Python Language Reference Manual

Release 2.5

Guido van Rossum
Fred L. Drake, Jr., editor

Python Software Foundation
Email: docs@python.org

A catalogue record for this book is available from the British Library.

First printing, August 2003 (23/8/2003).
Second printing (revised for version 2.5), November 2006 (3/11/2006).

Published by Network Theory Limited.

15 Royal Park
Bristol
BS8 3AL
United Kingdom

Email: info@network-theory.co.uk

ISBN 0-9541617-8-5

Further information about this book is available from
`http://www.network-theory.co.uk/python/language/`

Summary of changes made for this edition by Network Theory Ltd:

Minor editing of text and examples for layout in book format. Added a Publisher's preface. Renamed "Abstract" to "Overview". Some paragraphs and sentences edited or removed for conciseness. A complete set of differences can be found at
http://www.network-theory.co.uk/python/language/src/

Contents

Publisher's Preface

This manual is part of the official reference documentation for Python, an object-oriented programming language created by Guido van Rossum.

Python is *free software*. The term "free software" refers to your freedom to run, copy, distribute, study, change and improve the software. With Python you have all these freedoms.

You can support free software by becoming an associate member of the Free Software Foundation. The Free Software Foundation is a tax-exempt charity dedicated to promoting the right to use, study, copy, modify, and redistribute computer programs. It also helps to spread awareness of the ethical and political issues of freedom in the use of software. For more information visit the website `www.fsf.org`.

The development of Python itself is supported by the Python Software Foundation. Companies using Python can invest in the language by becoming sponsoring members of this group. Donations can also be made online through the Python website. Further information is available at `http://www.python.org/psf/`

Brian Gough
Publisher
November 2006

Overview

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for rapid application development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

This reference manual describes the syntax and "core semantics" of the language. It is terse, but attempts to be exact and complete. The semantics of non-essential built-in object types and of the built-in functions and modules are described in the *Python Library Reference Manual.* For an informal introduction to the language, see *An Introduction to Python.* For C or C++ programmers, two additional manuals exist: *Extending and Embedding the Python Interpreter* describes the high-level picture of how to write a Python extension module, and the *Python/C API Reference Manual* describes the interfaces available to C/C++ programmers in detail.

1 Introduction

This reference manual describes the Python programming language. It is not intended as a tutorial.

While I am trying to be as precise as possible, I chose to use English rather than formal specifications for everything except syntax and lexical analysis. This should make the document more understandable to the average reader, but will leave room for ambiguities. Consequently, if you were coming from Mars and tried to re-implement Python from this document alone, you might have to guess things and in fact you would probably end up implementing quite a different language. On the other hand, if you are using Python and wonder what the precise rules about a particular area of the language are, you should definitely be able to find them here. If you would like to see a more formal definition of the language, maybe you could volunteer your time—or invent a cloning machine.

It is dangerous to add too many implementation details to a language reference document—the implementation may change, and other implementations of the same language may work differently. On the other hand, there is currently only one Python implementation in widespread use (although alternate implementations exist), and its particular quirks are sometimes worth being mentioned, especially where the implementation imposes additional limitations. Therefore, you'll find short "implementation notes" sprinkled throughout the text.

Every Python implementation comes with a number of built-in and standard modules. These are not documented here, but in the separate *Python Library Reference Manual* document. A few built-in modules are mentioned when they interact in a significant way with the language definition.

1.1 Alternate Implementations

Though there is one Python implementation which is by far the most popular, there are some alternate implementations which are of particular interest to different audiences.

Known implementations include:

CPython This is the original and most-maintained implementation of Python, written in C. New language features generally appear here first.

Jython Python implemented in Java. This implementation can be used as a scripting language for Java applications, or can be used to create applications using the Java class libraries. It is also often used to create tests for Java libraries. More information can be found at the Jython website.[1]

Python for .NET This implementation actually uses the CPython implementation, but is a managed .NET application and makes .NET libraries available. This was created by Brian Lloyd. For more information, see the Python for .NET home page.[2]

IronPython An alternate Python for .NET. Unlike Python.NET, this is a complete Python implementation that generates IL, and compiles Python code directly to .NET assemblies. It was created by Jim Hugunin, the original creator of Jython. For more information, see the IronPython website.[3]

PyPy An implementation of Python written in Python; even the bytecode interpreter is written in Python. This is executed using CPython as the underlying interpreter. One of the goals of the project is to encourage experimentation with the language itself by making it easier to modify the interpreter (since it is written in Python). Additional information is available on the PyPy project's home page.[4]

Each of these implementations varies in some way from the language as documented in this manual, or introduces specific information beyond what's covered in the standard Python documentation. Please refer to the implementation-specific documentation to determine what else you need to know about the specific implementation you're using.

