Lexical analysis

- Line structure
  - Logical lines
  - Physical lines
  - Comments
  - Encoding declarations
  - Explicit line joining
  - Implicit line joining
  - Blank lines
  - Indentation
  - Whitespace between tokens
  - End marker
- Other tokens
- Names (identifiers and keywords)
  - Keywords
  - Soft Keywords
  - Reserved classes of identifiers
  - Non-ASCII characters in names
- Literals
- String and Bytes literals
  - Triple-quoted strings
  - String prefixes
  - Formal grammar
  - Escape sequences
    - Ignored end of line
    - Escaped characters
    - Octal character
    - Hexadecimal character
    - Named Unicode character

    - Hexadecimal Unicode characters
    - Unrecognized escape sequences
  - Bytes literals
  - Raw string literals
  - f-strings
  - t-strings
  - Formal grammar for f-strings
- Numeric literals
  - Integer literals
  - Floating-point literals
  - Imaginary literals
- Operators and delimiters

---

lexical analysis, parser, token

A Python program is read by a *parser*. Input to the parser is a stream of `tokens <token>`, 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 <encodings>` (UTF-8 by default), and decodes the text into `source characters <lexical-source-character>`. 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.

Character	Next token (or other relevant documentation)
• space • tab • formfeed	• `Whitespace <whitespace>`
• CR, LF	• `New line <line-structure>` • `Indentation <indentation>`

Character	Next token (or other relevant documentation)
• backslash (`\`)	• `Explicit line joining <explicit-joining>` • (Also significant in `string escape sequences <escape-sequences>`)
• hash (`#`)	• `Comment <comments>`
• quote (`'`, `"`)	• `String literal <strings>`
• ASCII letter (`a`-`z`, `A`-`Z`) • non-ASCII character	• `Name <identifiers>` • Prefixed `string or bytes literal <strings>`
• underscore (`_`)	• `Name <identifiers>` • (Can also be part of `numeric literals <numbers>`)
• number (`0`-`9`)	• `Numeric literal <numbers>`
• dot (`.`)	• `Numeric literal <numbers>` • `Operator <operators>`
• question mark (`?`) • dollar (`$`) • .. (the following uses zero-width space characters to render .. a literal backquote) > “ backquote (`□`) • control character	• Error (outside string literals and comments)
• other printing character	• `Operator or delimiter <operators>`
• end of file	• `End marker <endmarker-token>`

Line structure

line structure

A Python program is divided into a number of *logical lines*.

Logical lines

logical line, physical line, line joining, NEWLINE token

The end of a logical line is represented by the token `~token.NEWLINE`. Statements cannot cross logical line boundaries except where `!NEWLINE` is allowed by the syntax (e.g., between statements in compound statements). A logical line is constructed from one or more *physical lines* by following the `explicit <explicit-joining>` or `implicit <implicit-joining>` *line joining* rules.

Physical lines

A physical line is a sequence of characters terminated by one the following end-of-line sequences:

- the Unix form using ASCII LF (linefeed),
- the Windows form using the ASCII sequence CR LF (return followed by linefeed),
- the '`Classic Mac OS`__' form using the ASCII CR (return) character.

Regardless of platform, each of these sequences is replaced by a single ASCII LF (linefeed) character. (This is done even inside `string literals <strings>`.) Each line can use any of the sequences; they do not need to be consistent within a file.

The end of input also serves as an implicit terminator for the final physical line.

Formally:

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

Comments

comment, hash character single: # (hash); comment

A comment starts with a hash character (`#`) that is not part of a string literal, and ends at the end of the physical line. A comment signifies the end of the logical line unless the implicit line joining rules are invoked. Comments are ignored by the syntax.

Encoding declarations

source character set, encoding declarations (source file) single: # (hash); source encoding declaration

If a comment in the first or second line of the Python script matches the regular expression `coding[=:]\s*([-\w.]+)`, this comment is processed as an encoding declaration; the first group of this expression names the encoding of the source code file. The encoding declaration must appear on a line of its own. If it is the second line, the first line must also be a comment-only line. The recommended forms of an encoding expression are :

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

which is recognized also by GNU Emacs, and :

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

which is recognized by Bram Moolenaar's VIM.

If no encoding declaration is found, the default encoding is UTF-8. If the implicit or explicit encoding of a file is UTF-8, an initial UTF-8 byte-order mark (`b'\xef\xbb\xbf'`) is ignored rather than being a syntax error.

