nd> | <Pc> | <Mn> | <Mc> | <Other_ID_Continue>
xid_start:     <all characters in id_start whose NFKC normalization is
                in (id_start xid_continue*)">
xid_continue:  <all characters in id_continue whose NFKC normalization is
                in (id_continue*)">
identifier:    <NAME, except keywords>
```

Unicode 字符数据库中的 DerivedCoreProperties.txt 文件提供了一份非规范性的列表，包含了所有符合 Unicode 定义的有效标识符字符。

2.3.1. 关键字

以下名称被用作该语言的保留字或 *关键字*，不能用作普通标识符。它们的拼写必须与此处完全一致：

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

2.3.2. 软关键字

> *Added in version 3.10.*

某些名称仅在特定上下文中保留。这些被称为 *软关键字*:

- `match`、`case` 和 `_` 在 `match` 语句中使用时属于软关键字。
- `type` 在 `type` 语句中使用时属于软关键字。

这些标识符在特定上下文中语法上作为关键字使用，但这种区分是在解析器层面完成的，而非词法分析（分词）阶段。

作为软关键字，它们能够在用于相应语法的同时仍然保持与用作标识符名称的现有代码的兼容性。

> *在 3.12 版本发生变更:* `type` 现在是一个软关键字。

2.3.3. 保留的标识符类

某些标识符类（除了关键字）具有特殊含义。这些类的命名模式以下划线字符开头，并以下划线结尾：

`_*`

不会被 `from module import *` 所导入。

`_`

在 `match` 语句内部的 `case` 模式中，`_` 是一个 软关键字，它表示 通配符。

在此之外，交互式解释器会将最后一次求值的结果放到变量 `_` 中。（它与 `print` 等内置函数一起被存储于 `builtins` 模块。）

在其他地方，`_` 是一个常规标识符。它常常被用来命名 "特殊" 条目，但对 Python 本身来说毫无特殊之处。

> **备注:** `_` 常用于连接国际化文本；详见 `gettext` 模块文档。
>
> 它还经常被用来命名无需使用的变量。

`__*__`

系统定义的名称，通常简称为 "dunder" 。这些名称由解释器及其实现（包括标准库）定义。现有系统定义名称相关的论述详见 特殊方法名称 等章节。Python 未来版本中还将定义更多此类名称。任何情况下，*任何* 不显式遵从 `__*__` 名称的文档用法，都可能导致无警告提示的错误。

`__*`

类的私有名称。类定义时，此类名称以一种混合形式重写，以避免基类及派生类的 "私有" 属性之间产生名称冲突。详见 标识符（名称）。

2.4. 字面量

字面量是内置类型常量值的表示法。

在词法分析方面，Python有 字符串、字节 和 数值 字面量。

其他"字面量"通过 关键字 (`None`、`True`、`False`) 和特殊的 省略号标记 (`...`) 进行词法表示。

2.5. 字符串与字节串字面量

字符串字面量是用单引号（`'`）或双引号（`"`）括起来的文本。例如：

```
"spam"
'eggs'
```

用于开始字面量的引号也用于终止它，因此字符串字面量只能包含另一个引号（除非使用转义序列，见下文）。例如：

```
'Say "Hello", please.'
"Don't do that!"
```

除了这个限制外，引号字符（`'` 或 `"`）的选择不影响字面量的解析方式。

在字符串字面量内部，反斜杠（`\`）字符引入了一个 *转义序列*，其特殊含义取决于反斜杠后面的字符。例如，`\"` 表示双引号字符，并且 *不* 结束字符串：

```
>>> print("Say \"Hello\" to everyone!")
Say "Hello" to everyone!
```

有关此类序列的完整列表和更多详细信息，请参见下文的 转义序列。

2.5.1. 三引号字符串

字符串也可以用三组匹配的单引号或双引号括起来。这些通常被称为 *三引号字符串*:

```
"""这是一个三引号字符串。"""
```

在三引号字面量中，允许使用未转义的引号（并且会保留），但与起始引号相同的三个连续未转义引号（`'` 或 `"`）会终止字面量:

```
"""这个字符串内有 "引号"。"""
```

也允许使用未转义的新行，并且会保留:

```
'''这个三引号字符串
在下一行继续。'''
```

2.5.2. 字符串前缀

字符串字面量可以有一个可选的 *前缀*，该前缀会影响字面量内容的解析方式，例如:

```
b"data"
f'{result=}'
```

允许的前缀有:

- `b`: 字节串字面量
- `r`: 原始字符串
- `f`: 格式字符串字面量 ("f-string")
- `t`: 模板字符串字面量 ("t-string")
- `u`: 无效果（为向后兼容而允许）

详细信息请参见链接部分。

前缀不区分大小写（例如，`'B'` 与 `'b'` 效果相同）。 `'r'` 前缀可以与 `'f'`, `'t'` 或 `'b'` 组合使用，因此 `'fr'`, `'rf'`, `'tr'`, `'rt'`, `'br'` 和 `'rb'` 也是有效的前缀。

> *Added in version 3.3:* 新增原始字节串 `'rb'` 前缀，是 `'br'` 的同义词。
>
> 为简化 Python 2.x 和 3.x 双版本代码库的维护工作，重新引入了对 Unicode 传统字面量（`u'value'`）的支持。更多信息请参阅 **PEP 414**。

2.5.3. 正式语法

除了 "f-字符串" 和 "t-字符串" 之外，字符串字面量由以下词法定义描述。

这些定义使用 负向否定前瞻 (`!`) 来指示结束引号会终止字面量。

```
STRING:          [stringprefix] (stringcontent)
stringprefix:    <("r" | "u" | "b" | "br" | "rb"), case-insensitive>
stringcontent:
   | "'''" ( !"'''" longstringitem)* "'''"
   | '"""' ( !'"""' longstringitem)* '"""'
   | "'" ( !"'" stringitem)* "'"
   | '"' ( !'"' stringitem)* '"'
stringitem:       stringchar | stringescapeseq
stringchar:       <any source_character, except backslash and newline>
longstringitem:   stringitem | newline
stringescapeseq: "\" <any source_character>
```

请注意，与所有词法定义一样，空白符是重要的。特别是，前缀（如果有）必须紧接起始引号。

2.5.4. 转义序列

除非存在 '`r`' 或 '`R`' 前缀，在字符串和字节串字面值中的转义序列将按照类似于标准 C 所使用的规则进行解读。 可用的转义序列有：

转义序列	含意
`\`<newline>	字符串转义 - 忽略模式
`\\`	反斜杠
`\'`	单引号
`\"`	双引号
`\a`	ASCII 响铃（BEL）
`\b`	ASCII 退格符（BS）
`\f`	ASCII 换页符（FF）
`\n`	ASCII 换行符（LF）
`\r`	ASCII 回车符（CR）
`\t`	ASCII 水平制表符（TAB）
`\v`	ASCII 垂直制表符（VT）
`\`*ooo*	字符串转义 - 八进制

转义序列	含意
`\xhh`	字符串转义 - 十六进制
`\N{name}`	字符串转义 - 命名字符
`\uxxxx`	十六进制 Unicode 字符
`\Uxxxxxxxx`	十六进制 Unicode 字符

2.5.4.1. 忽略行尾

可以在行尾添加一个反斜杠来忽略换行符:

```
>>> 'This string will not include \
... backslashes or newline characters.'
'This string will not include backslashes or newline characters.'
```

同样的效果也可以使用 三重引号字符串，或者圆括号和 字符串字面量拼接 来达成。

2.5.4.2. 转义字符

要在非 原始 Python 字符串字面量中包含反斜杠，必须将其加倍。`\\` 转义序列表示单个反斜杠字符:

```
>>> print('C:\\Program Files')
C:\Program Files
```

同样，序列 `\'` 和 `\"` 分别表示单引号和双引号字符:

```
>>> print('\' and \"')
' and "
```

2.5.4.3. 八进制字符

序列 `\ooo` 表示一个八进制（基数为8）值为 *ooo* 的*字符*:

```
>>> '\120'
'P'
```

最多接受三个八进制数字（0到7）。

在字节串字面量中，*字符* 表示具有给定值的 *字节*。在字符串字面量中，它表示具有给定值的Unicode字符。

> *在 3.11 版本发生变更:* 值大于 `0o377` (255) 的八进制转义会产生一个 `DeprecationWarning`。

> *在 3.12 版本发生变更:* 值大于 `0o377` (255) 的八进制转义会产生一个 `SyntaxWarning`。在未来的Python版本中，它们将引发一个 `SyntaxError`。

2.5.4.4. 十六进制字符

序列 `\xhh` 表示一个十六进制（基数为16）值为 *hh* 的*字符*:

```
>>> '\x50'
'P'
```

与 C 标准不同，必须为两个十六进制数字。

在字节串字面量中，*字符* 表示具有给定值的 *字节*。在字符串字面量中，它表示具有给定值的Unicode字符。

2.5.4.5. 命名Unicode字符

序列 `\N{name}` 表示具有给定 *name* 的Unicode字符:

```
>>> '\N{LATIN CAPITAL LETTER P}'
'P'
>>> '\N{SNAKE}'
'🐍'
```

此序列不能出现在 字节串字面量 中。

> *在 3.3 版本发生变更:* 已添加对 名称别名 的支持。

2.5.4.6. 十六进制Unicode字符

这些序列 `\uxxxx` 和 `\Uxxxxxxxx` 表示具有给定十六进制（基数为16）值的Unicode字符。`\u` 需要正好四个数字；`\U` 需要正好八个数字。后者可以编码任何Unicode字符。

```
>>> '\u1234'
'ሴ'
>>> '\U0001f40d'
'🐍'
```

这些序列不能出现在 字节串字面量 中。

2.5.4.7. 未识别的转义序列

与标准C不同，所有未识别的转义序列在字符串中保持不变，即 *反斜杠保留在结果中*:

```
>>> print('\q')
\q
>>> list('\q')
['\\', 'q']
```

请注意，对于字节串字面量，仅在字符串字面量中识别的转义序列（`\N...`，`\u...`，`\U...`）属于未识别的转义类别。

> *在 3.6 版本发生变更:* 未识别的转义序列会产生 `DeprecationWarning`。

> *在 3.12 版本发生变更:* 未识别的转义序列会产生一个 `SyntaxWarning`。在未来的Python版本中，它们将引发一个 `SyntaxError`。

2.5.5. 字节串字面量

字节串字面值 总是带有 '`b`' 或 '`B`' 前缀；它们会产生 `bytes` 类型而不是 `str` 类型的实例。 它们只能包含 ASCII 字符；数值为 128 或以上的字节必须使用转义序列来表示 (通常为 十六进制字符 或 八进制字符)：

```
>>> b'\x89PNG\r\n\x1a\n'
b'\x89PNG\r\n\x1a\n'
>>> list(b'\x89PNG\r\n\x1a\n')
[137, 80, 78, 71, 13, 10, 26, 10]
```

同样，零字节必须使用转义序列表示（通常是 `\0` 或 `\x00`）。

2.5.6. 原始字符串字面量

字符串和字节串字面值都可以选择带有字符 '`r`' 或 '`R`' 作为前缀；这样的构造分别称为 *原始字符串字面值* 和 *原始字节串字面值* 并会将反斜杠视为字面字符。 因此，在原始字符串字面值中，转义序列 不会被特殊对待：are not treated specially:

```
>>> r'\d{4}-\d{2}-\d{2}'
'\\d{4}-\\d{2}-\\d{2}'
```

即使在原始字面量中，引号也可以用反斜杠转义，但反斜杠会保留在输出结果里；例如 `r"\""` 是由两个字符组成的有效字符串字面量：反斜杠和双引号；`r"\"` 则不是有效字符串字面量（原始字符串也不能以奇数个反斜杠结尾）。尤其是，*原始字面量不能以单个反斜杠结尾*（反斜杠会转义其后的引号)。还要注意，反斜杠加换行在字面量中被解释为两个字符，而 *不是* 连续行。

2.5.7. f-字符串

> *Added in version 3.6.*

一个 *格式字符串字面值* 或 *f-字符串* 是一个以 '`f`' 或 '`F`' 为前缀的字符串字面值。这些字符串可以包含替换字段，这些字段是用花括号 `{}` 分隔的表达式。尽管其他字符串字面量总是具有常量值，格式字符串实际上是运行时评估的表达式。

除非字面量标记为原始字符串，否则，与在普通字符串字面量中一样，转义序列也会被解码。解码后，用于字符串内容的语法如下：

```
f_string:          (literal_char | "{{" | "}}" | replacement_field)*
replacement_field: "{" f_expression ["="] ["!" conversion] [":" format_spec] "}"
f_expression:      (conditional_expression | "*" or_expr)
                     ("," conditional_expression | "," "*" or_expr)* [","]
                   | yield_expression
conversion:        "s" | "r" | "a"
format_spec:       (literal_char | replacement_field)*
literal_char:      <any code point except "{", "}" or NULL>
```

双花括号 '`{{`' 或 '`}}`' 被替换为单花括号，花括号外的字符串仍按字面量处理。单左花括号 '`{`' 标记以 Python 表达式开头的替换字段。在表达式后加等于号 '`=`'，可在求值后，同时显示表达式文

本及其结果（用于调试）。 随后是用叹号 `'!'` 标记的转换字段。还可以在冒号 `':'` 后附加格式说明符。替换字段以右花括号 `'}'` 为结尾。

格式字符串字面量中的表达式会与用圆括号包围的常规 Python 表达式一样处理，但有少量例外。空表达式是不被允许的，而 `lambda` 和赋值表达式 `:=` 都必须显式地用括号包围。 每个表达式都将在格式字符串字面量出现的上下文中按从左到右的顺序进行求值。 替换表达式可在单引号和三引号 f-字符串中包含换行符并可包含注释。 替换字段内 `#` 后面的所有内容都是注释（即使结尾花括号和引号也是）。 在这种情况下，替换字段必须在另一行中结束。