1.2 Notation

The descriptions of lexical analysis and syntax use a modified BNF grammar notation. This uses the following style of definition:

[1] `http://www.jython.org/`
[2] `http://www.zope.org/Members/Brian/PythonNet`
[3] `http://workspaces.gotdotnet.com/ironpython`
[4] `http://codespeak.net/pypy/`

```
name       ::=  lc_letter (lc_letter | "_")*
lc_letter  ::=  "a"..."z"
```

The first line says that a `name` is an `lc_letter` followed by a sequence of zero or more `lc_letter`s and underscores. An `lc_letter` in turn is any of the single characters '`a`' through '`z`'. (This rule is actually adhered to for the names defined in lexical and grammar rules in this document.)

Each rule begins with a name (which is the name defined by the rule) and `::=`. A vertical bar (`|`) is used to separate alternatives; it is the least binding operator in this notation. A star (`*`) means zero or more repetitions of the preceding item; likewise, a plus (`+`) means one or more repetitions, and a phrase enclosed in square brackets (`[ ]`) means zero or one occurrences (in other words, the enclosed phrase is optional). The `*` and `+` operators bind as tightly as possible; parentheses are used for grouping. Literal strings are enclosed in quotes. White space is only meaningful to separate tokens. Rules are normally contained on a single line; rules with many alternatives may be formatted alternatively with each line after the first beginning with a vertical bar.

In lexical definitions (as the example above), two more conventions are used: Two literal characters separated by three dots mean a choice of any single character in the given (inclusive) range of ASCII characters. A phrase between angular brackets (`<...>`) gives an informal description of the symbol defined; e.g., this could be used to describe the notion of 'control character' if needed.

Even though the notation used is almost the same, there is a big difference between the meaning of lexical and syntactic definitions: a lexical definition operates on the individual characters of the input source, while a syntax definition operates on the stream of tokens generated by the lexical analysis. All uses of BNF in the next chapter ("Lexical Analysis") are lexical definitions; uses in subsequent chapters are syntactic definitions.

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. In Python versions 2.3 and later, an encoding declaration can be used to indicate that string literals and comments use an encoding different from ASCII. For compatibility with older versions, Python only warns if it finds 8-bit characters; those warnings should be corrected by either declaring an explicit encoding, or using escape sequences if those bytes are binary data, instead of characters.

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

Future compatibility note: It may be tempting to assume that the character set for 8-bit characters is ISO Latin-1 (an ASCII superset that covers most western languages that use the Latin alphabet), but it is possible that in the future Unicode text editors will become common. These generally use the UTF-8 encoding, which is also an ASCII superset, but with very different use for the characters with ordinals 128-255. While there is no consensus on this subject yet, it is unwise to assume either Latin-1 or UTF-8, even though the current implementation appears to favor Latin-1. This applies both to the source character set and the run-time character set.

2.1 Line structure

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

2.1.1 Logical lines

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.

2.1.2 Physical lines

A physical line is a sequence of characters terminated by an end-of-line sequence. In source files, 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 Macintosh form using the ASCII `CR` (return) character. All of these forms can be used equally, regardless of platform.

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).

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 Encoding declarations

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

If an encoding is declared, the encoding name must be recognized by Python. The encoding is used for all lexical analysis, in particular to find the end of a string, and to interpret the contents of Unicode literals. String literals are converted to Unicode for syntactical analysis, then converted back to their original encoding before interpretation starts. The encoding declaration must appear on a line of its own.

2.1.5 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:
        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.

