Python Reference Manual

Release 2.5.3

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

19th December, 2008

Python Software Foundation
Email: docs@python.org

Abstract

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*. For an informal introduction to the language, see the *Python Tutorial*. 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.

CONTENTS

CHAPTER

ONE

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* 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.

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.

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.

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.

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:

```
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.

CHAPTER

TWO

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. New in version 2.3: 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:   # 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.

2.1.6 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',      # 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.

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
```

Changed 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.

Changed 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, exactly two hex digits are required.

(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 above 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
```

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

2.4.6 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

[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.

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:

```
$       ?
```

CHAPTER

THREE

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) 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* 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

[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/.

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 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`]`.

Sequences also support slicing: a`[`i`:`j`]` selects all items with index k such that $i <= k < j$. When used as an expression, a slice is a sequence of the same type. This implies that the index set is renumbered so that it starts at 0.

Some sequences also support "extended slicing" with a third "step" parameter: a`[`i`:`j`:`k`]` selects all items of a with index x where $x = i + n*k$, $n >= 0$ and $i <= x < j$.

Sequences are distinguished according to their mutability:

Immutable sequences An object of an immutable sequence type cannot change once it is created. (If the object contains references to other objects, these other objects may be mutable and may be changed; however, the collection of objects directly referenced by an immutable object cannot change.)

The following types are immutable sequences:

Strings The items of a string are characters. There is no separate character type; a character is represented by a string of one item. Characters represent (at least) 8-bit bytes. The built-in functions `chr()` and `ord()` convert between characters and nonnegative integers representing the byte values. Bytes with the values 0-127 usually represent the corresponding ASCII values, but the interpretation of values is up to the program. The string data type is also used to represent arrays of bytes, e.g., to hold data read from a file.
(On systems whose native character set is not ASCII, strings may use EBCDIC in their internal representation, provided the functions `chr()` and `ord()` implement a mapping between ASCII and EBCDIC, and string comparison preserves the ASCII order. Or perhaps someone can propose a better rule?)

Unicode The items of a Unicode object are Unicode code units. A Unicode code unit is represented by a Unicode object of one item and can hold either a 16-bit or 32-bit value representing a Unicode ordinal (the maximum value for the ordinal is given in `sys.maxunicode`, and depends on how Python is configured at compile time). Surrogate pairs may be present in the Unicode object, and will be reported as two separate items. The built-in functions `unichr()` and `ord()` convert between code units and nonnegative integers representing the Unicode ordinals as defined in the Unicode Standard 3.0. Conversion from and to other encodings are possible through the Unicode method `encode()` and the built-in function `unicode()`.

Tuples The items of a tuple are arbitrary Python objects. Tuples of two or more items are formed by comma-separated lists of expressions. A tuple of one item (a 'singleton') can be formed by affixing a comma to an expression (an expression by itself does not create a tuple, since parentheses must be usable for grouping of expressions). An empty tuple can be formed by an empty pair of parentheses.

Mutable sequences Mutable sequences can be changed after they are created. The subscription and slicing notations can be used as the target of assignment and `del` (delete) statements.

There is currently a single intrinsic mutable sequence type:

Lists The items of a list are arbitrary Python objects. Lists are formed by placing a comma-separated list of expressions in square brackets. (Note that there are no special cases needed to form lists of length 0 or 1.)

The extension module `array` provides an additional example of a mutable sequence type.

Set types These represent unordered, finite sets of unique, immutable objects. As such, they cannot be indexed by any subscript. However, they can be iterated over, and the built-in function `len()` returns the number of items in a set. Common uses for sets are fast membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.

For set elements, the same immutability rules apply as for dictionary keys. Note that numeric types obey the normal rules for numeric comparison: if two numbers compare equal (e.g., `1` and `1.0`), only one of them can be contained in a set.

There are currently two intrinsic set types:

Sets These represent a mutable set. They are created by the built-in `set()` constructor and can be modified afterwards by several methods, such as `add()`.

Frozen sets These represent an immutable set. They are created by the built-in `frozenset()` constructor. As a frozenset is immutable and hashable, it can be used again as an element of another set, or as a dictionary key.

Mappings These represent finite sets of objects indexed by arbitrary index sets. The subscript notation `a[k]` selects the item indexed by `k` from the mapping `a`; this can be used in expressions and as the target of assignments or `del` statements. The built-in function `len()` returns the number of items in a mapping.

There is currently a single intrinsic mapping type:

Dictionaries These represent finite sets of objects indexed by nearly arbitrary values. The only types of values not acceptable as keys are values containing lists or dictionaries or other mutable types that are compared by value rather than by object identity, the reason being that the efficient implementation of dictionaries requires a key's hash value to remain constant. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (e.g., `1` and `1.0`) then they can be used interchangeably to index the same dictionary entry.

Dictionaries are mutable; they can be created by the `{...}` notation (see section 5.2.6, "Dictionary Displays").

The extension modules `dbm`, `gdbm`, and `bsddb` provide additional examples of mapping types.

Callable types These are the types to which the function call operation (see section 5.3.4, "Calls") can be applied:

User-defined functions A user-defined function object is created by a function definition (see section 7.6, "Function definitions"). It should be called with an argument list containing the same number of items as the function's formal parameter list.

Special attributes:

Attribute	Meaning
func_doc	The function's documentation string, or None if unavailable
__doc__	Another way of spelling func_doc
func_name	The function's name
__name__	Another way of spelling func_name
__module__	The name of the module the function was defined in, or None if unavailable.
func_defaults	A tuple containing default argument values for those arguments that have defaults, or None if no ar
func_code	The code object representing the compiled function body.
func_globals	A reference to the dictionary that holds the function's global variables — the global namespace of th
func_dict	The namespace supporting arbitrary function attributes.
func_closure	None or a tuple of cells that contain bindings for the function's free variables.

Most of the attributes labelled "Writable" check the type of the assigned value.

Changed in version 2.4: func_name is now writable.

Function objects also support getting and setting arbitrary attributes, which can be used, for example, to attach metadata to functions. Regular attribute dot-notation is used to get and set such attributes. *Note that the current implementation only supports function attributes on user-defined functions. Function attributes on built-in functions may be supported in the future.*

Additional information about a function's definition can be retrieved from its code object; see the description of internal types below.

User-defined methods A user-defined method object combines a class, a class instance (or None) and any callable object (normally a user-defined function).

Special read-only attributes: im_self is the class instance object, im_func is the function object; im_class is the class of im_self for bound methods or the class that asked for the method for unbound methods; __doc__ is the method's documentation (same as im_func.__doc__); __name__ is the method name (same as im_func.__name__); __module__ is the name of the module the method was defined in, or None if unavailable. Changed in version 2.2: im_self used to refer to the class that defined the method.

Methods also support accessing (but not setting) the arbitrary function attributes on the underlying function object.

User-defined method objects may be created when getting an attribute of a class (perhaps via an instance of that class), if that attribute is a user-defined function object, an unbound user-defined method object, or a class method object. When the attribute is a user-defined method object, a new method object is only created if the class from which it is being retrieved is the same as, or a derived class of, the class stored in the original method object; otherwise, the original method object is used as it is.

When a user-defined method object is created by retrieving a user-defined function object from a class, its im_self attribute is None and the method object is said to be unbound. When one is created by retrieving a user-defined function object from a class via one of its instances, its im_self attribute is the instance, and the method object is said to be bound. In either case, the new method's im_class attribute is the class from which the retrieval takes place, and its im_func attribute is the original function object.

When a user-defined method object is created by retrieving another method object from a class or instance, the behaviour is the same as for a function object, except that the im_func attribute of the new instance is not the original method object but its im_func attribute.

When a user-defined method object is created by retrieving a class method object from a class or instance, its im_self attribute is the class itself (the same as the im_class attribute), and its im_func attribute is the function object underlying the class method.

When an unbound user-defined method object is called, the underlying function (im_func) is called, with the restriction that the first argument must be an instance of the proper class (im_class) or of a derived class thereof.

When a bound user-defined method object is called, the underlying function (im_func) is called, inserting the class instance (im_self) in front of the argument list. For instance, when C is a class which contains a definition for a function `f()`, and `x` is an instance of `C`, calling `x.f(1)` is equivalent to calling `C.f(x, 1)`.

When a user-defined method object is derived from a class method object, the "class instance" stored in `im_self` will actually be the class itself, so that calling either `x.f(1)` or `C.f(1)` is equivalent to calling `f(C,1)` where `f` is the underlying function.

Note that the transformation from function object to (unbound or bound) method object happens each time the attribute is retrieved from the class or instance. In some cases, a fruitful optimization is to assign the attribute to a local variable and call that local variable. Also notice that this transformation only happens for user-defined functions; other callable objects (and all non-callable objects) are retrieved without transformation. It is also important to note that user-defined functions which are attributes of a class instance are not converted to bound methods; this *only* happens when the function is an attribute of the class.

Generator functions A function or method which uses the `yield` statement (see section 6.8, "The `yield` statement") is called a *generator function*. Such a function, when called, always returns an iterator object which can be used to execute the body of the function: calling the iterator's `next()` method will cause the function to execute until it provides a value using the `yield` statement. When the function executes a `return` statement or falls off the end, a `StopIteration` exception is raised and the iterator will have reached the end of the set of values to be returned.

Built-in functions A built-in function object is a wrapper around a C function. Examples of built-in functions are `len()` and `math.sin()` (`math` is a standard built-in module). The number and type of the arguments are determined by the C function. Special read-only attributes: `__doc__` is the function's documentation string, or `None` if unavailable; `__name__` is the function's name; `__self__` is set to `None` (but see the next item); `__module__` is the name of the module the function was defined in or `None` if unavailable.

