Lexical analysis

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 8)

Unknown directive type "index".

```
.. index:: lexical analysis, parser, token
```

A Python program is read by a *parser*. Input to the parser is a stream of *tokens*, generated by the *lexical analyzer*. This chapter describes how the lexical analyzer breaks a file into tokens.

Python reads program text as Unicode code points; the encoding of a source file can be given by an encoding declaration and defaults to UTF-8, see PEP 3120 for details. If the source file cannot be decoded, a :exc:`SyntaxError` is raised.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 14); *backlink*

Unknown interpreted text role "exc".

Line structure

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 25)

Unknown directive type "index".

```
.. index:: line structure
```

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

Logical lines

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 35)

Unknown directive type "index".

```
.. index:: logical line, physical line, line joining, NEWLINE token
```

The end of a logical line is represented by the 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 or implicit *line joining* rules.

Physical lines

A physical line is a sequence of characters terminated by an end-of-line sequence. In source files and strings, any of the standard platform line termination sequences can be used - the Unix form using ASCII LF (linefeed), the Windows form using the ASCII sequence CR LF (return followed by linefeed), or the old Macintosh form using the ASCII CR (return) character. All of these forms can be used equally, regardless of platform. The end of input also serves as an implicit terminator for the final physical line.

When embedding Python, source code strings should be passed to Python APIs using the standard C conventions for newline characters (the `\n` character, representing ASCII LF, is the line terminator).

Comments

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 67)

Unknown directive type "index".

```
.. index:: 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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 81)

Unknown directive type "index".

```
.. index:: 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. In addition, if the first bytes of the file are the UTF-8 byte-order mark (`b'\xef\xbb\xbf'`), the declared file encoding is UTF-8 (this is supported, among others, by Microsoft's :program:`notepad`).

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 99); *backlink*

Unknown interpreted text role "program".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 104); *backlink*

Unknown interpreted text role "ref".

Explicit line joining

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 115)

Unknown directive type "index".

```
.. index:: 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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 160)

Unknown directive type "index".

```
.. index:: single: blank line
```

A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored (i.e., no 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 (i.e. one containing not even whitespace or a comment) terminates a multi-line statement.

Indentation

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 175)

Unknown directive type "index".

```
.. index:: 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 :exc:`TabError` is raised in that case.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 189); *backlink*

Unknown interpreted text role "exc".

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).

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 203)

Unknown directive type "index".

```
.. index:: INDENT token, DEDENT token
```

The indentation levels of consecutive lines are used to generate INDENT and 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 is needed between two tokens only if their concatenation could otherwise be interpreted as a different token (e.g., ab is one token, but a b is two tokens).

Other tokens

Besides NEWLINE, INDENT and DEDENT, the following categories of tokens exist: *identifiers*, *keywords*, *literals*, *operators*, and *delimiters*. Whitespace characters (other than 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.

Identifiers and keywords

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 278)

Unknown directive type "index".

```
.. index:: identifier, name
```

Identifiers (also referred to as *names*) are described by the following lexical definitions.

The syntax of identifiers in Python is based on the Unicode standard annex UAX-31, with elaboration and changes as defined below; see also PEP 3131 for further details.

Within the ASCII range (U+0001..U+007F), the valid characters for identifiers are the same as in Python 2.x: the uppercase and lowercase letters `A` through `Z`, the underscore `_` and, except for the first character, the digits `0` through `9`.

Python 3.0 introduces additional characters from outside the ASCII range (see PEP 3131). For these characters, the classification uses the version of the Unicode Character Database as included in the :mod:`unicodedata` module.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 292); *backlink*

Unknown interpreted text role "mod".

Identifiers are unlimited in length. Case is significant.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 298)

Unknown directive type "productionlist".

```
.. productionlist:: python-grammar
   identifier: `xid_start` `xid_continue`*
   id_start: <all characters in general categories Lu, Ll, Lt, Lm, Lo, Nl, the underscore, and characters with the Other
   id_continue: <all characters in `id_start`, plus characters in the categories Mn, Mc, Nd, Pc and others with the Othe
   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*">
```

The Unicode category codes mentioned above stand for:

- *Lu* - uppercase letters
- *Ll* - lowercase letters
- *Lt* - titlecase letters
- *Lm* - modifier letters
- *Lo* - other letters
- *Nl* - letter numbers
- *Mn* - nonspacing marks

- *Mc* - spacing combining marks
- *Nd* - decimal numbers
- *Pc* - connector punctuations
- *Other_ID_Start* - explicit list of characters in PropList.txt to support backwards compatibility
- *Other_ID_Continue* - likewise

All identifiers are converted into the normal form NFKC while parsing; comparison of identifiers is based on NFKC.