2.1.6 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',
               'April',   'Mei',      'Juni',
               'Juli',    'Augustus', 'September',
               'Oktober', 'November', 'December']
```

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.

2.1.7 Blank lines

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 implementation, an entirely blank logical line (i.e. one containing not even whitespace or a comment) terminates a multi-line statement.

2.1.8 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 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.

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).

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.)

2.1.9 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       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

In version 2.4, `None` became a constant and is now recognized by the compiler as a name for the built-in object `None`. Although it is not a keyword, you cannot assign a different object to it.

In version 2.5, both `as` and `with` are only recognized when the `with_statement` future feature has been enabled. It will always be enabled in Python 2.6. See section 7.5 for details. Note that using `as` and `with` as identifiers will always issue a warning, even when the `with_statement` future directive is not in effect.

2.3.2 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 *`'. The special identifier '`_`' is used in the interactive interpreter to store the result of the last evaluation; it is stored in the `__builtin__` module. When not in interactive mode, '`_`' has no special meaning and is not defined. See section 6.12, "The `import` statement."

Note: The name '_' is often used in conjunction with internationalization; refer to the documentation for the `gettext` module for more information on this convention.

`__*__` System-defined names. These names are defined by the interpreter and its implementation (including the standard library); applications should not expect to define additional names using this convention. The set of names of this class defined by Python may be extended in future versions. See section 3.4, "Special method names."

`__*` 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 5.2.1, "Identifiers (Names)."

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     ::=  [ stringprefix ]( shortstring
                       | longstring )
stringprefix      ::=  "r" | "u" | "ur"
                       | "R" | "U" | "UR"
                       | "Ur" | "uR"
shortstring       ::=  "’" shortstringitem* "’"
                       | ’"’ shortstringitem* ’"’
longstring        ::=  "’’’" longstringitem* "’’’"
                       | ’"""’ longstringitem* ’"""’
shortstringitem   ::=  shortstringchar | escapeseq
longstringitem    ::=  longstringchar | escapeseq
shortstringchar   ::=  <any source character except "\"
                       or newline or the quote>
longstringchar    ::=  <any source character except "\">
escapeseq         ::=  "\" <any ASCII character>
```

One syntactic restriction not indicated by these productions is that whitespace is not allowed between the `stringprefix` and the rest of the string literal. The source character set is defined by the encoding declaration;

it is ASCII if no encoding declaration is given in the source file; see section 2.1.4.

In plain English: String 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. String literals may optionally be prefixed with a letter '`r`' or '`R`'; such strings are called *raw strings* and use different rules for interpreting backslash escape sequences. A prefix of '`u`' or '`U`' makes the string a Unicode string. Unicode strings use the Unicode character set as defined by the Unicode Consortium and ISO 10646. Some additional escape sequences, described below, are available in Unicode strings. The two prefix characters may be combined; in this case, '`u`' must appear before '`r`'.

In triple-quoted strings, 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 `"`.)

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

Escape Sequence	Meaning	Notes
`\`*newline*	Ignored	
`\\`	Backslash (`\`)	
`\'`	Single quote (`'`)	
`\"`	Double quote (`"`)	
`\a`	ASCII Bell (BEL)	
`\b`	ASCII Backspace (BS)	
`\f`	ASCII Formfeed (FF)	
`\n`	ASCII Linefeed (LF)	
`\N{`*name*`}`	Character named *name* in the Unicode database (Unicode only)	
`\r`	ASCII Carriage Return (CR)	
`\t`	ASCII Horizontal Tab (TAB)	
`\u`*xxxx*	Character with 16-bit hex value *xxxx* (Unicode only)	(1)
`\U`*xxxxxxxx*	Character with 32-bit hex value *xxxxxxxx* (Unicode only)	(2)
`\v`	ASCII Vertical Tab (VT)	
`\`*ooo*	Character with octal value *ooo*	(3,5)
`\x`*hh*	Character with hex value *hh*	(4,5)

Notes:

(1) Individual code units which form parts of a surrogate pair can be encoded using this escape sequence.

(2) Any Unicode character can be encoded this way, but characters outside the Basic Multilingual Plane (BMP) will be encoded using a surrogate pair if Python is compiled to use 16-bit code units (the default). Individual code units which form parts of a surrogate pair can be encoded using this escape sequence.

(3) As in Standard C, up to three octal digits are accepted.

(4) Unlike in Standard C, at most two hex digits are accepted.

(5) In a string literal, hexadecimal and octal escapes denote the byte with the given value; it is not necessary that the byte encodes a character in the source character set. In a Unicode literal, these escapes denote a Unicode character with the given value.

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 is also important to note that the escape sequences marked as "(Unicode only)" in the table above fall into the category of unrecognized escapes for non-Unicode string literals.

When an '`r`' or '`R`' prefix is present, a character following a backslash is included in the string without change, and *all backslashes are left in the string.* For example, the string literal `r"\n"` consists of two characters: a backslash and a lowercase '`n`'. String quotes can be escaped with a backslash, but the backslash remains in the string; 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 string 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 string, *not* as a line continuation.