If an encoding is declared, the encoding name must be recognized by Python (see `standard-encodings`). The encoding is used for all lexical analysis, including string literals, comments and identifiers.

All lexical analysis, including string literals, comments and identifiers, works on Unicode text decoded using the source encoding. Any Unicode code point, except the NUL control character, can appear in Python source.

source_character: <any Unicode code point, except NUL>

Explicit line joining

physical line, line joining, line continuation, backslash character

Two or more physical lines may be joined into logical lines using backslash characters (`\`), as follows: when a physical line ends in a backslash that is not part of a string literal or comment, it is joined with the following forming a single logical line, deleting the backslash and the following end-of-line character. For example:

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # Looks like a valid date
        return 1
```

A line ending in a backslash cannot carry a comment. A backslash does not continue a comment. A backslash does not continue a token except for string literals (i.e., tokens other than string literals cannot be split across physical lines using a backslash). A backslash is illegal elsewhere on a line outside a string literal.

Implicit line joining

Expressions in parentheses, square brackets or curly braces can be split over more than one physical line without using backslashes. For example:

```
month_names = ['Januari', 'Februari', 'Maart',      # These are the
               'April',   'Mei',      'Juni',       # Dutch names
               'Juli',    'Augustus', 'September',  # for the months
               'Oktober', 'November', 'December']   # of the year
```

Implicitly continued lines can carry comments. The indentation of the continuation lines is not important. Blank continuation lines are allowed. There is no NEWLINE token between implicit continuation lines. Implicitly continued lines can also occur within triple-quoted strings (see below); in that case they cannot carry comments.

Blank lines

single: blank line

A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored (i.e., no `~token.NEWLINE` token is generated). During interactive input of statements, handling of a blank line may differ depending on the implementation of the read-eval-print loop. In the standard interactive interpreter, an entirely blank logical line (that is, one containing not even whitespace or a comment) terminates a multi-line statement.

Indentation

indentation, leading whitespace, space, tab, grouping, statement grouping

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

Tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight (this is intended to be the same rule as used by Unix). The total number of spaces preceding the first non-blank character then determines the line's indentation. Indentation cannot be split over multiple physical lines

using backslashes; the whitespace up to the first backslash determines the indentation.

Indentation is rejected as inconsistent if a source file mixes tabs and spaces in a way that makes the meaning dependent on the worth of a tab in spaces; a `TabError` is raised in that case.

Cross-platform compatibility note: because of the nature of text editors on non-UNIX platforms, it is unwise to use a mixture of spaces and tabs for the indentation in a single source file. It should also be noted that different platforms may explicitly limit the maximum indentation level.

A formfeed character may be present at the start of the line; it will be ignored for the indentation calculations above. Formfeed characters occurring elsewhere in the leading whitespace have an undefined effect (for instance, they may reset the space count to zero).

INDENT token, DEDENT token

The indentation levels of consecutive lines are used to generate `~token.INDENT` and `~token.DEDENT` tokens, using a stack, as follows.

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.

Here is an example of a correctly (though confusingly) indented piece of Python code:

```
def perm(l):
        # Compute the list of all permutations of l
    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
```

The following example shows various indentation errors:

```
def perm(l):                        # error: first line indented
for i in range(len(l)):              # error: not indented
   s = l[:i] + l[i+1:]
       p = perm(l[:i] + l[i+1:])   # error: unexpected indent
       for x in p:
               r.append(l[i:i+1] + x)
           return r                 # error: inconsistent dedent
```

(Actually, the first three errors are detected by the parser; only the last error is found by the lexical analyzer --- the indentation of `return r` does not match a level popped off the stack.)

Whitespace between tokens

Except at the beginning of a logical line or in string literals, the whitespace characters space, tab and formfeed can be used interchangeably to separate tokens:

whitespace: ' ' | tab | formfeed

Whitespace is needed between two tokens only if their concatenation could otherwise be interpreted as a different token. For example, `ab` is one token, but `a b` is two tokens. However, `+a` and `+ a` both produce two tokens, `+` and `a`, as `+a` is not a valid token.

End marker

At the end of non-interactive input, the lexical analyzer generates an `~token.ENDMARKER` token.

Other tokens

Besides `~token.NEWLINE`, `~token.INDENT` and `~token.DEDENT`, the following categories of tokens exist: *identifiers* and *keywords* (`~token.NAME`), *literals* (such as `~token.NUMBER` and `~token.STRING`), and other symbols (*operators* and *delimiters*, `~token.OP`). Whitespace characters (other than logical line terminators, discussed earlier) are not tokens, but serve to delimit tokens. Where ambiguity exists, a token comprises the longest possible string that forms a legal token, when read from left to right.

Names (identifiers and keywords)

identifier, name

`~token.NAME` tokens represent *identifiers*, *keywords*, and *soft keywords*.

Names are composed of the following characters:

- uppercase and lowercase letters (`A-Z` and `a-z`),
- the underscore (`_`),
- digits (`0` through `9`), which cannot appear as the first character, and
- non-ASCII characters. Valid names may only contain "letter-like" and "digit-like" characters; see `lexical-names-nonascii` for details.

Names must contain at least one character, but have no upper length limit. Case is significant.

Formally, names are described by the following lexical definitions:

NAME: name_start name_continue* name_start: "a"..."z" | "A"..."Z" | "_" | <non-ASCII character> name_continue: name_start | "0"..."9" identifier: <`NAME`, except keywords>

Note that not all names matched by this grammar are valid; see `lexical-names-nonascii` for details.

Keywords

single: keyword single: reserved word

The following names are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