A non-normative HTML file listing all valid identifier characters for Unicode 14.0.0 can be found at https://www.unicode.org/Public/14.0.0/ucd/DerivedCoreProperties.txt

Keywords

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 335)

Unknown directive type "index".

```
.. index::
   single: keyword
   single: reserved word
```

The following identifiers 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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 359)

Unknown directive type "index".

```
.. index:: soft keyword, keyword
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 361)

Unknown directive type "versionadded".

```
.. versionadded:: 3.10
```

Some identifiers are only reserved under specific contexts. These are known as *soft keywords*. The identifiers `match`, `case` and `_` can syntactically act as keywords in contexts related to the pattern matching statement, but this distinction is done at the parser level, not when tokenizing.

As soft keywords, their use with pattern matching is possible while still preserving compatibility with existing code that uses `match`, `case` and `_` as identifier names.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 374)

Unknown directive type "index".

```
.. index::
   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 :keyword:`match` statement, `_` is a :ref:`soft keyword <soft-keywords>` that denotes a :ref:`wildcard <wildcard-patterns>`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 390); *backlink*

Unknown interpreted text role "keyword".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 390); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 390); *backlink*

Unknown interpreted text role "ref".

Separately, the interactive interpreter makes the result of the last evaluation available in the variable `_`. (It is stored in the

:mod:`builtins` module, alongside built-in functions like `print`.)

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 394); *backlink*
>
> Unknown interpreted text role "mod".

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 :mod:`gettext` module for more information on this convention.
>
> > **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 404); *backlink*
> >
> > Unknown interpreted text role "mod".
>
> 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 :ref:`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.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 411); *backlink*
>
> Unknown interpreted text role "ref".

__*

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 :ref:`atom-identifiers`.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 419); *backlink*
>
> Unknown interpreted text role "ref".

Literals

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 430)
>
> Unknown directive type "index".
>
> ```
> .. index:: literal, constant
> ```

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

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 435)
>
> Unknown directive type "index".
>
> ```
> .. index:: string literal, bytes literal, ASCII
>    single: ' (single quote); string literal
>    single: " (double quote); string literal
>    single: u'; string literal
>    single: u"; string literal
> ```

String and Bytes literals

String literals are described by the following lexical definitions:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 447)
>
> Unknown directive type "productionlist".
>
> ```
> .. productionlist:: python-grammar
>    stringliteral: [`stringprefix`](`shortstring` | `longstring`)
>    stringprefix: "r" | "u" | "R" | "U" | "f" | "F"
>                : | "fr" | "Fr" | "fR" | "FR" | "rf" | "rF" | "Rf" | "RF"
>    shortstring: "'" `shortstringitem`* "'" | '"' `shortstringitem`* '"'
>    longstring: "'''" `longstringitem`* "'''" | '"""' `longstringitem`* '"""'
>    shortstringitem: `shortstringchar` | `stringescapeseq`
>    longstringitem: `longstringchar` | `stringescapeseq`
>    shortstringchar: <any source character except "\" or newline or the quote>
>    longstringchar: <any source character except "\">
>    stringescapeseq: "\" <any source character>
> ```

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 459)
>
> Unknown directive type "productionlist".
>
> ```
> .. productionlist:: python-grammar
> ```

```
bytesliteral: `bytesprefix`(`shortbytes` | `longbytes`)
bytesprefix: "b" | "B" | "br" | "Br" | "bR" | "BR" | "rb" | "rB" | "Rb" | "RB"
shortbytes: "'" `shortbytesitem`* "'" | '"' `shortbytesitem`* '"'
longbytes: "'''" `longbytesitem`* "'''" | '"""' `longbytesitem`* '"""'
shortbytesitem: `shortbyteschar` | `bytesescapeseq`
longbytesitem: `longbyteschar` | `bytesescapeseq`
shortbyteschar: <any ASCII character except "\" or newline or the quote>
longbyteschar: <any ASCII character except "\">
bytesescapeseq: "\" <any ASCII character>
```

One syntactic restriction not indicated by these productions is that whitespace is not allowed between the :token:`~python-grammar:stringprefix` or :token:`~python-grammar:bytesprefix` and the rest of the literal. The source character set is defined by the encoding declaration; it is UTF-8 if no encoding declaration is given in the source file; see section :ref:`encodings`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 470); *backlink*

Unknown interpreted text role "token".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 470); *backlink*

Unknown interpreted text role "token".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 470); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 476)