When an '`r`' or '`R`' prefix is used in conjunction with a '`u`' or '`U`' prefix, then the `\uXXXX` and `\UXXXXXXXX` escape sequences are processed while *all other backslashes are left in the string.* For example, the string literal `ur"\u0062\n"` consists of three Unicode characters: 'LATIN SMALL LETTER B', 'REVERSE SOLIDUS', and 'LATIN SMALL LETTER N'. Backslashes can be escaped with a preceding backslash; however, both remain in the string. As a result, `\uXXXX` escape sequences are only recognized when there are an odd number of backslashes.

2.4.2 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:

```
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).

2.4.3 Numeric literals

There are four types of numeric literals: plain integers, long 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`.

2.4.4 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 literals that are greater than the largest representable plain integer (e.g., 2147483647 when using 32-bit arithmetic) are accepted as if they were long integers instead.[1] There is no limit for long integer literals apart from what can be stored in available memory.

Some examples of plain integer literals (first row) and long integer literals (second and third rows):

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

[1]In versions of Python prior to 2.4, octal and hexadecimal literals in the range just above the largest representable plain integer but below the largest unsigned 32-bit number (on a machine using 32-bit arithmetic), 4294967296, were taken as the negative plain integer obtained by subtracting 4294967296 from their unsigned value.

2.4.5 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+
```

Note that the integer and exponent parts of floating point numbers can look like octal integers, but are 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. Some examples of floating point literals:

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

Note that floating-point literals do not include a sign; a phrase like `-1.0` is actually an expression composed of the unary operator `-` and the literal `1.0`.

2.4.6 Imaginary literals

Imaginary literals are described by the following lexical definitions:

```
imagnumber  ::=  (floatnumber | intpart) ("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
```

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 an ellipsis in slices. 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:

```
'       "       #       \
```

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

```
$       ?
```

3 Data model

3.1 Objects, values and types

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer," code is also represented by objects.)

Every object has an identity, a type and a value. An object's *identity* never changes once it has been created; you may think of it as the object's address in memory. The '`is`' operator compares the identity of two objects; the `id()` function returns an integer representing its identity (currently implemented as its address). An object's *type* is also unchangeable.[1] An object's type determines the operations that the object supports (e.g., "does it have a length?") and also defines the possible values for objects of that type. The `type()` function returns an object's type (which is an object itself). The *value* of some objects can change. Objects whose value can change are said to be *mutable*; objects whose value is unchangeable once they are created are called *immutable*. (The value of an immutable container object that contains a reference to a mutable object can change when the latter's value is changed; however the container is still considered immutable, because the collection of objects it contains cannot be changed. So, immutability is not strictly the same as having an unchangeable value, it is more subtle.) An object's mutability is determined by its type; for instance, numbers, strings and tuples are immutable, while dictionaries and lists are mutable.

Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected. An implementation is allowed to postpone garbage collection or omit it altogether—it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable. (Implementation note: the current implementation uses a reference-counting scheme with (optional)

[1]Since Python 2.2, a gradual merging of types and classes has been started that makes this and a few other assertions made in this manual not 100% accurate and complete: for example, it *is* now possible in some cases to change an object's type, under certain controlled conditions. Until this manual undergoes extensive revision, it must now be taken as authoritative only regarding "classic classes", that are still the default, for compatibility purposes, in Python 2.2 and 2.3. For more information, see `http://www.python.org/doc/newstyle.html`.

delayed detection of cyclically linked garbage, which collects most objects as soon as they become unreachable, but is not guaranteed to collect garbage containing circular references. See the *Python Library Reference Manual* for information on controlling the collection of cyclic garbage.)

Note that the use of the implementation's tracing or debugging facilities may keep objects alive that would normally be collectable. Also note that catching an exception with a '`try` ... `except`' statement may keep objects alive.

Some objects contain references to "external" resources such as open files or windows. It is understood that these resources are freed when the object is garbage-collected, but since garbage collection is not guaranteed to happen, such objects also provide an explicit way to release the external resource, usually a `close()` method. Programs are strongly recommended to explicitly close such objects. The '`try` ... `finally`' statement provides a convenient way to do this.