```
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
```

Soft Keywords

soft keyword, keyword

3.10

Some names are only reserved under specific contexts. These are known as *soft keywords*:

- `match`, `case`, and `_`, when used in the `match` statement.
- `type`, when used in the `type` statement.
- `lazy`, when used before an `import` statement.

These syntactically act as keywords in their specific contexts, but this distinction is done at the parser level, not when tokenizing.

As soft keywords, their use in the grammar is possible while still preserving compatibility with existing code that uses these names as identifier names.

3.12 `type` is now a soft keyword.

3.15 `lazy` is now a soft keyword.

single: _, identifiers single: __, identifiers

Reserved classes of identifiers

Certain classes of identifiers (besides keywords) have special meanings. These classes are identified by the patterns of leading and trailing underscore characters:

`_*`
Not imported by `from module import *`.

`_`
In a `case` pattern within a `match` statement, `_` is a `soft keyword <soft-keywords>` that denotes a `wildcard <wildcard-patterns>`.

Separately, the interactive interpreter makes the result of the last evaluation available in the variable `_`. (It is stored in the `builtins` module, alongside built-in functions like `print`.)

Elsewhere, `_` is a regular identifier. It is often used to name "special" items, but it is not special to Python itself.

> [!NOTE] The name `_` is often used in conjunction with internationalization; refer to the documentation for the `gettext` module for more information on this convention.
>
> It is also commonly used for unused variables.

`__*__`
System-defined names, informally known as "dunder" names. These names are defined by the interpreter and its implementation (including the standard library). Current system names are

discussed in the `specialnames` section and elsewhere. More will likely be defined in future versions of Python. *Any* use of `__*__` names, in any context, that does not follow explicitly documented use, is subject to breakage without warning.

`__*`
Class-private names. Names in this category, when used within the context of a class definition, are re-written to use a mangled form to help avoid name clashes between "private" attributes of base and derived classes. See section `atom-identifiers`.

Non-ASCII characters in names

Names that contain non-ASCII characters need additional normalization and validation beyond the rules and grammar explained `above <identifiers>`. For example, `ř_1`, `μ`, or `日本語` are valid names, but `r□2`, `€`, or `□□`are not.

This section explains the exact rules.

All names are converted into the normalization form NFKC while parsing. This means that, for example, some typographic variants of characters are converted to their "basic" form. For example, `fiⁿₐˡᵢᶻₐᵗᵢᵒₙ` normalizes to `finalization`, so Python treats them as the same name:

```
>>> fiⁿₐˡᵢᶻₐᵗᵢᵒₙ = 3
>>> finalization
3
```

> [!NOTE] Normalization is done at the lexical level only. Run-time functions that take names as *strings* generally do not normalize their arguments. For example, the variable defined above is accessible at run time in the `globals` dictionary as `globals()["finalization"]` but not `globals()["fiⁿₐˡᵢᶻₐᵗᵢᵒₙ"]`.

Similarly to how ASCII-only names must contain only letters, digits and the underscore, and cannot start with a digit, a valid name must start with a character in the "letter-like" set `xid_start`, and the remaining characters must be in the "letter- and digit-like" set `xid_continue`.

These sets are based on the *XID_Start* and *XID_Continue* sets as defined by the Unicode standard annex UAX-31. Python's `xid_start` additionally includes the underscore (`_`). Note that Python does not necessarily conform to UAX-31.

A non-normative listing of characters in the *XID_Start* and *XID_Continue* sets as defined by Unicode is available in the DerivedCoreProperties.txt file in the Unicode Character Database. For reference, the construction rules for the `xid_*` sets are given below.

The set `id_start` is defined as the union of:

- Unicode category `<Lu>` - uppercase letters (includes `A` to `Z`)
- Unicode category `<Ll>` - lowercase letters (includes `a` to `z`)
- Unicode category `<Lt>` - titlecase letters
- Unicode category `<Lm>` - modifier letters
- Unicode category `<Lo>` - other letters
- Unicode category `<Nl>` - letter numbers
- {`"_"`} - the underscore
- `<Other_ID_Start>` - an explicit set of characters in PropList.txt to support backwards compatibility

The set `xid_start` then closes this set under NFKC normalization, by removing all characters whose normalization is not of the form `id_start id_continue*`.

The set `id_continue` is defined as the union of:

- `id_start` (see above)
- Unicode category `<Nd>` - decimal numbers (includes `0` to `9`)
- Unicode category `<Pc>` - connector punctuations
- Unicode category `<Mn>` - nonspacing marks
- Unicode category `<Mc>` - spacing combining marks
- `<Other_ID_Continue>` - another explicit set of characters in PropList.txt to support backwards compatibility

Again, `xid_continue` closes this set under NFKC normalization.

Unicode categories use the version of the Unicode Character Database as included in the `unicodedata` module.

- `3131` -- Supporting Non-ASCII Identifiers
- `672` -- Unicode-related Security Considerations for Python

Literals

literal, constant

Literals are notations for constant values of some built-in types.

In terms of lexical analysis, Python has `string, bytes <strings>` and `numeric <numbers>` literals.

Other "literals" are lexically denoted using `keywords <keywords>` (`None`, `True`, `False`) and the special `ellipsis token <lexical-ellipsis>` (`...`).

string literal, bytes literal, ASCII single: ' (single quote); string literal single: " (double quote); string literal

String and Bytes literals

String literals are text enclosed in single quotes (`'`) or double quotes (`"`). For example:

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

The quote used to start the literal also terminates it, so a string literal can only contain the other quote (except with escape sequences, see below). For example:

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

Except for this limitation, the choice of quote character (`'` or `"`) does not affect how the literal is parsed.

Inside a string literal, the backslash (`\`) character introduces an `escape sequence`, which has special meaning depending on the character after the backslash. For example, `\"` denotes the double quote character, and does *not* end the string:

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

See `escape sequences <escape-sequences>` below for a full list of such sequences, and more details.

triple-quoted string single: """; string literal single: '''; string literal

Triple-quoted strings

Strings can also be enclosed in matching groups of three single or double quotes. These are generally referred to as `triple-quoted strings`:

```
"""This is a triple-quoted string."""
```

In triple-quoted literals, unescaped quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the literal, if they are of the same kind (`'` or `"`) used at the start:

```
"""This string has "quotes" inside."""
```

Unescaped newlines are also allowed and retained:

```
'''This triple-quoted string
continues on the next line.'''
```

single: u'; string literal single: u"; string literal

String prefixes

String literals can have an optional `prefix` that influences how the content of the literal is parsed, for example:

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

The allowed prefixes are:

- `b`: `Bytes literal <bytes-literal>`
- `r`: `Raw string <raw-strings>`
- `f`: `Formatted string literal <f-strings>` ("f-string")
- `t`: `Template string literal <t-strings>` ("t-string")
- `u`: No effect (allowed for backwards compatibility)

See the linked sections for details on each type.

Prefixes are case-insensitive (for example, '`B`' works the same as '`b`'). The '`r`' prefix can be combined with '`f`', '`t`' or '`b`', so '`fr`', '`rf`', '`tr`', '`rt`', '`br`', and '`rb`' are also valid prefixes.

3.3 The `'rb'` prefix of raw bytes literals has been added as a synonym of `'br'`.

Support for the unicode legacy literal (`u'value'`) was reintroduced to simplify the maintenance of dual Python 2.x and 3.x codebases. See `414` for more information.

Formal grammar

String literals, except `"f-strings" <f-strings>` and `"t-strings" <t-strings>`, are described by the following lexical definitions.

These definitions use `negative lookaheads <lexical-lookaheads>` (`!`) to indicate that an ending quote ends the literal.

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>

Note that as in all lexical definitions, whitespace is significant. In particular, the prefix (if any) must be immediately followed by the starting quote.

physical line, escape sequence, Standard C, C single: (backslash); escape sequence single: \; escape sequence single: a; escape sequence single: b; escape sequence single: f; escape sequence single: n; escape sequence single: r; escape sequence single: t; escape sequence single: v; escape sequence single: x; escape sequence single: N; escape sequence single: u; escape sequence single: U; escape sequence

Escape sequences

Unless an '`r`' or '`R`' prefix is present, escape sequences in string and bytes literals are interpreted according to rules similar to those used by Standard C. The recognized escape sequences are:

Escape Sequence	Meaning
`\`<newline>	`string-escape-ignore`
`\\`	`Backslash <string-escape-escaped-char>`
`\'`	`Single quote <string-escape-escaped-char>`
`\"`	`Double quote <string-escape-escaped-char>`
`\a`	ASCII Bell (BEL)
`\b`	ASCII Backspace (BS)
`\f`	ASCII Formfeed (FF)
`\n`	ASCII Linefeed (LF)
`\r`	ASCII Carriage Return (CR)
`\t`	ASCII Horizontal Tab (TAB)
`\v`	ASCII Vertical Tab (VT)
`\\\{ooo}`	`string-escape-oct`
`\x{hh}`	`string-escape-hex`
`\N\{{name}\}`	`string-escape-named`
`\u{xxxx}`	`Hexadecimal Unicode character <string-escape-long-hex>`
`\U{xxxxxxxx}`	`Hexadecimal Unicode character <string-escape-long-hex>`

Ignored end of line

A backslash can be added at the end of a line to ignore the newline:

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

The same result can be achieved using `triple-quoted strings <strings>`, or parentheses and `string literal concatenation <string-concatenation>`.

Escaped characters

To include a backslash in a non-`raw <raw-strings>` Python string literal, it must be doubled. The `\\` escape sequence denotes a single backslash character:

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

Similarly, the `\'` and `\"` sequences denote the single and double quote character, respectively:

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

Octal character

The sequence `\\\\{ooo}` denotes a *character* with the octal (base 8) value *ooo*:

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

Up to three octal digits (0 through 7) are accepted.

In a bytes literal, *character* means a *byte* with the given value. In a string literal, it means a Unicode character with the given value.

3.11 Octal escapes with value larger than `0o377` (255) produce a `DeprecationWarning`.

3.12 Octal escapes with value larger than `0o377` (255) produce a `SyntaxWarning`. In a future Python version they will raise a `SyntaxError`.

Hexadecimal character

The sequence `\\x{hh}` denotes a *character* with the hex (base 16) value *hh*:

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

Unlike in Standard C, exactly two hex digits are required.

In a bytes literal, *character* means a *byte* with the given value. In a string literal, it means a Unicode character with the given value.

Named Unicode character

The sequence `\\N\\{{name}\\}` denotes a Unicode character with the given *name*:

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

This sequence cannot appear in `bytes literals <bytes-literal>`.

3.3 Support for name aliases has been added.

Hexadecimal Unicode characters

These sequences `\\u{xxxx}` and `\\U{xxxxxxxx}` denote the Unicode character with the given hex (base 16) value. Exactly four digits are required for `\u`; exactly eight digits are required for `\U`. The latter can encode any Unicode character.

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

These sequences cannot appear in `bytes literals <bytes-literal>`.

unrecognized escape sequence

Unrecognized escape sequences

Unlike in Standard C, all unrecognized escape sequences are left in the string unchanged, that is, *the backslash is left in the result*:

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

Note that for bytes literals, the escape sequences only recognized in string literals (`\N...`, `\u...`, `\U...`) fall into the category of unrecognized escapes.

3.6 Unrecognized escape sequences produce a `DeprecationWarning`.

3.12 Unrecognized escape sequences produce a `SyntaxWarning`. In a future Python version they will raise a `SyntaxError`.

single: b'; bytes literal single: b"; bytes literal

Bytes literals

`Bytes literals` are always prefixed with '`b`' or '`B`'; they produce an instance of the `bytes` type instead of the `str` type. They may only contain ASCII characters; bytes with a numeric value of 128 or greater must be expressed with escape sequences (typically `string-escape-hex` or `string-escape-oct`):