Built-in methods This is really a different disguise of a built-in function, this time containing an object passed to the C function as an implicit extra argument. An example of a built-in method is *alist*`.append()`, assuming *alist* is a list object. In this case, the special read-only attribute `__self__` is set to the object denoted by *list*.

Class Types Class types, or "new-style classes," are callable. These objects normally act as factories for new instances of themselves, but variations are possible for class types that override `__new__()`. The arguments of the call are passed to `__new__()` and, in the typical case, to `__init__()` to initialize the new instance.

Classic Classes Class objects are described below. When a class object is called, a new class instance (also described below) is created and returned. This implies a call to the class's `__init__()` method if it has one. Any arguments are passed on to the `__init__()` method. If there is no `__init__()` method, the class must be called without arguments.

Class instances Class instances are described below. Class instances are callable only when the class has a `__call__()` method; `x(arguments)` is a shorthand for `x.__call__(arguments)`.

Modules Modules are imported by the `import` statement (see section 6.12, "The `import` statement"). A module object has a namespace implemented by a dictionary object (this is the dictionary referenced by the func_globals attribute of functions defined in the module). Attribute references are translated to lookups in this dictionary, e.g., `m.x` is equivalent to `m.__dict__["x"]`. A module object does not contain the code object used to initialize the module (since it isn't needed once the initialization is done).

Attribute assignment updates the module's namespace dictionary, e.g., '`m.x = 1`' is equivalent to '`m.__dict__["x"] = 1`'.

Special read-only attribute: `__dict__` is the module's namespace as a dictionary object.

Predefined (writable) attributes: `__name__` is the module's name; `__doc__` is the module's documentation string, or `None` if unavailable; `__file__` is the pathname of the file from which the module was loaded, if it was loaded from a file. The `__file__` attribute is not present for C modules that are statically linked into the interpreter; for extension modules loaded dynamically from a shared library, it is the pathname of the shared library file.

Classes Class objects are created by class definitions (see section 7.7, "Class definitions"). A class has a namespace implemented by a dictionary object. Class attribute references are translated to lookups in this dictionary, e.g., 'C.x' is translated to 'C.__dict__["x"]'. When the attribute name is not found there, the attribute search continues in the base classes. The search is depth-first, left-to-right in the order of occurrence in the base class list.

When a class attribute reference (for class C, say) would yield a user-defined function object or an unbound user-defined method object whose associated class is either C or one of its base classes, it is transformed into an unbound user-defined method object whose im_class attribute is C. When it would yield a class method object, it is transformed into a bound user-defined method object whose im_class and im_self attributes are both C. When it would yield a static method object, it is transformed into the object wrapped by the static method object. See section 3.4.2 for another way in which attributes retrieved from a class may differ from those actually contained in its __dict__.

Class attribute assignments update the class's dictionary, never the dictionary of a base class.

A class object can be called (see above) to yield a class instance (see below).

Special attributes: __name__ is the class name; __module__ is the module name in which the class was defined; __dict__ is the dictionary containing the class's namespace; __bases__ is a tuple (possibly empty or a singleton) containing the base classes, in the order of their occurrence in the base class list; __doc__ is the class's documentation string, or None if undefined.

Class instances A class instance is created by calling a class object (see above). A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes. If a class attribute is found that is a user-defined function object or an unbound user-defined method object whose associated class is the class (call it C) of the instance for which the attribute reference was initiated or one of its bases, it is transformed into a bound user-defined method object whose im_class attribute is C and whose im_self attribute is the instance. Static method and class method objects are also transformed, as if they had been retrieved from class C; see above under "Classes". See section 3.4.2 for another way in which attributes of a class retrieved via its instances may differ from the objects actually stored in the class's __dict__. If no class attribute is found, and the object's class has a __getattr__() method, that is called to satisfy the lookup.

Attribute assignments and deletions update the instance's dictionary, never a class's dictionary. If the class has a __setattr__() or __delattr__() method, this is called instead of updating the instance dictionary directly.

Class instances can pretend to be numbers, sequences, or mappings if they have methods with certain special names. See section 3.4, "Special method names."

Special attributes: __dict__ is the attribute dictionary; __class__ is the instance's class.

Files A file object represents an open file. File objects are created by the open() built-in function, and also by os.popen(), os.fdopen(), and the makefile() method of socket objects (and perhaps by other functions or methods provided by extension modules). The objects sys.stdin, sys.stdout and sys.stderr are initialized to file objects corresponding to the interpreter's standard input, output and error streams. See the *Python Library Reference* for complete documentation of file objects.

Internal types A few types used internally by the interpreter are exposed to the user. Their definitions may change with future versions of the interpreter, but they are mentioned here for completeness.

Code objects Code objects represent *byte-compiled* executable Python code, or *bytecode*. The difference between a code object and a function object is that the function object contains an explicit reference to the function's globals (the module in which it was defined), while a code object contains no context; also the default argument values are stored in the function object, not in the code object (because they represent values calculated at run-time). Unlike function objects, code objects are immutable and contain no references (directly or indirectly) to mutable objects.

Special read-only attributes: co_name gives the function name; co_argcount is the number of positional arguments (including arguments with default values); co_nlocals is the number of local variables used by the function (including arguments); co_varnames is a tuple containing the names of the local variables (starting with the argument names); co_cellvars is a tuple containing the names of local variables that are referenced by nested functions; co_freevars is a tuple containing the names of free variables; co_code is a string representing the sequence of bytecode instructions; co_consts is a tuple containing the literals used by the bytecode; co_names is a tuple containing the names used by the bytecode; co_filename is the filename from which the code was compiled; co_firstlineno is the first line number of the function; co_lnotab is a string encoding the mapping from byte code offsets to line numbers (for details see the source code of the interpreter); co_stacksize is the required stack size (including local variables); co_flags is an integer encoding a number of flags for the interpreter.

The following flag bits are defined for co_flags: bit 0x04 is set if the function uses the '*arguments' syntax to accept an arbitrary number of positional arguments; bit 0x08 is set if the function uses the '**keywords' syntax to accept arbitrary keyword arguments; bit 0x20 is set if the function is a generator.

Future feature declarations ('from __future__ import division') also use bits in co_flags to indicate whether a code object was compiled with a particular feature enabled: bit 0x2000 is set if the function was compiled with future division enabled; bits 0x10 and 0x1000 were used in earlier versions of Python.

Other bits in co_flags are reserved for internal use.

If a code object represents a function, the first item in co_consts is the documentation string of the function, or None if undefined.

Frame objects Frame objects represent execution frames. They may occur in traceback objects (see below).

Special read-only attributes: f_back is to the previous stack frame (towards the caller), or None if this is the bottom stack frame; f_code is the code object being executed in this frame; f_locals is the dictionary used to look up local variables; f_globals is used for global variables; f_builtins is used for built-in (intrinsic) names; f_restricted is a flag indicating whether the function is executing in restricted execution mode; f_lasti gives the precise instruction (this is an index into the bytecode string of the code object).

Special writable attributes: f_trace, if not None, is a function called at the start of each source code line (this is used by the debugger); f_exc_type, f_exc_value, f_exc_traceback represent the last exception raised in the parent frame provided another exception was ever raised in the current frame (in all other cases they are None); f_lineno is the current line number of the frame — writing to this from within a trace function jumps to the given line (only for the bottom-most frame). A debugger can implement a Jump command (aka Set Next Statement) by writing to f_lineno.

Traceback objects Traceback objects represent a stack trace of an exception. A traceback object is created when an exception occurs. When the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered, the stack trace is made available to the program. (See section 7.4, "The try statement.") It is accessible as sys.exc_traceback, and also as the third item of the tuple returned by sys.exc_info(). The latter is the preferred interface, since it works correctly when the program is using multiple threads. When the program contains no suitable handler, the stack trace is written (nicely formatted) to the standard error stream; if the interpreter is interactive, it is also made available to the user as sys.last_traceback.

Special read-only attributes: tb_next is the next level in the stack trace (towards the frame where the exception occurred), or None if there is no next level; tb_frame points to the execution frame of the current level; tb_lineno gives the line number where the exception occurred; tb_lasti indicates the precise instruction. The line number and last instruction in the traceback may differ from the line number of its frame object if the exception occurred in a try statement with no matching except clause or with a finally clause.

Slice objects Slice objects are used to represent slices when *extended slice syntax* is used. This is a slice using two colons, or multiple slices or ellipses separated by commas, e.g., a[i:j:step], a[i:j, k:l], or a[..., i:j]. They are also created by the built-in slice() function.

Special read-only attributes: `start` is the lower bound; `stop` is the upper bound; `step` is the step value; each is `None` if omitted. These attributes can have any type.

Slice objects support one method:

`indices`(*self, length*)

This method takes a single integer argument *length* and computes information about the extended slice that the slice object would describe if applied to a sequence of *length* items. It returns a tuple of three integers; respectively these are the *start* and *stop* indices and the *step* or stride length of the slice. Missing or out-of-bounds indices are handled in a manner consistent with regular slices. New in version 2.3.

Static method objects Static method objects provide a way of defeating the transformation of function objects to method objects described above. A static method object is a wrapper around any other object, usually a user-defined method object. When a static method object is retrieved from a class or a class instance, the object actually returned is the wrapped object, which is not subject to any further transformation. Static method objects are not themselves callable, although the objects they wrap usually are. Static method objects are created by the built-in `staticmethod()` constructor.

Class method objects A class method object, like a static method object, is a wrapper around another object that alters the way in which that object is retrieved from classes and class instances. The behaviour of class method objects upon such retrieval is described above, under "User-defined methods". Class method objects are created by the built-in `classmethod()` constructor.

3.3 New-style and classic classes

Classes and instances come in two flavors: old-style or classic, and new-style.

Up to Python 2.1, old-style classes were the only flavour available to the user. The concept of (old-style) class is unrelated to the concept of type: if *x* is an instance of an old-style class, then `x.__class__` designates the class of *x*, but `type(x)` is always `<type 'instance'>`. This reflects the fact that all old-style instances, independently of their class, are implemented with a single built-in type, called `instance`.

New-style classes were introduced in Python 2.2 to unify classes and types. A new-style class neither more nor less than a user-defined type. If *x* is an instance of a new-style class, then `type(x)` is the same as `x.__class__`.

The major motivation for introducing new-style classes is to provide a unified object model with a full meta-model. It also has a number of immediate benefits, like the ability to subclass most built-in types, or the introduction of "descriptors", which enable computed properties.

For compatibility reasons, classes are still old-style by default. New-style classes are created by specifying another new-style class (i.e. a type) as a parent class, or the "top-level type" `object` if no other parent is needed. The behaviour of new-style classes differs from that of old-style classes in a number of important details in addition to what `type` returns. Some of these changes are fundamental to the new object model, like the way special methods are invoked. Others are "fixes" that could not be implemented before for compatibility concerns, like the method resolution order in case of multiple inheritance.