Some objects contain references to other objects; these are called *containers*. Examples of containers are tuples, lists and dictionaries. The references are part of a container's value. In most cases, when we talk about the value of a container, we imply the values, not the identities of the contained objects; however, when we talk about the mutability of a container, only the identities of the immediately contained objects are implied. So, if an immutable container (like a tuple) contains a reference to a mutable object, its value changes if that mutable object is changed.

Types affect almost all aspects of object behavior. Even the importance of object identity is affected in some sense: for immutable types, operations that compute new values may actually return a reference to any existing object with the same type and value, while for mutable objects this is not allowed. E.g., after '`a = 1; b = 1`', `a` and `b` may or may not refer to the same object with the value one, depending on the implementation, but after '`c = []; d = []`', `c` and `d` are guaranteed to refer to two different, unique, newly created empty lists. (Note that '`c = d = []`' assigns the same object to both `c` and `d`.)

3.2 The standard type hierarchy

Below is a list of the types that are built into Python. Extension modules (written in C, Java, or other languages, depending on the implementation) can define additional types. Future versions of Python may add types to the type hierarchy (e.g., rational numbers, efficiently stored arrays of integers, etc.).

Some of the type descriptions below contain a paragraph listing 'special attributes.' These are attributes that provide access to the implementation and are not intended for general use. Their definition may change in the future.

None This type has a single value. There is a single object with this value. This object is accessed through the built-in name `None`. It is used to signify the absence of a value in many situations, e.g., it is returned from functions that don't explicitly return anything. Its truth value is false.

NotImplemented This type has a single value. There is a single object with this value. This object is accessed through the built-in name `NotImplemented`. Numeric methods and rich comparison methods may return this value if they do not implement the operation for the operands provided. (The interpreter will then try the reflected operation, or some other fallback, depending on the operator.) Its truth value is true.

Ellipsis This type has a single value. There is a single object with this value. This object is accessed through the built-in name `Ellipsis`. It is used to indicate the presence of the '`...`' syntax in a slice. Its truth value is true.

Numbers These are created by numeric literals and returned as results by arithmetic operators and arithmetic built-in functions. Numeric objects are immutable; once created their value never changes. Python numbers are of course strongly related to mathematical numbers, but subject to the limitations of numerical representation in computers.

Python distinguishes between integers, floating point numbers, and complex numbers:

Integers These represent elements from the mathematical set of integers (positive and negative).

There are three types of integers:

Plain integers These represent whole numbers in the range -2147483648 through 2147483647. (The range may be larger on machines with a larger natural word size, but not smaller.) When the result of an operation would fall outside this range, the result is normally returned as a long integer (in some cases, the exception `OverflowError` is raised instead). For the purpose of shift and mask operations, integers are assumed to have a binary, 2's complement notation using 32 or more bits, and hiding no bits from the user (i.e., all 4294967296 different bit patterns correspond to different values).

Long integers These represent numbers in an unlimited range, subject to available (virtual) memory only. For the purpose of shift and mask operations, a binary representation is assumed, and negative numbers are represented in a variant of 2's complement which gives the illusion of an infinite string of sign bits extending to the left.

Booleans These represent the truth values False and True. The two objects representing the values False and True are the only Boolean objects. The Boolean type is a subtype of plain integers, and Boolean values behave like the values 0 and 1, respectively, in almost all contexts, the exception being that when converted to a string, the strings `"False"` or `"True"` are returned, respectively.

The rules for integer representation are intended to give the most meaningful interpretation of shift and mask operations involving negative integers and the least surprises when switching between the plain and long integer domains. Any operation except left shift, if it yields a result in the plain integer domain without causing overflow, will yield the same result in the long integer domain or when using mixed operands.

Floating point numbers These represent machine-level double-precision floating point numbers. You are at the mercy of the underlying machine architecture (and C or Java implementation) for the accepted range and handling of overflow. Python does not support single-precision floating point numbers; the savings in processor and memory usage that are usually the reason for using these is dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating point numbers.

Complex numbers These represent complex numbers as a pair of machine-level double-precision floating point numbers. The same caveats apply as for floating point numbers. The real and imaginary parts of a complex number `z` can be retrieved through the read-only attributes `z.real` and `z.imag`.

Sequences These represent finite ordered sets indexed by non-negative numbers. The built-in function `len()` returns the number of items of a sequence. When the length of a sequence is n, the index set contains the numbers 0, 1, . . . , $n-1$. Item i of sequence a is selected by a`[`i`]`.

Index