```
>>> f"abc{a # This is a comment }"
... + 3}"
'abc5'
```

> *在 3.7 版本发生变更:* Python 3.7 以前， 因为实现的问题，不允许在格式字符串字面量表达式中使用 `await` 表达式与包含 `async for` 子句的推导式。

> *在 3.12 版本发生变更:* 在 Python 3.12 之前，不允许在 f-字符串的替换字段中使用注释。

表达式里含等号 `'='` 时，输出内容包括表达式文本、`'='` 、求值结果。输出内容可以保留表达式中左花括号 `'{'` 后，及 `'='` 后的空格。没有指定格式时，`'='` 默认调用表达式的 `repr()`。指定了格式时，默认调用表达式的 `str()`，除非声明了转换字段 `'!r'`。

> *Added in version 3.8:* 等号 `'='`。

指定了转换符时，表达式求值的结果会先转换，再格式化。转换符 `'!s'` 调用 `str()` 转换求值结果，`'!r'` 调用 `repr()`，`'!a'` 调用 `ascii()`。

然后使用 `format()` 协议对结果进行格式化。 格式说明符将传给表达式或转换结果的 `__format__()` 方法。 如果省略格式说明符则将传入空字符串。 格式化后的结果将包括在整个字符串的最终值中。

最高层级的格式说明符可以包括嵌套的替换字段。 这些嵌套字段也可以包括它们自己的转换字段和格式说明符，但是不可再包括更深层嵌套的替换字段。 这里的 格式说明符微语言 与 `str.format()` 方法所使用的相同。

格式字符串字面量可以拼接，但是一个替换字段不能拆分到多个字面量。

格式字符串字面量示例如下：

```
>>> name = "Fred"
>>> f"He said his name is {name!r}."
"He said his name is 'Fred'."
>>> f"He said his name is {repr(name)}."  # repr() is equivalent to !r
"He said his name is 'Fred'."
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}"  # nested fields
'result:      12.35'
>>> today = datetime(year=2017, month=1, day=27)
>>> f"{today:%B %d, %Y}"  # using date format specifier
```

```
'January 27, 2017'
>>> f"{today=:%B %d, %Y}" # using date format specifier and debugging
'today=January 27, 2017'
>>> number = 1024
>>> f"{number:#0x}"  # using integer format specifier
'0x400'
>>> foo = "bar"
>>> f"{ foo = }" # preserves whitespace
" foo = 'bar'"
>>> line = "The mill's closed"
>>> f"{line = }"
'line = "The mill\'s closed"'
>>> f"{line = :20}"
"line = The mill's closed   "
>>> f"{line = !r:20}"
'line = "The mill\'s closed" '
```

允许在替换字段中重用外层 f-字符串的引号类型:

```
>>> a = dict(x=2)
>>> f"abc {a["x"]} def"
'abc 2 def'
```

> *在 3.12 版本发生变更:* 在 Python 3.12 之前不允许在替换字段中重用与外层 f-字符串相同的引号类型。

替换字段中也允许使用反斜杠并会以与在其他场景下相同的方式求值:

```
>>> a = ["a", "b", "c"]
>>> print(f"List a contains:\n{"\n".join(a)}")
List a contains:
a
b
c
```

> *在 3.12 版本发生变更:* 在 Python 3.12 之前，f-字符串的替换字段内不允许使用反斜杠。

即便未包含表达式，格式字符串字面量也不能用作文档字符串。