This manual is not up-to-date with respect to new-style classes. For now, please see http://www.python.org/doc/newstyle/ for more information.

The plan is to eventually drop old-style classes, leaving only the semantics of new-style classes. This change will probably only be feasible in Python 3.0. new-style classic old-style

3.4 Special method names

A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python's approach to *operator overloading*, allowing classes to define their own behavior with respect to language operators. For instance, if a class defines a method named `__getitem__()`, and `x` is an instance of this class, then `x[i]` is equivalent[2] to `x.__getitem__(i)`. Except where mentioned, attempts to execute an operation raise an exception when no appropriate method is defined.

When implementing a class that emulates any built-in type, it is important that the emulation only be implemented to the degree that it makes sense for the object being modelled. For example, some sequences may work well with retrieval of individual elements, but extracting a slice may not make sense. (One example of this is the `NodeList` interface in the W3C's Document Object Model.)

3.4.1 Basic customization

`__new__`(*cls*[, ...])

Called to create a new instance of class *cls*. `__new__()` is a static method (special-cased so you need not declare it as such) that takes the class of which an instance was requested as its first argument. The remaining arguments are those passed to the object constructor expression (the call to the class). The return value of `__new__()` should be the new object instance (usually an instance of *cls*).

Typical implementations create a new instance of the class by invoking the superclass's `__new__()` method using '`super(`*currentclass*`,` *cls*`).__new__(`*cls*`[, ...])`' with appropriate arguments and then modifying the newly-created instance as necessary before returning it.

If `__new__()` returns an instance of *cls*, then the new instance's `__init__()` method will be invoked like '`__init__(`*self*`[, ...])`', where *self* is the new instance and the remaining arguments are the same as were passed to `__new__()`.

If `__new__()` does not return an instance of *cls*, then the new instance's `__init__()` method will not be invoked.

`__new__()` is intended mainly to allow subclasses of immutable types (like int, str, or tuple) to customize instance creation.

`__init__`(*self*[, ...])

Called when the instance is created. The arguments are those passed to the class constructor expression. If a base class has an `__init__()` method, the derived class's `__init__()` method, if any, must explicitly call it to ensure proper initialization of the base class part of the instance; for example: '`BaseClass.__init__(`*self*`, [`*args*`...])`'. As a special constraint on constructors, no value may be returned; doing so will cause a `TypeError` to be raised at runtime.

`__del__`(*self*)

Called when the instance is about to be destroyed. This is also called a destructor. If a base class has a `__del__()` method, the derived class's `__del__()` method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance. Note that it is possible (though not recommended!) for the `__del__()` method to postpone destruction of the instance by creating a new reference to it. It may then be called at a later time when this new reference is deleted. It is not guaranteed that `__del__()` methods are called for objects that still exist when the interpreter exits.

Note: '`del x`' doesn't directly call `x.__del__()` — the former decrements the reference count for `x` by one, and the latter is only called when `x`'s reference count reaches zero. Some common situations that may prevent the reference count of an object from going to zero include: circular references between objects (e.g., a doubly-linked list or a tree data structure with parent and child pointers); a reference to the object on the stack frame of a function that caught an exception (the traceback stored in `sys.exc_traceback` keeps the stack frame alive); or a reference to the object on the stack frame that raised an unhandled exception in interactive mode (the traceback stored in `sys.last_traceback` keeps the stack frame alive). The first situation can only be remedied by explicitly breaking the cycles; the latter two situations can be resolved by storing `None` in `sys.exc_traceback` or `sys.last_traceback`. Circular references which are garbage are detected when the option cycle detector is enabled (it's on by default), but can only be cleaned up if there are no Python-level `__del__()` methods involved. Refer to the documentation for the `gc` module for more information

[2] This, and other statements, are only roughly true for instances of new-style classes.

about how `__del__()` methods are handled by the cycle detector, particularly the description of the `garbage` value.

> **Warning:** Due to the precarious circumstances under which `__del__()` methods are invoked, exceptions that occur during their execution are ignored, and a warning is printed to `sys.stderr` instead. Also, when `__del__()` is invoked in response to a module being deleted (e.g., when execution of the program is done), other globals referenced by the `__del__()` method may already have been deleted. For this reason, `__del__()` methods should do the absolute minimum needed to maintain external invariants. Starting with version 1.5, Python guarantees that globals whose name begins with a single underscore are deleted from their module before other globals are deleted; if no other references to such globals exist, this may help in assuring that imported modules are still available at the time when the `__del__()` method is called.

`__repr__`(*self*)
Called by the `repr()` built-in function and by string conversions (reverse quotes) to compute the "official" string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value (given an appropriate environment). If this is not possible, a string of the form '<*...some useful description...*>' should be returned. The return value must be a string object. If a class defines `__repr__()` but not `__str__()`, then `__repr__()` is also used when an "informal" string representation of instances of that class is required.

This is typically used for debugging, so it is important that the representation is information-rich and unambiguous.

`__str__`(*self*)
Called by the `str()` built-in function and by the `print` statement to compute the "informal" string representation of an object. This differs from `__repr__()` in that it does not have to be a valid Python expression: a more convenient or concise representation may be used instead. The return value must be a string object.

`__lt__`(*self, other*)
`__le__`(*self, other*)
`__eq__`(*self, other*)
`__ne__`(*self, other*)
`__gt__`(*self, other*)
`__ge__`(*self, other*)
New in version 2.1. These are the so-called "rich comparison" methods, and are called for comparison operators in preference to `__cmp__()` below. The correspondence between operator symbols and method names is as follows: *x*<*y* calls *x*`.__lt__(`*y*`)`, *x*<=*y* calls *x*`.__le__(`*y*`)`, *x*==*y* calls *x*`.__eq__(`*y*`)`, *x*`!=`*y* and *x*<>*y* call *x*`.__ne__(`*y*`)`, *x*>*y* calls *x*`.__gt__(`*y*`)`, and *x*>=*y* calls *x*`.__ge__(`*y*`)`.

A rich comparison method may return the singleton `NotImplemented` if it does not implement the operation for a given pair of arguments. By convention, `False` and `True` are returned for a successful comparison. However, these methods can return any value, so if the comparison operator is used in a Boolean context (e.g., in the condition of an `if` statement), Python will call `bool()` on the value to determine if the result is true or false.

There are no implied relationships among the comparison operators. The truth of *x*==*y* does not imply that *x*`!=`*y* is false. Accordingly, when defining `__eq__()`, one should also define `__ne__()` so that the operators will behave as expected.

There are no reflected (swapped-argument) versions of these methods (to be used when the left argument does not support the operation but the right argument does); rather, `__lt__()` and `__gt__()` are each other's reflection, `__le__()` and `__ge__()` are each other's reflection, and `__eq__()` and `__ne__()` are their own reflection.

Arguments to rich comparison methods are never coerced.

`__cmp__`(*self, other*)
Called by comparison operations if rich comparison (see above) is not defined. Should return a negative integer

if `self < other`, zero if `self == other`, a positive integer if `self > other`. If no `__cmp__()`, `__eq__()` or `__ne__()` operation is defined, class instances are compared by object identity ("address"). See also the description of `__hash__()` for some important notes on creating objects which support custom comparison operations and are usable as dictionary keys. (Note: the restriction that exceptions are not propagated by `__cmp__()` has been removed since Python 1.5.)

`__rcmp__`(*self, other*)
Changed in version 2.1: No longer supported.

`__hash__`(*self*)
Called for the key object for dictionary operations, and by the built-in function `hash()`. Should return a 32-bit integer usable as a hash value for dictionary operations. The only required property is that objects which compare equal have the same hash value; it is advised to somehow mix together (e.g., using exclusive or) the hash values for the components of the object that also play a part in comparison of objects. If a class does not define a `__cmp__()` method it should not define a `__hash__()` operation either; if it defines `__cmp__()` or `__eq__()` but not `__hash__()`, its instances will not be usable as dictionary keys. If a class defines mutable objects and implements a `__cmp__()` or `__eq__()` method, it should not implement `__hash__()`, since the dictionary implementation requires that a key's hash value is immutable (if the object's hash value changes, it will be in the wrong hash bucket).

Changed in version 2.5: `__hash__()` may now also return a long integer object; the 32-bit integer is then derived from the hash of that object.

`__nonzero__`(*self*)
Called to implement truth value testing, and the built-in operation `bool()`; should return `False` or `True`, or their integer equivalents `0` or `1`. When this method is not defined, `__len__()` is called, if it is defined (see below). If a class defines neither `__len__()` nor `__nonzero__()`, all its instances are considered true.

`__unicode__`(*self*)
Called to implement `unicode()` builtin; should return a Unicode object. When this method is not defined, string conversion is attempted, and the result of string conversion is converted to Unicode using the system default encoding.

3.4.2 Customizing attribute access

The following methods can be defined to customize the meaning of attribute access (use of, assignment to, or deletion of `x.name`) for class instances.

`__getattr__`(*self, name*)
Called when an attribute lookup has not found the attribute in the usual places (i.e. it is not an instance attribute nor is it found in the class tree for `self`). `name` is the attribute name. This method should return the (computed) attribute value or raise an `AttributeError` exception.

Note that if the attribute is found through the normal mechanism, `__getattr__()` is not called. (This is an intentional asymmetry between `__getattr__()` and `__setattr__()`.) This is done both for efficiency reasons and because otherwise `__setattr__()` would have no way to access other attributes of the instance. Note that at least for instance variables, you can fake total control by not inserting any values in the instance attribute dictionary (but instead inserting them in another object). See the `__getattribute__()` method below for a way to actually get total control in new-style classes.

`__setattr__`(*self, name, value*)
Called when an attribute assignment is attempted. This is called instead of the normal mechanism (i.e. store the value in the instance dictionary). *name* is the attribute name, *value* is the value to be assigned to it.

If `__setattr__()` wants to assign to an instance attribute, it should not simply execute '`self.`*name* `= value`' — this would cause a recursive call to itself. Instead, it should insert the value in the dictionary of instance attributes, e.g., '`self.__dict__[`*name*`] = value`'. For new-style classes, rather than accessing the instance dictionary, it should call the base class method with the same name, for example, '`object.__-`

setattr__(self, name, value)'.

__delattr__(*self, name*)
Like __setattr__() but for attribute deletion instead of assignment. This should only be implemented if 'del obj.*name*' is meaningful for the object.

More attribute access for new-style classes

The following methods only apply to new-style classes.

__getattribute__(*self, name*)
Called unconditionally to implement attribute accesses for instances of the class. If the class also defines __getattr__(), the latter will not be called unless __getattribute__() either calls it explicitly or raises an AttributeError. This method should return the (computed) attribute value or raise an AttributeError exception. In order to avoid infinite recursion in this method, its implementation should always call the base class method with the same name to access any attributes it needs, for example, 'object.__getattribute__(self, name)'.

Implementing Descriptors

The following methods only apply when an instance of the class containing the method (a so-called *descriptor* class) appears in the class dictionary of another new-style class, known as the *owner* class. In the examples below, "the attribute" refers to the attribute whose name is the key of the property in the owner class' __dict__. Descriptors can only be implemented as new-style classes themselves.