Unknown directive type "index".

```
.. index:: triple-quoted string, Unicode Consortium, raw string
   single: """; string literal
   single: '''; string literal
```

In plain English: Both types of literals can be enclosed in matching single quotes (`'`) or double quotes (`"`). They can also be enclosed in matching groups of three single or double quotes (these are generally referred to as *triple-quoted strings*). The backslash (`\`) character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 487)

Unknown directive type "index".

```
.. index::
   single: b'; bytes literal
   single: b"; bytes literal
```

Bytes literals are always prefixed with `'b'` or `'B'`; they produce an instance of the :class:`bytes` type instead of the :class:`str` type. They may only contain ASCII characters; bytes with a numeric value of 128 or greater must be expressed with escapes.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 491); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 491); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 496)

Unknown directive type "index".

```
.. index::
   single: r'; raw string literal
   single: r"; raw string literal
```

Both string and bytes literals may optionally be prefixed with a letter `'r'` or `'R'`; such strings are called :dfn:`raw strings` and treat backslashes as literal characters. As a result, in string literals, `'\U'` and `'\u'` escapes in raw strings are not treated specially. Given that Python 2.x's raw unicode literals behave differently than Python 3.x's the `'ur'` syntax is not supported.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 500); *backlink*

Unknown interpreted text role "dfn".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 507)

Unknown directive type "versionadded".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 511)

Unknown directive type "versionadded".

```
.. versionadded:: 3.3
   Support for the unicode legacy literal (``u'value'``) was reintroduced
   to simplify the maintenance of dual Python 2.x and 3.x codebases.
   See :pep:`414` for more information.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 516)

Unknown directive type "index".

```
.. index::
   single: f'; formatted string literal
   single: f"; formatted string literal
```

A string literal with `'f'` or `'F'` in its prefix is a :dfn:`formatted string literal`; see :ref:`f-strings`. The `'f'` may be combined with `'r'`, but not with `'b'` or `'u'`, therefore raw formatted strings are possible, but formatted bytes literals are not.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 520); *backlink*

Unknown interpreted text role "dfn".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 520); *backlink*

Unknown interpreted text role "ref".

In triple-quoted literals, unescaped newlines and quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the literal. (A "quote" is the character used to open the literal, i.e. either `'` or `"`.)

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 529)

Unknown directive type "index".

```
.. index:: 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
```

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	Notes
`\newline`	Backslash and newline ignored	
`\\`	Backslash (`\`)	
`\'`	Single quote (`'`)	
`\"`	Double quote (`"`)	
`\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`	Character with octal value *ooo*	(1,3)
`\xhh`	Character with hex value *hh*	(2,3)

Escape sequences only recognized in string literals are:

Escape Sequence	Meaning	Notes
`\N{name}`	Character named *name* in the Unicode database	(4)
`\uxxxx`	Character with 16-bit hex value *xxxx*	(5)
`\Uxxxxxxxx`	Character with 32-bit hex value *xxxxxxxx*	(6)

Notes:

1. As in Standard C, up to three octal digits are accepted.
2. Unlike in Standard C, exactly two hex digits are required.
3. In a bytes literal, hexadecimal and octal escapes denote the byte with the given value. In a string literal, these escapes denote a Unicode character with the given value.
4. **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 608)

   Unknown directive type "versionchanged".

   ```
   .. versionchanged:: 3.3
      Support for name aliases [#]_ has been added.
   ```

5. Exactly four hex digits are required.
6. Any Unicode character can be encoded this way. Exactly eight hex digits are required.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 619)

Unknown directive type "index".

```
.. index:: unrecognized escape sequence
```

Unlike Standard C, all unrecognized escape sequences are left in the string unchanged, i.e., *the backslash is left in the result*. (This behavior is useful when debugging: if an escape sequence is mistyped, the resulting output is more easily recognized as broken.) It is also important to note that the escape sequences only recognized in string literals fall into the category of unrecognized escapes for bytes literals.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 628)

Unknown directive type "versionchanged".

```
.. versionchanged:: 3.6
   Unrecognized escape sequences produce a :exc:`DeprecationWarning`.  In
   a future Python version they will be a :exc:`SyntaxWarning` and
   eventually a :exc:`SyntaxError`.
```

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.

String literal concatenation

Multiple adjacent string or bytes literals (delimited by whitespace), possibly using different quoting conventions, are allowed, and their meaning is the same as their concatenation. Thus, `"hello" 'world'` is equivalent to `"helloworld"`. This feature can be used to reduce the number of backslashes needed, to split long strings conveniently across long lines, or even to add comments to parts of strings, for example:

```
re.compile("[A-Za-z_]"       # letter or underscore
           "[A-Za-z0-9_]*"   # letter, digit or underscore
          )
```

Note that this feature is defined at the syntactical level, but implemented at compile time. The '+' operator must be used to concatenate string expressions at run time. Also note that literal concatenation can use different quoting styles for each component (even mixing raw strings and triple quoted strings), and formatted string literals may be concatenated with plain string literals.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 666)

Unknown directive type "index".