```
>>> def foo():
...     f"Not a docstring"
...
>>> foo.__doc__ is None
True
```

参阅 **PEP 498**，了解格式字符串字面量的提案，以及与格式字符串机制相关的 `str.format()`。

2.5.8. t-strings

> *Added in version 3.14.*

一个 *模板字符串字面值* 或 *t-字符串* 是一个以 '`t`' 或 '`T`' 为前缀的字符串字面值。这些字符串遵循与格式字符串字面值 相同的语法和求值规则，但有以下区别:

- 模板字符串字面值不会求值为 `str` 对象，而是会求值为一个 `string.templatelib.Template` 对象。
- `format()` 协议未被使用。相反，格式说明符和转换（如果有）将被传递给为每个评估表达式创建的新的 `Interpolation` 对象。处理生成的 `Template` 对象的代码将决定如何处理格式说明符和转换。
- 包含嵌套替换字段的格式说明符会在传递给 `Interpolation` 对象之前进行急切求值。例如，形如 `{amount:.{precision}f}` 的插值表达式会先计算内部表达式 `{precision}` 以确定 `format_spec` 属性的值。若 `precision` 的值为 `2`，则最终的格式说明符将是 `'.2f'`。
- 当插值表达式中包含等号 `'='` 时，该表达式的文本（包括等号本身及其周围的空白符）会被追加到相关插值位置之前的字面值字符串之后。该表达式对应的 `Interpolation` 实例会按常规方式创建，只不过其 `conversion` 属性默认会被设为 `'r'`（即使用 `repr()` 函数）。如果提供了显式的转换说明符或格式说明符，将会覆盖这一默认行为。

2.6. 数值字面量

`NUMBER` 标记表示数字字面量，共有三种类型：整数、浮点数和虚数。

```
NUMBER: integer | floatnumber | imagnumber
```

数字字面量的数值等价于将其作为字符串传递给 `int`、`float` 或 `complex` 类构造函数时的值。注意，这些构造函数的有效输入并不都属于合法的字面量格式。

数字字面量不包含符号；像 `-1` 这样的短语实际上是由一元运算符 `'-'` 和字面量 `1` 组成的表达式。

2.6.1. 整数字面量

整数字面量表示整数。例如：

```
7
3
2147483647
```

整数字面量的长度没有限制，仅受可用内存的存储能力限制：

```
7922816251426433759354395033679228162514264337593543950336
```

下划线可用于对数字进行分组以增强可读性，且在确定字面量的数值时会被忽略。例如，以下字面量是等价的：

```
100_000_000_000
100000000000
1_00_00_00_00_000
```

下划线只能出现在数字之间。例如，`_123`、`321_` 和 `123__321` 均 *不是* 有效的字面量。

整数可以分别使用前缀 `0b`、`0o` 和 `0x` 指定为二进制（基数 2）、八进制（基数 8）或十六进制（基数 16）。十六进制数字 10 到 15 用字母 `A-F` 表示，大小写不敏感。例如：

```
0b100110111
0b_1110_0101
0o177
0o377
0xdeadbeef
0xDead_Beef
```

下划线可以紧跟在进制前缀之后。例如，`0x_1f` 是有效的字面量，但 `0_x1f` 和 `0x__1f` 不是。

非零十进制数中不允许有前导零。例如，`0123` 不是有效的字面量。这是为了与 C 风格的八进制字面量区分开，Python 在 3.0 版本之前曾使用这种风格。

形式上，整数字面量由以下词法定义描述：

```
integer:      decinteger | bininteger | octinteger | hexinteger | zerointeger
decinteger:   nonzerodigit (["_"] digit)*
bininteger:   "0" ("b" | "B") (["_"] bindigit)+
octinteger:   "0" ("o" | "O") (["_"] octdigit)+
hexinteger:   "0" ("x" | "X") (["_"] hexdigit)+
zerointeger:  "0"+ (["_"] "0")*
nonzerodigit: "1"..."9"
digit:        "0"..."9"
bindigit:     "0" | "1"
octdigit:     "0"..."7"
hexdigit:     digit | "a"..."f" | "A"..."F"
```

> *在 3.6 版本发生变更:* 现已支持在字面量中，用下划线分组数字。

2.6.2. 浮点数字面量

浮点（float）字面量，例如 `3.14` 或 `1.5`，表示 实数的近似值。

它们由 *整数* 部分和 *小数* 部分组成，每个部分均由十进制数字构成。两部分由小数点 `.` 分隔。：

```
2.71828
4.0
```

Unlike in integer literals, leading zeros are allowed. For example, `077.010` is legal, and denotes the same number as `77.01`.

与整数字面量一样，浮点字面量中的数字之间可以使用单个下划线来提高可读性。：