```
>>> 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]
```

Similarly, a zero byte must be expressed using an escape sequence (typically `\0` or `\x00`).

single: r'; raw string literal single: r"; raw string literal

Raw string literals

Both string and bytes literals may optionally be prefixed with a letter '`r`' or '`R`'; such constructs are called `raw string literals` and `raw bytes literals` respectively and treat backslashes as literal characters. As a result, in raw string literals, `escape sequences <escape-sequences>` are not treated specially:

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

Even in a raw literal, quotes can be escaped with a backslash, but the backslash remains in the result; for example, `r"\""` is a valid string literal consisting of two characters: a backslash and a double quote; `r"\"` is not a valid string literal (even a raw string cannot end in an odd number of backslashes). Specifically, *a raw literal cannot end in a single backslash* (since the backslash would escape the following quote character). Note also that a single backslash followed by a newline is interpreted as those two characters as part of the literal, *not* as a line continuation.

single: formatted string literal single: interpolated string literal single: string; formatted literal single: string; interpolated literal single: f-string single: fstring single: f'; formatted string literal single: f"; formatted string literal single: {} (curly brackets); in formatted string literal single: ! (exclamation); in formatted string literal single: : (colon); in formatted string literal single: = (equals); for help in debugging using string literals

f-strings

3.6

3.7 The `await` and `async for` can be used in expressions within f-strings.

3.8 Added the debug specifier (`=`)

3.12 Many restrictions on expressions within f-strings have been removed. Notably, nested strings, comments, and backslashes are now permitted.

A `formatted string literal` or `f-string` is a string literal that is prefixed with '`f`' or '`F`'. Unlike other string literals, f-strings do not have a constant value. They may contain *replacement fields* delimited by curly braces `{}`. Replacement fields contain expressions which are evaluated at run time. For example:

```
>>> who = 'nobody'
>>> nationality = 'Spanish'
>>> f'{who.title()} expects the {nationality} Inquisition!'
'Nobody expects the Spanish Inquisition!'
```

Any doubled curly braces (`{{` or `}}`) outside replacement fields are replaced with the corresponding single curly brace:

```
>>> print(f'{{...}}')
{...}
```

Other characters outside replacement fields are treated like in ordinary string literals. This means that escape sequences are decoded (except when a literal is also marked as a raw string), and newlines are possible in triple-quoted f-strings:

```
>>> name = 'Galahad'
>>> favorite_color = 'blue'
>>> print(f'{name}:\t{favorite_color}')
Galahad:       blue
>>> print(rf"C:\Users\{name}")
C:\Users\Galahad
>>> print(f'''Three shall be the number of the counting
... and the number of the counting shall be three.''')
Three shall be the number of the counting
and the number of the counting shall be three.
```

Expressions in formatted string literals are treated like regular Python expressions. Each expression is evaluated in the context where the formatted string literal appears, in order from left to right. An empty expression is not allowed, and both `lambda` and assignment expressions `:=` must be surrounded by explicit parentheses:

```
>>> f'{(half := 1/2)}, {half * 42}'
'0.5, 21.0'
```

Reusing the outer f-string quoting type inside a replacement field is permitted:

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

Backslashes are also allowed in replacement fields and are evaluated the same way as in any other context:

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

It is possible to nest f-strings:

```
>>> name = 'world'
>>> f'Repeated:{f' hello {name}' * 3}'
'Repeated: hello world hello world hello world'
```

Portable Python programs should not use more than 5 levels of nesting.

CPython does not limit nesting of f-strings.

Replacement expressions can contain newlines in both single-quoted and triple-quoted f-strings and they can contain comments. Everything that comes after a `#` inside a replacement field is a comment (even closing braces and quotes). This means that replacement fields with comments must be closed in a different line:

```
>>> a = 2
>>> f"abc{a  # This comment  }"  continues until the end of the line
...       + 3}"
'abc5'
```

After the expression, replacement fields may optionally contain:

- a *debug specifier* -- an equal sign (`=`), optionally surrounded by whitespace on one or both sides;
- a *conversion specifier* -- `!s`, `!r` or `!a`; and/or
- a *format specifier* prefixed with a colon (`:`).

See the `Standard Library section on f-strings <stdtypes-fstrings>` for details on how these fields are evaluated.

As that section explains, *format specifiers* are passed as the second argument to the `format` function to format a replacement field value. For example, they can be used to specify a field width and padding characters using the `Format Specification Mini-Language <formatspec>`:

```
>>> number = 14.3
>>> f'{number:20.7f}'
'          14.3000000'
```

Top-level format specifiers may include nested replacement fields:

```
>>> field_size = 20
>>> precision = 7
>>> f'{number:{field_size}.{precision}f}'
'          14.3000000'
```

These nested fields may include their own conversion fields and `format specifiers <formatspec>`:

```
>>> number = 3
>>> f'{number:{field_size}}'
'                   3'
>>> f'{number:{field_size:05}}'
```