```
.. index::
   single: formatted string literal
   single: interpolated string literal
   single: string; formatted literal
   single: string; interpolated literal
   single: f-string
   single: fstring
   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
```

Formatted string literals

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 682)

Unknown directive type "versionadded".

```
.. versionadded:: 3.6
```

A :dfn:`formatted string literal` or :dfn:`f-string` is a string literal that is prefixed with `'f'` or `'F'`. These strings may contain replacement fields, which are expressions delimited by curly braces `{}`. While other string literals always have a constant value, formatted strings are really expressions evaluated at run time.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 684); *backlink*

Unknown interpreted text role "dfn".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 684); *backlink*

Unknown interpreted text role "dfn".

Escape sequences are decoded like in ordinary string literals (except when a literal is also marked as a raw string). After decoding, the grammar for the contents of the string is:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 694)

Unknown directive type "productionlist".

```
.. productionlist:: python-grammar
   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` | NULL | `replacement_field`)*
literal_char: <any code point except "{", "}" or NULL>
```

The parts of the string outside curly braces are treated literally, except that any doubled curly braces `'{{'` or `'}}'` are replaced with the corresponding single curly brace. A single opening curly bracket `'{'` marks a replacement field, which starts with a Python expression. To display both the expression text and its value after evaluation, (useful in debugging), an equal sign `'='` may be added after the expression. A conversion field, introduced by an exclamation point `'!'` may follow. A format specifier may also be appended, introduced by a colon `':'`. A replacement field ends with a closing curly bracket `'}'`.

Expressions in formatted string literals are treated like regular Python expressions surrounded by parentheses, with a few exceptions. An empty expression is not allowed, and both :keyword:`lambda` and assignment expressions `:=` must be surrounded by explicit parentheses. Replacement expressions can contain line breaks (e.g. in triple-quoted strings), but they cannot contain comments. Each expression is evaluated in the context where the formatted string literal appears, in order from left to right.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 714); *backlink*

Unknown interpreted text role "keyword".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 723)

Unknown directive type "versionchanged".

```
.. versionchanged:: 3.7
   Prior to Python 3.7, an :keyword:`await` expression and comprehensions
   containing an :keyword:`async for` clause were illegal in the expressions
   in formatted string literals due to a problem with the implementation.
```

When the equal sign `'='` is provided, the output will have the expression text, the `'='` and the evaluated value. Spaces after the opening brace `'{'`, within the expression and after the `'='` are all retained in the output. By default, the `'='` causes the :func:`repr` of the expression to be provided, unless there is a format specified. When a format is specified it defaults to the :func:`str` of the expression unless a conversion `'!r'` is declared.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 728); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 728); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 736)

Unknown directive type "versionadded".

```
.. versionadded:: 3.8
   The equal sign ``'='``.
```

If a conversion is specified, the result of evaluating the expression is converted before formatting. Conversion `'!s'` calls :func:`str` on the result, `'!r'` calls :func:`repr`, and `'!a'` calls :func:`ascii`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 739); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 739); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 739); *backlink*

Unknown interpreted text role "func".

The result is then formatted using the :func:`format` protocol. The format specifier is passed to the :meth:`__format__` method of the expression or conversion result. An empty string is passed when the format specifier is omitted. The formatted result is then included in the final value of the whole string.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 743); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 743); *backlink*

Unknown interpreted text role "meth".

Top-level format specifiers may include nested replacement fields. These nested fields may include their own conversion fields and :ref:`format specifiers <formatspec>`, but may not include more deeply-nested replacement fields. The :ref:`format specifier mini-language <formatspec>` is the same as that used by the :meth:`str.format` method.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 749); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 749); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 749); *backlink*

Unknown interpreted text role "meth".

Formatted string literals may be concatenated, but replacement fields cannot be split across literals.

Some examples of formatted string literals:

```
>>> 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" '
```

A consequence of sharing the same syntax as regular string literals is that characters in the replacement fields must not conflict with the quoting used in the outer formatted string literal:

```
f"abc {a["x"]} def"    # error: outer string literal ended prematurely
f"abc {a['x']} def"    # workaround: use different quoting
```

Backslashes are not allowed in format expressions and will raise an error:

```
f"newline: {ord('\n')}"  # raises SyntaxError
```

To include a value in which a backslash escape is required, create a temporary variable.