```
96_485.332_123
3.14_15_93
```

整数部分或小数部分可以为空，但不能同时为空。例如：

```
10.  # (等同于 10.0)
.001  # (等同于 0.001)
```

整数部分和小数部分之后可以选择性地跟随一个 *指数部分*：字母 `e` 或 `E`，后面跟一个可选的符号（`+` 或 `-`），以及一个格式与整数和小数部分相同的数字。这里的 `e` 或 `E` 表示"乘以10的...次幂":

```
1.0e3  # (代表 1.0×10³, or 1000.0)
1.166e-5  # (代表 1.166×10⁻⁵, or 0.00001166)
6.02214076e+23  # (代表 6.02214076×10²³ 或 602214076000000000000000)
```

对于仅包含整数部分和指数部分的浮点字面量，小数点可以省略:

```
1e3  # (等同于 1.e3 和 1.0e3)
0e0  # (等同于 0)
```

形式上，浮点字面量由以下词法定义描述:

```
floatnumber:
   | digitpart "." [digitpart] [exponent]
   | "." digitpart [exponent]
   | digitpart exponent
digitpart: digit (["_"] digit)*
exponent:  ("e" | "E") ["+" | "-"] digitpart
```

> *在 3.6 版本发生变更:* 现已支持在字面量中，用下划线分组数字。

2.6.3. 虚数字面量

Python 拥有 复数 对象，但没有直接的复数字面量。相反，*虚数字面量* 表示实部为零的复数。

例如，在数学中，复数 3+4.2*i* 被写作实数 3 加上虚数 4.2*i*。Python 使用类似的语法，只是虚数单位写作 `j` 而非 *i*:

```
3+4.2j
```

这是一个由 整数字面量 `3`、 运算符 '`+`' 和 虚数字面量 `4.2j` 组成的表达式。由于这是三个独立的词法单元，它们之间允许存在空白符:

```
3 + 4.2j
```

每个词法单元内部不允许有空白符。特别地，`j` 后缀不能与其前面的数字分隔开。

`j` 前面的数字部分与浮点字面量的语法规则相同。因此，以下是有效的虚数字面量:

```
4.2j
3.14j
10.j
.001j
1e100j
3.14e-10j
3.14_15_93j
```

与浮点字面量不同，如果虚数部分仅包含整数部分，则小数点可以省略。该数值仍会被计算为浮点数，而非整数:

```
10j
0j
10000000000000000000000000j    # 等同于 1e+24j
```

`j` 后缀在语法上是大小写不敏感的。这意味着你可以使用 `J` 替代:

```
3.14J   # 等同于 3.14j
```

形式上，虚数字面量由以下词法定义描述:

```
imagnumber: (floatnumber | digitpart) ("j" | "J")
```

2.7. 运算符与定界符

以下语法定义了 *运算符* 和 *定界符* 标记，即通用的 OP 标记类型。 这些标记及其名称的列表 也可在 token 模块文档中找到。

```
OP:
   | assignment_operator
   | bitwise_operator
   | comparison_operator
   | enclosing_delimiter
   | other_delimiter
   | arithmetic_operator
   | "..."
   | other_op

assignment_operator:    "+=" | "-=" | "*=" | "**=" | "/=" | "//=" | "%=" |
                        "&=" | "|=" | "^=" | "<<=" | ">>=" | "@=" | ":="
bitwise_operator:       "&" | "|" | "^" | "~" | "<<" | ">>"
comparison_operator:    "<=" | ">=" | "<" | ">" | "==" | "!="
enclosing_delimiter:    "(" | ")" | "[" | "]" | "{" | "}"
other_delimiter:        "," | ":" | "!" | ";" | "=" | "->"
arithmetic_operator:    "+" | "-" | "**" | "*" | "//" | "/" | "%"
other_op:               "." | "@"
```

备注: 通常，*运算符* 用于组合 表达式，而 *定界符* 则用于其他用途。然而，这两类标记之间并没有明确、正式的区分标准。

某些记号既可用作运算符也可用作定界符，具体取决于使用场景。例如，`*` 既是乘法运算符，也是用于序列解包的定界符；而 `@` 既是矩阵乘法运算符，也是引入装饰器的定界符。

对于某些记号而言，其分类界限并不明确。例如，有些人认为 `.`、`(` 和 `)` 是定界符，而另一些人则将其视为 `getattr()` 运算符和函数调用运算符。

Python 中的部分运算符（如 `and`、`or` 和 `not in`）使用 关键字 记号而非"符号"（运算符记号）实现。

连续三个点号的序列 (`...`) 具有表示一个 `Ellipsis` 字面值的特殊含义。