```
'0000000000000000003'
```

However, these nested fields may not include more deeply nested replacement fields.

Formatted string literals cannot be used as `docstrings <docstring>`, even if they do not include expressions:

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

- `498` -- Literal String Interpolation
- `701` -- Syntactic formalization of f-strings
- `str.format`, which uses a related format string mechanism.

t-strings

3.14

A `template string literal` or `t-string` is a string literal that is prefixed with '`t`' or '`T`'. These strings follow the same syntax rules as `formatted string literals <f-strings>`. For differences in evaluation rules, see the `Standard Library section on t-strings <stdtypes-tstrings>`

Formal grammar for f-strings

F-strings are handled partly by the `lexical analyzer`, which produces the tokens `~token.FSTRING_START`, `~token.FSTRING_MIDDLE` and `~token.FSTRING_END`, and partly by the parser, which handles expressions in the replacement field. The exact way the work is split is a CPython implementation detail.

Correspondingly, the f-string grammar is a mix of `lexical and syntactic definitions <notation-lexical-vs-syntactic>`.

Whitespace is significant in these situations:

- There may be no whitespace in `~token.FSTRING_START` (between the prefix and quote).
- Whitespace in `~token.FSTRING_MIDDLE` is part of the literal string contents.
- In `fstring_replacement_field`, if `f_debug_specifier` is present, all whitespace after the opening brace until the `f_debug_specifier`, as well as whitespace immediately following `f_debug_specifier`, is retained as part of the expression.

> The expression is not handled in the tokenization phase; it is retrieved from the source code using locations of the `{` token and the token after `=`.

The `FSTRING_MIDDLE` definition uses `negative lookaheads <lexical-lookaheads>` (`!`) to indicate special characters (backslash, newline, `{`, `}`) and sequences (`f_quote`).

fstring: FSTRING_START fstring_middle* FSTRING_END

FSTRING_START: fstringprefix ("'" | '"' | "'''" | '"""') FSTRING_END: f_quote fstringprefix: <("f" | "fr" | "rf"), case-insensitive> f_debug_specifier: '=' f_quote: <the quote character(s) used in FSTRING_START>

fstring_middle:
fstring_replacement_field
FSTRING_MIDDLE

FSTRING_MIDDLE:
(!"" !`newline` !'{' !'}' !`f_quote`) source_character
stringescapeseq
"{{"
"}}"
<newline, in triple-quoted f-strings only>

fstring_replacement_field:
'{' f_expression [f_debug_specifier] [fstring_conversion] [fstring_full_format_spec] '}'

fstring_conversion:
"!" ("s" | "r" | "a")

fstring_full_format_spec:
':' fstring_format_spec*

fstring_format_spec:
FSTRING_MIDDLE
fstring_replacement_field

f_expression:
','.(conditional_expression | "*" or_expr)+ [","]
yield_expression

> [!NOTE] In the above grammar snippet, the `f_quote` and `FSTRING_MIDDLE` rules are context-sensitive -- they depend on the contents of `FSTRING_START` of the nearest enclosing `fstring`.

> Constructing a more traditional formal grammar from this template is left as an exercise for the reader.

The grammar for t-strings is identical to the one for f-strings, with *t* instead of *f* at the beginning of rule and token names and in the prefix.

tstring: TSTRING_START tstring_middle* TSTRING_END

<rest of the t-string grammar is omitted; see above>

Numeric literals

number, numeric literal, integer literal floating-point literal, hexadecimal literal octal literal, binary literal, decimal literal, imaginary literal, complex literal

`~token.NUMBER` tokens represent numeric literals, of which there are three types: integers, floating-point numbers, and imaginary numbers.

NUMBER: integer | floatnumber | imagnumber

The numeric value of a numeric literal is the same as if it were passed as a string to the `int`, `float` or `complex` class constructor, respectively. Note that not all valid inputs for those constructors are also valid literals.

Numeric literals do not include a sign; a phrase like `-1` is actually an expression composed of the unary operator '`-`' and the literal `1`.

single: 0b; integer literal single: 0o; integer literal single: 0x; integer literal single: _ (underscore); in numeric literal

Integer literals

Integer literals denote whole numbers. For example:

```
7
3
2147483647
```

There is no limit for the length of integer literals apart from what can be stored in available memory:

```
7922816251426433759354395033679228162514264337593543950336
```

Underscores can be used to group digits for enhanced readability, and are ignored for determining the numeric value of the literal. For example, the following literals are equivalent:

```
100_000_000_000
100000000000
1_00_00_00_00_000
```

Underscores can only occur between digits. For example, `_123`, `321_`, and `123__321` are *not* valid literals.

Integers can be specified in binary (base 2), octal (base 8), or hexadecimal (base 16) using the prefixes `0b`, `0o` and `0x`, respectively. Hexadecimal digits 10 through 15 are represented by letters `A`-`F`, case-insensitive. For example:

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

An underscore can follow the base specifier. For example, `0x_1f` is a valid literal, but `0_x1f` and `0x__1f` are not.

Leading zeros in a non-zero decimal number are not allowed. For example, `0123` is not a valid literal. This is for disambiguation with C-style octal literals, which Python used before version 3.0.

Formally, integer literals are described by the following lexical definitions:

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"

3.6 Underscores are now allowed for grouping purposes in literals.

single: . (dot); in numeric literal single: e; in numeric literal single: _ (underscore); in numeric literal

Floating-point literals

Floating-point (float) literals, such as `3.14` or `1.5`, denote `approximations of real numbers <datamodel-float>`.

They consist of *integer* and *fraction* parts, each composed of decimal digits. The parts are separated by a decimal point, `.`:

```
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`.

As in integer literals, single underscores may occur between digits to help readability:

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

Either of these parts, but not both, can be empty. For example:

```
10.  # (equivalent to 10.0)
.001  # (equivalent to 0.001)
```

Optionally, the integer and fraction may be followed by an *exponent*: the letter `e` or `E`, followed by an optional sign, `+` or `-`, and a number in the same format as the integer and fraction parts. The `e` or `E` represents "times ten raised to the power of":

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

In floats with only integer and exponent parts, the decimal point may be omitted:

```
1e3  # (equivalent to 1.e3 and 1.0e3)
0e0  # (equivalent to 0.)
```

Formally, floating-point literals are described by the following lexical definitions:

floatnumber:
digitpart "." [digitpart] [exponent]
"." digitpart [exponent]
digitpart exponent

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

3.6 Underscores are now allowed for grouping purposes in literals.

single: j; in numeric literal

Imaginary literals

Python has `complex number <typesnumeric>` objects, but no complex literals. Instead, *imaginary literals* denote complex numbers with a zero real part.

For example, in math, the complex number 3+4.2*i* is written as the real number 3 added to the imaginary number 4.2*i*. Python uses a similar syntax, except the imaginary unit is written as `j` rather than *i*:

```
3+4.2j
```

This is an expression composed of the `integer literal <integers>` `3`, the `operator <operators>` '`+`', and the `imaginary literal <imaginary>` `4.2j`. Since these are three separate tokens, whitespace is allowed between them:

```
3 + 4.2j
```

No whitespace is allowed *within* each token. In particular, the `j` suffix, may not be separated from the number before it.

The number before the `j` has the same syntax as a floating-point literal. Thus, the following are valid imaginary literals:

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

Unlike in a floating-point literal the decimal point can be omitted if the imaginary number only has an integer part. The number is still evaluated as a floating-point number, not an integer:

```
10j
0j
10000000000000000000000000j   # equivalent to 1e+24j
```

The `j` suffix is case-insensitive. That means you can use `J` instead:

```
3.14J   # equivalent to 3.14j
```

Formally, imaginary literals are described by the following lexical definition:

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

Operators and delimiters

single: operators single: delimiters

The following grammar defines `operator` and `delimiter` tokens, that is, the generic `~token.OP` token type. A `list of these tokens and their names <token_operators_delimiters>` is also available in the `!token` module documentation.

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: "." | "@"

> [!NOTE] Generally, *operators* are used to combine `expressions <expressions>`, while *delimiters* serve other purposes. However, there is no clear, formal distinction between the two categories.
>
> Some tokens can serve as either operators or delimiters, depending on usage. For example, `*` is both the multiplication operator and a delimiter used for sequence unpacking, and `@` is both the matrix multiplication and a delimiter that introduces decorators.

> For some tokens, the distinction is unclear. For example, some people consider `.`, `(`, and `)` to be delimiters, while others see the `getattr` operator and the function call operator(s).
>
> Some of Python's operators, like `and`, `or`, and `not in`, use `keyword <keywords>` tokens rather than "symbols" (operator tokens).

A sequence of three consecutive periods (`...`) has a special meaning as an `Ellipsis` literal.

□□□ #2
□ □□□ □□□ 2026-10-31 17:26:17 UTC
□ □□□ □□□ 2026-10-31 17:26:20 UTC