Chapter 2: Lexical analysis

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 uses the (7-bit) ASCII character set for program text and string literals. 8-bit characters may be used in string literals and comments but their interpretation is platform dependent; the proper way to insert 8-bit characters in string literals is by using octal or hexadecimal escape sequences.

The run-time character set depends on the I/O devices connected to the program but is generally a superset of ASCII.

2.1 Line structure

A Python program is divided in a number of logical lines.

2.1.1 Logical lines

The end of each 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.

2.1.2 Physical lines

A physical line ends in whatever the current platform's convention is for terminating lines. On UNIX, this is the ASCII LF (linefeed) character. On DOS/Windows, it is the ASCII sequence CR LF (return followed by linefeed). On Macintosh, it is the ASCII CR (return) character.

2.1.3 Comments

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≠≠; they are not tokens.

2.1.4 Explicit line joining

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.

2.1.5 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. Implicit continued lines can also occur within triple-quoted strings (see below); in that case they cannot carry comments.

2.1.6 Blank lines

A logical line that contains only spaces, tabs, formfeeds, and possibly a comment, is ignored (i.e., no NEWLINE token is generated), except that during interactive input of statements, an entirely blank logical line (i.e. one containing not even whitespace or a comment) terminates a multi-line statement.

2.1.7 Indentation

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.

First, tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to there 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.

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

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.)

2.1.8 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).

2.2 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.

2.3 Identifiers and keywords

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

```
identifier:       (letter|"_") (letter|digit|"_")*
letter:            lowercase | uppercase
lowercase:        "a"..."z"
uppercase:        "A"..."Z"
digit:             "0"..."9"
```

Identifiers are unlimited in length. Case is significant.

2.3.1 Keywords

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:

```
and           elif          global       not           try
break        else          if            or            while
class        except       import       pass
continue    finally     in            print
def           for           is             raise
del           from         lambda       return
```

2.3.2 Reserved classes of identifiers

Certain classes of identifiers (besides keywords) have special meanings. These are:

Table 1: Special Meanings of Identifiers

Form	Meaning
XE "import"XE "from" _*	Not imported by from *module* import *
__*__	System-defined name
XE "name:mangling" __*	Class-private name mangling

(XXX need section references here.)

2.4 Literals

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

2.4.1 String literals

String literals are described by the following lexical definitions:

```
stringliteral:    shortstring | longstring
shortstring:       "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring:        "'''" longstringitem* "'''" | '"""' longstringitem*
'"""'
shortstringitem: shortstringchar | escapeseq
longstringitem:   longstringchar | escapeseq
shortstringchar: <any ASCII character except "\" or newline or the quote>
longstringchar:   <any ASCII character except "\">
escapeseq:         "\" <any ASCII character>
```

In plain English: String literals can be enclosed in single quotes (') or double quotes ("). They can also be enclosed in 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.

In “long strings” (strings surrounded by sets of three quotes), unescaped newlines and quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the string. (A “quote” is the character used to open the string, i.e. either ' or ".)

Escape sequences in strings are interpreted according to rules similar to those used by Standard C. The recognized escape sequences are:

Table 2: Escape Sequences

Escape Sequence	Meaning
newline	Ignored
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
XE "ASCII"\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	ASCII character with octal value *ooo*
\x*xx...*	ASCII character with hex value *xx...*

In strict compatibility with Standard C, up to three octal digits are accepted, but an unlimited number of hex digits is taken to be part of the hex escape (and then the lower 8 bits of the resulting hex number are used in all current implementations...).

Unlike Standard C, all unrecognized escape sequences are left in the string unchanged, i.e.,

the backslash is left in the string. (This behavior is useful when debugging: if an escape sequence is mistyped, the resulting output is more easily recognized as broken. It also helps a great deal for string literals used as regular expressions or otherwise passed to other modules that do their own escape handling.)

2.4.1.1 String literal concatenation

Multiple adjacent string 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:

```
regex.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.

2.4.2 Numeric literals

There are four types of numeric literals: plain integers, long integers, floating point numbers, and imaginary numbers.

2.4.2.1 Integer and long integer literals

Integer and long integer literals are described by the following lexical definitions:

```
longinteger:     integer ("l"|"L")
integer:          decimalinteger | octinteger | hexinteger
decimalinteger: nonzerodigit digit* | "0"
octinteger:      "0" octdigit+
hexinteger:      "0" ("x"|"X") hexdigit+
nonzerodigit:    "1"..."9"
octdigit:         "0"..."7"
hexdigit:         digit|"a"..."f"|"A"..."F"
```

Although both lower case 'l' and upper case 'L' are allowed as suffix for long integers, it is strongly recommended to always use 'L', since the letter 'l' looks too much like the digit '1'.

Plain integer decimal literals must be at most 2147483647 (i.e., the largest positive integer, using 32-bit arithmetic). Plain octal and hexadecimal literals may be as large as 4294967295, but values larger than 2147483647 are converted to a negative value by subtracting 4294967296. There is no limit for long integer literals apart from what can be stored in available memory.

Some examples of plain and long integer literals:

```
7       2147483647                                  0177     0x80000000
3L      79228162514264337593543950336L     0377L    0x100000000L
```

2.4.2.2 Floating point literals

Floating point literals are described by the following lexical definitions:

```
floatnumber:     pointfloat | exponentfloat
pointfloat:       [intpart] fraction | intpart "."
exponentfloat:   (intpart | pointfloat) exponent
intpart:          digit+
fraction:         "." digit+
exponent:         ("e"|"E") ["+"|"-"] digit+
```

The allowed range of floating point literals is implementation-dependent. Some examples of floating point literals:

```
3.14     10.      .001     1e100    3.14e-10
```

2.4.2.3 Imaginary literals

Imaginary literals are described by the following lexical definitions:

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

An imaginary literals 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     10 j     .001j    1e100j   3.14e-10j
```

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`.

2.5 Operators

The following tokens are operators:

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

The comparison operators `<>` and `!=` are alternate spellings of the same operator; != is the preferred spelling, <> is obsolescent.

2.6 Delimiters

The following tokens serve as delimiters in the grammar:

```
(          )          [          ]          {          }
,          :          .          `          =          ;
```

The period can also occur in floating-point and imaginary literals. A sequence of three periods has a special meaning as ellipses in slices.

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

```
'           "           #           \
```

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

```
@           $           ?
```