__get__(*self, instance, owner*)
Called to get the attribute of the owner class (class attribute access) or of an instance of that class (instance attribute access). *owner* is always the owner class, while *instance* is the instance that the attribute was accessed through, or None when the attribute is accessed through the *owner*. This method should return the (computed) attribute value or raise an AttributeError exception.

__set__(*self, instance, value*)
Called to set the attribute on an instance *instance* of the owner class to a new value, *value*.

__delete__(*self, instance*)
Called to delete the attribute on an instance *instance* of the owner class.

Invoking Descriptors

In general, a descriptor is an object attribute with "binding behavior", one whose attribute access has been overridden by methods in the descriptor protocol: __get__(), __set__(), and __delete__(). If any of those methods are defined for an object, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, a.x has a lookup chain starting with a.__dict__['x'], then type(a).__dict__['x'], and continuing through the base classes of type(a) excluding metaclasses.

However, if the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined and how they were called. Note that descriptors are only invoked for new style objects or classes (ones that subclass object() or type()).

The starting point for descriptor invocation is a binding, a.x. How the arguments are assembled depends on a:

Direct Call The simplest and least common call is when user code directly invokes a descriptor method: x.__get__(a).

Instance Binding If binding to a new-style object instance, `a.x` is transformed into the call: `type(a).__dict__['x'].__get__(a, type(a))`.

Class Binding If binding to a new-style class, `A.x` is transformed into the call: `A.__dict__['x'].__get__(None, A)`.

Super Binding If `a` is an instance of `super`, then the binding `super(B, obj).m()` searches `obj.__class__.__mro__` for the base class `A` immediately preceding `B` and then invokes the descriptor with the call: `A.__dict__['m'].__get__(obj, A)`.

For instance bindings, the precedence of descriptor invocation depends on the which descriptor methods are defined. Data descriptors define both `__get__()` and `__set__()`. Non-data descriptors have just the `__get__()` method. Data descriptors always override a redefinition in an instance dictionary. In contrast, non-data descriptors can be overridden by instances.

Python methods (including `staticmethod()` and `classmethod()`) are implemented as non-data descriptors. Accordingly, instances can redefine and override methods. This allows individual instances to acquire behaviors that differ from other instances of the same class.

The `property()` function is implemented as a data descriptor. Accordingly, instances cannot override the behavior of a property.

__slots__

By default, instances of both old and new-style classes have a dictionary for attribute storage. This wastes space for objects having very few instance variables. The space consumption can become acute when creating large numbers of instances.

The default can be overridden by defining *__slots__* in a new-style class definition. The *__slots__* declaration takes a sequence of instance variables and reserves just enough space in each instance to hold a value for each variable. Space is saved because *__dict__* is not created for each instance.

`__slots__`

This class variable can be assigned a string, iterable, or sequence of strings with variable names used by instances. If defined in a new-style class, *__slots__* reserves space for the declared variables and prevents the automatic creation of *__dict__* and *__weakref__* for each instance. New in version 2.2.

Notes on using *__slots__*

- Without a *__dict__* variable, instances cannot be assigned new variables not listed in the *__slots__* definition. Attempts to assign to an unlisted variable name raises `AttributeError`. If dynamic assignment of new variables is desired, then add `'__dict__'` to the sequence of strings in the *__slots__* declaration. Changed in version 2.3: Previously, adding `'__dict__'` to the *__slots__* declaration would not enable the assignment of new attributes not specifically listed in the sequence of instance variable names.
- Without a *__weakref__* variable for each instance, classes defining *__slots__* do not support weak references to its instances. If weak reference support is needed, then add `'__weakref__'` to the sequence of strings in the *__slots__* declaration. Changed in version 2.3: Previously, adding `'__weakref__'` to the *__slots__* declaration would not enable support for weak references.
- *__slots__* are implemented at the class level by creating descriptors (3.4.2) for each variable name. As a result, class attributes cannot be used to set default values for instance variables defined by *__slots__*; otherwise, the class attribute would overwrite the descriptor assignment.
- If a class defines a slot also defined in a base class, the instance variable defined by the base class slot is inaccessible (except by retrieving its descriptor directly from the base class). This renders the meaning of the program undefined. In the future, a check may be added to prevent this.

- The action of a *__slots__* declaration is limited to the class where it is defined. As a result, subclasses will have a *__dict__* unless they also define *__slots__*.
- *__slots__* do not work for classes derived from "variable-length" built-in types such as `long`, `str` and `tuple`.
- Any non-string iterable may be assigned to *__slots__*. Mappings may also be used; however, in the future, special meaning may be assigned to the values corresponding to each key.

3.4.3 Customizing class creation

By default, new-style classes are constructed using `type()`. A class definition is read into a separate namespace and the value of class name is bound to the result of `type(name, bases, dict)`.

When the class definition is read, if *__metaclass__* is defined then the callable assigned to it will be called instead of `type()`. The allows classes or functions to be written which monitor or alter the class creation process:

- Modifying the class dictionary prior to the class being created.
- Returning an instance of another class – essentially performing the role of a factory function.

`__metaclass__`
: This variable can be any callable accepting arguments for `name`, `bases`, and `dict`. Upon class creation, the callable is used instead of the built-in `type()`. New in version 2.2.

The appropriate metaclass is determined by the following precedence rules:

- If `dict['__metaclass__']` exists, it is used.
- Otherwise, if there is at least one base class, its metaclass is used (this looks for a *__class__* attribute first and if not found, uses its type).
- Otherwise, if a global variable named __metaclass__ exists, it is used.
- Otherwise, the old-style, classic metaclass (types.ClassType) is used.

The potential uses for metaclasses are boundless. Some ideas that have been explored including logging, interface checking, automatic delegation, automatic property creation, proxies, frameworks, and automatic resource locking/synchronization.

3.4.4 Emulating callable objects

`__call__`(*self*[, *args...*])
: Called when the instance is "called" as a function; if this method is defined, *x*`(arg1, arg2, ...)` is a shorthand for *x*`.__call__(arg1, arg2, ...)`.

3.4.5 Emulating container types

The following methods can be defined to implement container objects. Containers usually are sequences (such as lists or tuples) or mappings (like dictionaries), but can represent other containers as well. The first set of methods is used either to emulate a sequence or to emulate a mapping; the difference is that for a sequence, the allowable keys should be the integers *k* for which `0 <= `*k*` < `*N* where *N* is the length of the sequence, or slice objects, which define a range of items. (For backwards compatibility, the method `__getslice__()` (see below) can also be defined to handle simple, but not extended slices.) It is also recommended that mappings provide the methods `keys()`, `values()`, `items()`, `has_key()`, `get()`, `clear()`, `setdefault()`, `iterkeys()`, `itervalues()`, iteritems(), pop(), popitem(), copy(), and update() behaving similar to those for Python's standard dictionary objects. The UserDict module provides a DictMixin class to help create those methods from a base set of __getitem__(), __setitem__(), __delitem__(), and keys(). Mutable sequences should provide methods append(), count(), index(), extend(), insert(), pop(), remove(), reverse() and sort(), like Python standard list objects. Finally, sequence types should implement addition (meaning concatenation) and multiplication (meaning repetition) by defining the methods __add__(), __radd__(), __iadd__(), __mul__(), __rmul__() and __imul__() described below; they should not define __coerce__() or other numerical operators. It is recommended that both mappings and sequences implement the __contains__() method to allow efficient use of the in operator; for mappings, in should be equivalent of has_key(); for sequences, it should search through the values. It is further recommended that both mappings and sequences implement the __iter__() method to allow efficient iteration through the container; for mappings, __iter__() should be the same as iterkeys(); for sequences, it should iterate through the values.

__len__(*self*)
: Called to implement the built-in function len(). Should return the length of the object, an integer >= 0. Also, an object that doesn't define a __nonzero__() method and whose __len__() method returns zero is considered to be false in a Boolean context.

__getitem__(*self, key*)
: Called to implement evaluation of *self*[*key*]. For sequence types, the accepted keys should be integers and slice objects. Note that the special interpretation of negative indexes (if the class wishes to emulate a sequence type) is up to the __getitem__() method. If *key* is of an inappropriate type, TypeError may be raised; if of a value outside the set of indexes for the sequence (after any special interpretation of negative values), IndexError should be raised. For mapping types, if *key* is missing (not in the container), KeyError should be raised. **Note:** for loops expect that an IndexError will be raised for illegal indexes to allow proper detection of the end of the sequence.

__setitem__(*self, key, value*)
: Called to implement assignment to *self*[*key*]. Same note as for __getitem__(). This should only be implemented for mappings if the objects support changes to the values for keys, or if new keys can be added, or for sequences if elements can be replaced. The same exceptions should be raised for improper *key* values as for the __getitem__() method.

__delitem__(*self, key*)
: Called to implement deletion of *self*[*key*]. Same note as for __getitem__(). This should only be implemented for mappings if the objects support removal of keys, or for sequences if elements can be removed from the sequence. The same exceptions should be raised for improper *key* values as for the __getitem__() method.

__iter__(*self*)
: This method is called when an iterator is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container, and should also be made available as the method iterkeys().

  Iterator objects also need to implement this method; they are required to return themselves. For more information on iterator objects, see "Iterator Types" in the *Python Library Reference*.

The membership test operators (in and not in) are normally implemented as an iteration through a sequence. However, container objects can supply the following special method with a more efficient implementation, which also does not require the object be a sequence.

__contains__(*self, item*)
: Called to implement membership test operators. Should return true if *item* is in *self*, false otherwise. For mapping objects, this should consider the keys of the mapping rather than the values or the key-item pairs.

3.4.6 Additional methods for emulation of sequence types

The following optional methods can be defined to further emulate sequence objects. Immutable sequences methods should at most only define `__getslice__()`; mutable sequences might define all three methods.

`__getslice__`(*self, i, j*)
: **Deprecated since release 2.0.** Support slice objects as parameters to the `__getitem__()` method.

  Called to implement evaluation of *self*`[`*i*`:`*j*`]`. The returned object should be of the same type as *self*. Note that missing *i* or *j* in the slice expression are replaced by zero or `sys.maxint`, respectively. If negative indexes are used in the slice, the length of the sequence is added to that index. If the instance does not implement the `__len__()` method, an `AttributeError` is raised. No guarantee is made that indexes adjusted this way are not still negative. Indexes which are greater than the length of the sequence are not modified. If no `__getslice__()` is found, a slice object is created instead, and passed to `__getitem__()` instead.

`__setslice__`(*self, i, j, sequence*)
: Called to implement assignment to *self*`[`*i*`:`*j*`]`. Same notes for *i* and *j* as for `__getslice__()`.

  This method is deprecated. If no `__setslice__()` is found, or for extended slicing of the form *self*`[`*i*`:`*j*`:`*k*`]`, a slice object is created, and passed to `__setitem__()`, instead of `__setslice__()` being called.

`__delslice__`(*self, i, j*)
: Called to implement deletion of *self*`[`*i*`:`*j*`]`. Same notes for *i* and *j* as for `__getslice__()`. This method is deprecated. If no `__delslice__()` is found, or for extended slicing of the form *self*`[`*i*`:`*j*`:`*k*`]`, a slice object is created, and passed to `__delitem__()`, instead of `__delslice__()` being called.

Notice that these methods are only invoked when a single slice with a single colon is used, and the slice method is available. For slice operations involving extended slice notation, or in absence of the slice methods, `__getitem__()`, `__setitem__()` or `__delitem__()` is called with a slice object as argument.

The following example demonstrate how to make your program or module compatible with earlier versions of Python (assuming that methods `__getitem__()`, `__setitem__()` and `__delitem__()` support slice objects as arguments):