```
>>> newline = ord('\n')
>>> f"newline: {newline}"
'newline: 10'
```

Formatted string literals cannot be used as docstrings, even if they do not include expressions.

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

See also PEP 498 for the proposal that added formatted string literals, and :meth:`str.format`, which uses a related format string mechanism.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 820); *backlink*

Unknown interpreted text role "meth".

Numeric literals

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 829)

Unknown directive type "index".

```
.. index:: number, numeric literal, integer literal
   floating point literal, hexadecimal literal
   octal literal, binary literal, decimal literal, imaginary literal, complex literal
```

There are three types of numeric literals: integers, floating point numbers, and imaginary numbers. There are no complex literals (complex numbers can be formed by adding a real number and an imaginary number).

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 842)

Unknown directive type "index".

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

Integer literals

Integer literals are described by the following lexical definitions:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 855)

Unknown directive type "productionlist".

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

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

Underscores are ignored for determining the numeric value of the literal. They can be used to group digits for enhanced readability. One underscore can occur between digits, and after base specifiers like `0x`.

Note that leading zeros in a non-zero decimal number are not allowed. This is for disambiguation with C-style octal literals, which Python used before version 3.0.

Some examples of integer literals:

```
7     2147483647                        0o177    0b100110111
3     79228162514264337593543950336     0o377    0xdeadbeef
      100_000_000_000                   0b_1110_0101
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 884)

Unknown directive type "versionchanged".

```
.. versionchanged:: 3.6
   Underscores are now allowed for grouping purposes in literals.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 888)

Unknown directive type "index".

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

Floating point literals

Floating point literals are described by the following lexical definitions:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 899)

Unknown directive type "productionlist".

```
.. productionlist:: python-grammar
   floatnumber: `pointfloat` | `exponentfloat`
   pointfloat: [`digitpart`] `fraction` | `digitpart` "."
   exponentfloat: (`digitpart` | `pointfloat`) `exponent`
   digitpart: `digit` (["_"] `digit`)*
   fraction: "." `digitpart`
   exponent: ("e" | "E") ["+" | "-"] `digitpart`
```

Note that the integer and exponent parts are always interpreted using radix 10. For example, `077e010` is legal, and denotes the same number as `77e10`. The allowed range of floating point literals is implementation-dependent. As in integer literals, underscores are supported for digit grouping.

Some examples of floating point literals:

```
3.14    10.    .001    1e100    3.14e-10    0e0    3.14_15_93
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 916)

Unknown directive type "versionchanged".

```
.. versionchanged:: 3.6
   Underscores are now allowed for grouping purposes in literals.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 920)

Unknown directive type "index".

```
.. index::
   single: j; in numeric literal
```

Imaginary literals

Imaginary literals are described by the following lexical definitions:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 929)

Unknown directive type "productionlist".

```
.. productionlist:: python-grammar
   imagnumber: (`floatnumber` | `digitpart`) ("j" | "J")
```

An imaginary literal yields a complex number with a real part of 0.0. Complex numbers are represented as a pair of floating point numbers and have the same restrictions on their range. To create a complex number with a nonzero real part, add a floating point number to it, e.g., `(3+4j)`. Some examples of imaginary literals:

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

Operators

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 946)

Unknown directive type "index".

```
.. index:: single: operators
```

The following tokens are operators:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 950)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

   +       -       *       **      /       //      %      @
   <<      >>      &       |       ^       ~       :=
   <       >       <=      >=      ==      !=
```

Delimiters

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 963)

Unknown directive type "index".

```
.. index:: single: delimiters
```

The following tokens serve as delimiters in the grammar:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 967)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

   (       )       [       ]       {       }
   ,       :       .       ;       @       =       ->
   +=      -=      *=      /=      //=     %=      @=
   &=      |=      ^=      >>=     <<=     **=
```

The period can also occur in floating-point and imaginary literals. A sequence of three periods has a special meaning as an ellipsis literal. The second half of the list, the augmented assignment operators, serve lexically as delimiters, but also perform an operation.

The following printing ASCII characters have special meaning as part of other tokens or are otherwise significant to the lexical analyzer:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 982)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

   '       "       #       \
```

The following printing ASCII characters are not used in Python. Their occurrence outside string literals and comments is an unconditional error:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\reference\(cpython-main)(Doc)(reference)lexical_analysis.rst, line 989)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

   $       ?       `
```

Footnotes

[1] https://www.unicode.org/Public/11.0.0/ucd/NameAliases.txt