```
class MyClass:
    ...
    def __getitem__(self, index):
        ...
    def __setitem__(self, index, value):
        ...
    def __delitem__(self, index):
        ...

    if sys.version_info < (2, 0):
        # They won't be defined if version is at least 2.0 final

        def __getslice__(self, i, j):
            return self[max(0, i):max(0, j):]
        def __setslice__(self, i, j, seq):
            self[max(0, i):max(0, j):] = seq
        def __delslice__(self, i, j):
            del self[max(0, i):max(0, j):]
    ...
```

Note the calls to `max()`; these are necessary because of the handling of negative indices before the `__*slice__()` methods are called. When negative indexes are used, the `__*item__()` methods receive them as provided, but the `__*slice__()` methods get a "cooked" form of the index values. For each negative index value, the length of the sequence is added to the index before calling the method (which may still result in a negative index); this is the customary handling of negative indexes by the built-in sequence types, and the `__*item__()` methods are expected to do this as well. However, since they should already be doing that, negative indexes cannot be passed in; they must be constrained to the bounds of the sequence before being passed to the `__*item__()` methods. Calling `max(0, i)` conveniently returns the proper value.

3.4.7 Emulating numeric types

The following methods can be defined to emulate numeric objects. Methods corresponding to operations that are not supported by the particular kind of number implemented (e.g., bitwise operations for non-integral numbers) should be left undefined.

`__add__`(*self, other*)
`__sub__`(*self, other*)
`__mul__`(*self, other*)
`__floordiv__`(*self, other*)
`__mod__`(*self, other*)
`__divmod__`(*self, other*)
`__pow__`(*self, other*[*, modulo*])
`__lshift__`(*self, other*)
`__rshift__`(*self, other*)
`__and__`(*self, other*)
`__xor__`(*self, other*)
`__or__`(*self, other*)

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`). For instance, to evaluate the expression *x*+*y*, where *x* is an instance of a class that has an `__add__()` method, `x.__add__(y)` is called. The `__divmod__()` method should be the equivalent to using `__floordiv__()` and `__mod__()`; it should not be related to `__truediv__()` (described below). Note that `__pow__()` should be defined to accept an optional third argument if the ternary version of the built-in `pow()` function is to be supported.

If one of those methods does not support the operation with the supplied arguments, it should return `NotImplemented`.

`__div__`(*self, other*)
`__truediv__`(*self, other*)

The division operator (`/`) is implemented by these methods. The `__truediv__()` method is used when `__future__.division` is in effect, otherwise `__div__()` is used. If only one of these two methods is defined, the object will not support division in the alternate context; `TypeError` will be raised instead.

`__radd__`(*self, other*)
`__rsub__`(*self, other*)
`__rmul__`(*self, other*)
`__rdiv__`(*self, other*)
`__rtruediv__`(*self, other*)
`__rfloordiv__`(*self, other*)
`__rmod__`(*self, other*)
`__rdivmod__`(*self, other*)
`__rpow__`(*self, other*)
`__rlshift__`(*self, other*)
`__rrshift__`(*self, other*)
`__rand__`(*self, other*)
`__rxor__`(*self, other*)
`__ror__`(*self, other*)

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `/`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation and the operands are of different types.[3] For instance, to evaluate the expression $x-y$, where y is an instance of a class that has an `__rsub__()` method, `y.__rsub__(x)` is called if `x.__sub__(y)` returns *NotImplemented*.

Note that ternary `pow()` will not try calling `__rpow__()` (the coercion rules would become too complicated).

Note: If the right operand's type is a subclass of the left operand's type and that subclass provides the reflected method for the operation, this method will be called before the left operand's non-reflected method. This behavior allows subclasses to override their ancestors' operations.

`__iadd__`(*self, other*)
`__isub__`(*self, other*)
`__imul__`(*self, other*)
`__idiv__`(*self, other*)
`__itruediv__`(*self, other*)
`__ifloordiv__`(*self, other*)
`__imod__`(*self, other*)
`__ipow__`(*self, other*[, *modulo*])
`__ilshift__`(*self, other*)
`__irshift__`(*self, other*)
`__iand__`(*self, other*)
`__ixor__`(*self, other*)
`__ior__`(*self, other*)

These methods are called to implement the augmented arithmetic operations (`+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=`, `|=`). These methods should attempt to do the operation in-place (modifying *self*) and return the result (which could be, but does not have to be, *self*). If a specific method is not defined, the augmented operation falls back to the normal methods. For instance, to evaluate the expression x+=y, where x is an instance of a class that has an `__iadd__()` method, `x.__iadd__(y)` is called. If x is an instance of a class that does not define a `__iadd__()` method, `x.__add__(y)` and `y.__radd__(x)` are considered, as with the evaluation of $x+y$.

`__neg__`(*self*)
`__pos__`(*self*)
`__abs__`(*self*)
`__invert__`(*self*)

Called to implement the unary arithmetic operations (`-`, `+`, `abs()` and `~`).

`__complex__`(*self*)
`__int__`(*self*)
`__long__`(*self*)
`__float__`(*self*)

Called to implement the built-in functions `complex()`, `int()`, `long()`, and `float()`. Should return a value of the appropriate type.

`__oct__`(*self*)
`__hex__`(*self*)

Called to implement the built-in functions `oct()` and `hex()`. Should return a string value.

`__index__`(*self*)

Called to implement `operator.index()`. Also called whenever Python needs an integer object (such as in slicing). Must return an integer (int or long). New in version 2.5.

`__coerce__`(*self, other*)

Called to implement "mixed-mode" numeric arithmetic. Should either return a 2-tuple containing *self* and *other* converted to a common numeric type, or `None` if conversion is impossible. When the common type would be the type of `other`, it is sufficient to return `None`, since the interpreter will also ask the other object to attempt

[3]For operands of the same type, it is assumed that if the non-reflected method (such as `__add__()`) fails the operation is not supported, which is why the reflected method is not called.

a coercion (but sometimes, if the implementation of the other type cannot be changed, it is useful to do the conversion to the other type here). A return value of `NotImplemented` is equivalent to returning `None`.

3.4.8 Coercion rules

This section used to document the rules for coercion. As the language has evolved, the coercion rules have become hard to document precisely; documenting what one version of one particular implementation does is undesirable. Instead, here are some informal guidelines regarding coercion. In Python 3.0, coercion will not be supported.

- If the left operand of a % operator is a string or Unicode object, no coercion takes place and the string formatting operation is invoked instead.
- It is no longer recommended to define a coercion operation. Mixed-mode operations on types that don't define coercion pass the original arguments to the operation.
- New-style classes (those derived from `object`) never invoke the `__coerce__()` method in response to a binary operator; the only time `__coerce__()` is invoked is when the built-in function `coerce()` is called.
- For most intents and purposes, an operator that returns `NotImplemented` is treated the same as one that is not implemented at all.
- Below, `__op__()` and `__rop__()` are used to signify the generic method names corresponding to an operator; `__iop__()` is used for the corresponding in-place operator. For example, for the operator '`+`', `__add__()` and `__radd__()` are used for the left and right variant of the binary operator, and `__iadd__()` for the in-place variant.
- For objects *x* and *y*, first *x*`.__op__(`*y*`)` is tried. If this is not implemented or returns `NotImplemented`, *y*`.__rop__(`*x*`)` is tried. If this is also not implemented or returns `NotImplemented`, a `TypeError` exception is raised. But see the following exception:
- Exception to the previous item: if the left operand is an instance of a built-in type or a new-style class, and the right operand is an instance of a proper subclass of that type or class and overrides the base's `__rop__()` method, the right operand's `__rop__()` method is tried *before* the left operand's `__op__()` method.

  This is done so that a subclass can completely override binary operators. Otherwise, the left operand's `__op__()` method would always accept the right operand: when an instance of a given class is expected, an instance of a subclass of that class is always acceptable.
- When either operand type defines a coercion, this coercion is called before that type's `__op__()` or `__rop__()` method is called, but no sooner. If the coercion returns an object of a different type for the operand whose coercion is invoked, part of the process is redone using the new object.
- When an in-place operator (like '`+=`') is used, if the left operand implements `__iop__()`, it is invoked without any coercion. When the operation falls back to `__op__()` and/or `__rop__()`, the normal coercion rules apply.
- In *x*+*y*, if *x* is a sequence that implements sequence concatenation, sequence concatenation is invoked.
- In *x*`*`*y*, if one operator is a sequence that implements sequence repetition, and the other is an integer (`int` or `long`), sequence repetition is invoked.
- Rich comparisons (implemented by methods `__eq__()` and so on) never use coercion. Three-way comparison (implemented by `__cmp__()`) does use coercion under the same conditions as other binary operations use it.
- In the current implementation, the built-in numeric types `int`, `long` and `float` do not use coercion; the type `complex` however does use it. The difference can become apparent when subclassing these types. Over time, the type `complex` may be fixed to avoid coercion. All these types implement a `__coerce__()` method, for use by the built-in `coerce()` function.

3.4.9 With Statement Context Managers

New in version 2.5.

A *context manager* is an object that defines the runtime context to be established when executing a `with` statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the `with` statement (described in section 7.5), but can also be used by directly invoking their methods.

Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc.

For more information on context managers, see "Context Types" in the *Python Library Reference*.

`__enter__`(*self*)

Enter the runtime context related to this object. The `with` statement will bind this method's return value to the target(s) specified in the `as` clause of the statement, if any.

`__exit__`(*self, exc_type, exc_value, traceback*)

Exit the runtime context related to this object. The parameters describe the exception that caused the context to be exited. If the context was exited without an exception, all three arguments will be `None`.

If an exception is supplied, and the method wishes to suppress the exception (i.e., prevent it from being propagated), it should return a true value. Otherwise, the exception will be processed normally upon exit from this method.

Note that `__exit__` methods should not reraise the passed-in exception; this is the caller's responsibility.

See Also:

PEP 0343, "*The "with" statement*"

The specification, background, and examples for the Python `with` statement.

CHAPTER

FOUR

Execution model

4.1 Naming and binding

Names refer to objects. Names are introduced by name binding operations. Each occurrence of a name in the program text refers to the *binding* of that name established in the innermost function block containing the use.

A *block* is a piece of Python program text that is executed as a unit. The following are blocks: a module, a function body, and a class definition. Each command typed interactively is a block. A script file (a file given as standard input to the interpreter or specified on the interpreter command line the first argument) is a code block. A script command (a command specified on the interpreter command line with the '**-c**' option) is a code block. The file read by the built-in function `execfile()` is a code block. The string argument passed to the built-in function `eval()` and to the `exec` statement is a code block. The expression read and evaluated by the built-in function `input()` is a code block.

A code block is executed in an *execution frame*. A frame contains some administrative information (used for debugging) and determines where and how execution continues after the code block's execution has completed.

A *scope* defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. If the definition occurs in a function block, the scope extends to any blocks contained within the defining one, unless a contained block introduces a different binding for the name. The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods.

When a name is used in a code block, it is resolved using the nearest enclosing scope. The set of all such scopes visible to a code block is called the block's *environment*.

If a name is bound in a block, it is a local variable of that block. If a name is bound at the module level, it is a global variable. (The variables of the module code block are local and global.) If a variable is used in a code block but not defined there, it is a *free variable*.

When a name is not found at all, a `NameError` exception is raised. If the name refers to a local variable that has not been bound, a `UnboundLocalError` exception is raised. `UnboundLocalError` is a subclass of `NameError`.

The following constructs bind names: formal parameters to functions, `import` statements, class and function definitions (these bind the class or function name in the defining block), and targets that are identifiers if occurring in an assignment, `for` loop header, or in the second position of an `except` clause header. The `import` statement of the form "'`from ...import *`"' binds all names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

A target occurring in a `del` statement is also considered bound for this purpose (though the actual semantics are to unbind the name). It is illegal to unbind a name that is referenced by an enclosing scope; the compiler will report a `SyntaxError`.

Each assignment or import statement occurs within a block defined by a class or function definition or at the module level (the top-level code block).

If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block. This can lead to errors when a name is used within a block before it is bound. This rule is subtle. Python lacks declarations and allows name binding operations to occur anywhere within a code block. The local variables of a code block can be determined by scanning the entire text of the block for name binding operations.

If the global statement occurs within a block, all uses of the name specified in the statement refer to the binding of that name in the top-level namespace. Names are resolved in the top-level namespace by searching the global namespace, i.e. the namespace of the module containing the code block, and the builtin namespace, the namespace of the module `__builtin__`. The global namespace is searched first. If the name is not found there, the builtin namespace is searched. The global statement must precede all uses of the name.

The built-in namespace associated with the execution of a code block is actually found by looking up the name `__builtins__` in its global namespace; this should be a dictionary or a module (in the latter case the module's dictionary is used). By default, when in the `__main__` module, `__builtins__` is the built-in module `__builtin__` (note: no 's'); when in any other module, `__builtins__` is an alias for the dictionary of the `__builtin__` module itself. `__builtins__` can be set to a user-created dictionary to create a weak form of restricted execution.

Note: Users should not touch `__builtins__`; it is strictly an implementation detail. Users wanting to override values in the built-in namespace should `import` the `__builtin__` (no 's') module and modify its attributes appropriately.

The namespace for a module is automatically created the first time a module is imported. The main module for a script is always called `__main__`.

The global statement has the same scope as a name binding operation in the same block. If the nearest enclosing scope for a free variable contains a global statement, the free variable is treated as a global.

A class definition is an executable statement that may use and define names. These references follow the normal rules for name resolution. The namespace of the class definition becomes the attribute dictionary of the class. Names defined at the class scope are not visible in methods.

4.1.1 Interaction with dynamic features

There are several cases where Python statements are illegal when used in conjunction with nested scopes that contain free variables.

If a variable is referenced in an enclosing scope, it is illegal to delete the name. An error will be reported at compile time.

If the wild card form of import — '`import *`' — is used in a function and the function contains or is a nested block with free variables, the compiler will raise a `SyntaxError`.

If `exec` is used in a function and the function contains or is a nested block with free variables, the compiler will raise a `SyntaxError` unless the exec explicitly specifies the local namespace for the `exec`. (In other words, '`exec obj`' would be illegal, but '`exec obj in ns`' would be legal.)

The `eval()`, `execfile()`, and `input()` functions and the `exec` statement do not have access to the full environment for resolving names. Names may be resolved in the local and global namespaces of the caller. Free variables are not resolved in the nearest enclosing namespace, but in the global namespace.[1] The `exec` statement and the `eval()` and `execfile()` functions have optional arguments to override the global and local namespace. If only one namespace is specified, it is used for both.

4.2 Exceptions

Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions. An exception is *raised* at the point where the error is detected; it may be *handled* by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error

[1] This limitation occurs because the code that is executed by these operations is not available at the time the module is compiled.

occurred.

The Python interpreter raises an exception when it detects a run-time error (such as division by zero). A Python program can also explicitly raise an exception with the `raise` statement. Exception handlers are specified with the `try` ... `except` statement. The `try` ... `finally` statement specifies cleanup code which does not handle the exception, but is executed whether an exception occurred or not in the preceding code.

Python uses the "termination" model of error handling: an exception handler can find out what happened and continue execution at an outer level, but it cannot repair the cause of the error and retry the failing operation (except by re-entering the offending piece of code from the top).

When an exception is not handled at all, the interpreter terminates execution of the program, or returns to its interactive main loop. In either case, it prints a stack backtrace, except when the exception is `SystemExit`.

Exceptions are identified by class instances. The `except` clause is selected depending on the class of the instance: it must reference the class of the instance or a base class thereof. The instance can be received by the handler and can carry additional information about the exceptional condition.

Exceptions can also be identified by strings, in which case the `except` clause is selected by object identity. An arbitrary value can be raised along with the identifying string which can be passed to the handler.

Deprecated since release 2.5. String exceptions should not be used in new code. They will not be supported in a future version of Python. Old code should be rewritten to use class exceptions instead.

> **Warning:** Messages to exceptions are not part of the Python API. Their contents may change from one version of Python to the next without warning and should not be relied on by code which will run under multiple versions of the interpreter.

See also the description of the `try` statement in section 7.4 and `raise` statement in section 6.9.

CHAPTER

FIVE

Expressions

This chapter explains the meaning of the elements of expressions in Python.

Syntax Notes: In this and the following chapters, extended BNF notation will be used to describe syntax, not lexical analysis. When (one alternative of) a syntax rule has the form

```
name  ::=  othername
```

and no semantics are given, the semantics of this form of `name` are the same as for `othername`.

5.1 Arithmetic conversions

When a description of an arithmetic operator below uses the phrase "the numeric arguments are converted to a common type," the arguments are coerced using the coercion rules listed at 3.4.8. If both arguments are standard numeric types, the following coercions are applied:

- If either argument is a complex number, the other is converted to complex;
- otherwise, if either argument is a floating point number, the other is converted to floating point;
- otherwise, if either argument is a long integer, the other is converted to long integer;
- otherwise, both must be plain integers and no conversion is necessary.

Some additional rules apply for certain operators (e.g., a string left argument to the '%' operator). Extensions can define their own coercions.

5.2 Atoms

Atoms are the most basic elements of expressions. The simplest atoms are identifiers or literals. Forms enclosed in reverse quotes or in parentheses, brackets or braces are also categorized syntactically as atoms. The syntax for atoms is:

```
atom       ::=  identifier | literal | enclosure
enclosure  ::=  parenth_form | list_display
                | generator_expression | dict_display
                | string_conversion | yield_atom
```

5.2.1 Identifiers (Names)

An identifier occurring as an atom is a name. See section 2.3 for lexical definition and section 4.1 for documentation of naming and binding.

When the name is bound to an object, evaluation of the atom yields that object. When a name is not bound, an attempt to evaluate it raises a `NameError` exception.

Private name mangling: When an identifier that textually occurs in a class definition begins with two or more underscore characters and does not end in two or more underscores, it is considered a *private name* of that class. Private names are transformed to a longer form before code is generated for them. The transformation inserts the class name in front of the name, with leading underscores removed, and a single underscore inserted in front of the class name. For example, the identifier `__spam` occurring in a class named `Ham` will be transformed to `_Ham__spam`. This transformation is independent of the syntactical context in which the identifier is used. If the transformed name is extremely long (longer than 255 characters), implementation defined truncation may happen. If the class name consists only of underscores, no transformation is done.

5.2.2 Literals

Python supports string literals and various numeric literals:

```
literal  ::=  stringliteral | integer | longinteger
              | floatnumber | imagnumber
```

Evaluation of a literal yields an object of the given type (string, integer, long integer, floating point number, complex number) with the given value. The value may be approximated in the case of floating point and imaginary (complex) literals. See section 2.4 for details.

All literals correspond to immutable data types, and hence the object's identity is less important than its value. Multiple evaluations of literals with the same value (either the same occurrence in the program text or a different occurrence) may obtain the same object or a different object with the same value.

5.2.3 Parenthesized forms

A parenthesized form is an optional expression list enclosed in parentheses:

```
parenth_form  ::=  "(" [expression_list] ")"
```

A parenthesized expression list yields whatever that expression list yields: if the list contains at least one comma, it yields a tuple; otherwise, it yields the single expression that makes up the expression list.

An empty pair of parentheses yields an empty tuple object. Since tuples are immutable, the rules for literals apply (i.e., two occurrences of the empty tuple may or may not yield the same object).

Note that tuples are not formed by the parentheses, but rather by use of the comma operator. The exception is the empty tuple, for which parentheses *are* required — allowing unparenthesized "nothing" in expressions would cause ambiguities and allow common typos to pass uncaught.

5.2.4 List displays

A list display is a possibly empty series of expressions enclosed in square brackets:

```
list_display         ::=  "[" [expression_list | list_comprehension] "]"
list_comprehension   ::=  expression list_for
list_for             ::=  "for" target_list "in" old_expression_list [list_iter]
old_expression_list  ::=  old_expression [("," old_expression)+ [","]]
list_iter            ::=  list_for | list_if
list_if              ::=  "if" old_expression [list_iter]
```

A list display yields a new list object. Its contents are specified by providing either a list of expressions or a list comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and placed into the list object in that order. When a list comprehension is supplied, it consists of a single expression followed by at least one `for` clause and zero or more `for` or `if` clauses. In this case, the elements of the new list are those that would be produced by considering each of the `for` or `if` clauses a block, nesting from left to right, and evaluating the expression to produce a list element each time the innermost block is reached[1].

5.2.5 Generator expressions

A generator expression is a compact generator notation in parentheses:

```
generator_expression  ::=  "(" expression genexpr_for ")"
genexpr_for           ::=  "for" target_list "in" or_test [genexpr_iter]
genexpr_iter          ::=  genexpr_for | genexpr_if
genexpr_if            ::=  "if" old_expression [genexpr_iter]
```

A generator expression yields a new generator object. It consists of a single expression followed by at least one `for` clause and zero or more `for` or `if` clauses. The iterating values of the new generator are those that would be produced by considering each of the `for` or `if` clauses a block, nesting from left to right, and evaluating the expression to yield a value that is reached the innermost block for each iteration.

Variables used in the generator expression are evaluated lazily when the `next()` method is called for generator object (in the same fashion as normal generators). However, the leftmost `for` clause is immediately evaluated so that error produced by it can be seen before any other possible error in the code that handles the generator expression. Subsequent `for` clauses cannot be evaluated immediately since they may depend on the previous `for` loop. For example: '`(x*y for x in range(10) for y in bar(x))`'.

The parentheses can be omitted on calls with only one argument. See section 5.3.4 for the detail.

5.2.6 Dictionary displays

A dictionary display is a possibly empty series of key/datum pairs enclosed in curly braces:

```
dict_display    ::=  "{" [key_datum_list] "}"
key_datum_list  ::=  key_datum ("," key_datum)* [","]
key_datum       ::=  expression ":" expression
```

A dictionary display yields a new dictionary object.

The key/datum pairs are evaluated from left to right to define the entries of the dictionary: each key object is used as a key into the dictionary to store the corresponding datum.

Restrictions on the types of the key values are listed earlier in section 3.2. (To summarize, the key type should be hashable, which excludes all mutable objects.) Clashes between duplicate keys are not detected; the last datum (textually rightmost in the display) stored for a given key value prevails.

[1] In Python 2.3, a list comprehension "leaks" the control variables of each '`for`' it contains into the containing scope. However, this behavior is deprecated, and relying on it will not work once this bug is fixed in a future release

5.2.7 String conversions

A string conversion is an expression list enclosed in reverse (a.k.a. backward) quotes:

```
string_conversion  ::=  "`" expression_list "`"
```

A string conversion evaluates the contained expression list and converts the resulting object into a string according to rules specific to its type.

If the object is a string, a number, `None`, or a tuple, list or dictionary containing only objects whose type is one of these, the resulting string is a valid Python expression which can be passed to the built-in function `eval()` to yield an expression with the same value (or an approximation, if floating point numbers are involved).

(In particular, converting a string adds quotes around it and converts "funny" characters to escape sequences that are safe to print.)

Recursive objects (for example, lists or dictionaries that contain a reference to themselves, directly or indirectly) use '`...`' to indicate a recursive reference, and the result cannot be passed to `eval()` to get an equal value (`SyntaxError` will be raised instead).

The built-in function `repr()` performs exactly the same conversion in its argument as enclosing it in parentheses and reverse quotes does. The built-in function `str()` performs a similar but more user-friendly conversion.

5.2.8 Yield expressions

```
yield_atom        ::=  "(" yield_expression ")"
yield_expression  ::=  "yield" [expression_list]
```

New in version 2.5.

The `yield` expression is only used when defining a generator function, and can only be used in the body of a function definition. Using a `yield` expression in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

When a generator function is called, it returns an iterator known as a generator. That generator then controls the execution of a generator function. The execution starts when one of the generator's methods is called. At that time, the execution proceeds to the first `yield` expression, where it is suspended again, returning the value of `expression_list` to generator's caller. By suspended we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, and the internal evaluation stack. When the execution is resumed by calling one of the generator's methods, the function can proceed exactly as if the `yield` expression was just another external call. The value of the `yield` expression after resuming depends on the method which resumed the execution.

All of this makes generator functions quite similar to coroutines; they yield multiple times, they have more than one entry point and their execution can be suspended. The only difference is that a generator function cannot control where should the execution continue after it yields; the control is always transfered to the generator's caller.

The following generator's methods can be used to control the execution of a generator function:

`next``()`

Starts the execution of a generator function or resumes it at the last executed `yield` expression. When a generator function is resumed with a `next()` method, the current `yield` expression always evaluates to `None`. The execution then continues to the next `yield` expression, where the generator is suspended again, and the value of the `expression_list` is returned to `next()`'s caller. If the generator exits without yielding another value, a `StopIteration` exception is raised.

`send``(`*value*`)`

Resumes the execution and "sends" a value into the generator function. The `value` argument becomes the result of the current `yield` expression. The `send()` method returns the next value yielded by the generator, or raises `StopIteration` if the generator exits without yielding another value. When `send()` is called to start the generator, it must be called with `None` as the argument, because there is no `yield` expression that could receieve the value.

throw(*type*[, *value*[, *traceback*]])
Raises an exception of type `type` at the point where generator was paused, and returns the next value yielded by the generator function. If the generator exits without yielding another value, a `StopIteration` exception is raised. If the generator function does not catch the passed-in exception, or raises a different exception, then that exception propagates to the caller.

close()
Raises a `GeneratorExit` at the point where the generator function was paused. If the generator function then raises `StopIteration` (by exiting normally, or due to already being closed) or `GeneratorExit` (by not catching the exception), close returns to its caller. If the generator yields a value, a `RuntimeError` is raised. If the generator raises any other exception, it is propagated to the caller. `close` does nothing if the generator has already exited due to an exception or normal exit.

Here is a simple example that demonstrates the behavior of generators and generator functions:

```
>>> def echo(value=None):
...     print "Execution starts when 'next()' is called for the first time."
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except GeneratorExit:
...                 # never catch GeneratorExit
...                 raise
...             except Exception, e:
...                 value = e
...     finally:
...         print "Don't forget to clean up when 'close()' is called."
...
>>> generator = echo(1)
>>> print generator.next()
Execution starts when 'next()' is called for the first time.
1
>>> print generator.next()
None
>>> print generator.send(2)
2
>>> generator.throw(TypeError, "spam")
TypeError('spam',)
>>> generator.close()
Don't forget to clean up when 'close()' is called.
```

See Also:

PEP 0342, "*Coroutines via Enhanced Generators*"
The proposal to enhance the API and syntax of generators, making them usable as simple coroutines.

5.3 Primaries

Primaries represent the most tightly bound operations of the language. Their syntax is:

```
primary ::= atom | attributeref | subscription | slicing | call
```

5.3.1 Attribute references

An attribute reference is a primary followed by a period and a name:

```
attributeref  ::=  primary "." identifier
```

The primary must evaluate to an object of a type that supports attribute references, e.g., a module, list, or an instance. This object is then asked to produce the attribute whose name is the identifier. If this attribute is not available, the exception `AttributeError` is raised. Otherwise, the type and value of the object produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

5.3.2 Subscriptions

A subscription selects an item of a sequence (string, tuple or list) or mapping (dictionary) object:

```
subscription  ::=  primary "[" expression_list "]"
```

The primary must evaluate to an object of a sequence or mapping type.

If the primary is a mapping, the expression list must evaluate to an object whose value is one of the keys of the mapping, and the subscription selects the value in the mapping that corresponds to that key. (The expression list is a tuple except if it has exactly one item.)

If the primary is a sequence, the expression (list) must evaluate to a plain integer. If this value is negative, the length of the sequence is added to it (so that, e.g., `x[-1]` selects the last item of `x`.) The resulting value must be a nonnegative integer less than the number of items in the sequence, and the subscription selects the item whose index is that value (counting from zero).

A string's items are characters. A character is not a separate data type but a string of exactly one character.

5.3.3 Slicings

A slicing selects a range of items in a sequence object (e.g., a string, tuple or list). Slicings may be used as expressions or as targets in assignment or `del` statements. The syntax for a slicing:

```
slicing           ::=  simple_slicing | extended_slicing
simple_slicing    ::=  primary "[" short_slice "]"
extended_slicing  ::=  primary "[" slice_list "]"
slice_list        ::=  slice_item ("," slice_item)* [","]
slice_item        ::=  expression | proper_slice | ellipsis
proper_slice      ::=  short_slice | long_slice
short_slice       ::=  [lower_bound] ":" [upper_bound]
long_slice        ::=  short_slice ":" [stride]
lower_bound       ::=  expression
upper_bound       ::=  expression
stride            ::=  expression
ellipsis          ::=  "..."
```

There is ambiguity in the formal syntax here: anything that looks like an expression list also looks like a slice list, so any subscription can be interpreted as a slicing. Rather than further complicating the syntax, this is disambiguated by defining that in this case the interpretation as a subscription takes priority over the interpretation as a slicing (this is the case if the slice list contains no proper slice nor ellipses). Similarly, when the slice list has exactly one short slice and no trailing comma, the interpretation as a simple slicing takes priority over that as an extended slicing.

The semantics for a simple slicing are as follows. The primary must evaluate to a sequence object. The lower and upper bound expressions, if present, must evaluate to plain integers; defaults are zero and the `sys.maxint`, respectively. If either bound is negative, the sequence's length is added to it. The slicing now selects all items with index k such that i

<= *k* < *j* where *i* and *j* are the specified lower and upper bounds. This may be an empty sequence. It is not an error if *i* or *j* lie outside the range of valid indexes (such items don't exist so they aren't selected).

The semantics for an extended slicing are as follows. The primary must evaluate to a mapping object, and it is indexed with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of an ellipsis slice item is the built-in `Ellipsis` object. The conversion of a proper slice is a slice object (see section 3.2) whose `start`, `stop` and `step` attributes are the values of the expressions given as lower bound, upper bound and stride, respectively, substituting `None` for missing expressions.

5.3.4 Calls

A call calls a callable object (e.g., a function) with a possibly empty series of arguments:

```
call                  ::=  primary "(" [argument_list [","]
                            | expression genexpr_for] ")"
argument_list         ::=  positional_arguments ["," keyword_arguments]
                            ["," "*" expression]
                            ["," "**" expression]
                           | keyword_arguments ["," "*" expression]
                            ["," "**" expression]
                           | "*" expression ["," "**" expression]
                           | "**" expression
positional_arguments  ::=  expression ("," expression)*
keyword_arguments     ::=  keyword_item ("," keyword_item)*
keyword_item          ::=  identifier "=" expression
```

A trailing comma may be present after the positional and keyword arguments but does not affect the semantics.

The primary must evaluate to a callable object (user-defined functions, built-in functions, methods of built-in objects, class objects, methods of class instances, and certain class instances themselves are callable; extensions may define additional callable object types). All argument expressions are evaluated before the call is attempted. Please refer to section 7.6 for the syntax of formal parameter lists.

If keyword arguments are present, they are first converted to positional arguments, as follows. First, a list of unfilled slots is created for the formal parameters. If there are N positional arguments, they are placed in the first N slots. Next, for each keyword argument, the identifier is used to determine the corresponding slot (if the identifier is the same as the first formal parameter name, the first slot is used, and so on). If the slot is already filled, a `TypeError` exception is raised. Otherwise, the value of the argument is placed in the slot, filling it (even if the expression is `None`, it fills the slot). When all arguments have been processed, the slots that are still unfilled are filled with the corresponding default value from the function definition. (Default values are calculated, once, when the function is defined; thus, a mutable object such as a list or dictionary used as default value will be shared by all calls that don't specify an argument value for the corresponding slot; this should usually be avoided.) If there are any unfilled slots for which no default value is specified, a `TypeError` exception is raised. Otherwise, the list of filled slots is used as the argument list for the call.

If there are more positional arguments than there are formal parameter slots, a `TypeError` exception is raised, unless a formal parameter using the syntax '`*identifier`' is present; in this case, that formal parameter receives a tuple containing the excess positional arguments (or an empty tuple if there were no excess positional arguments).

If any keyword argument does not correspond to a formal parameter name, a `TypeError` exception is raised, unless a formal parameter using the syntax '`**identifier`' is present; in this case, that formal parameter receives a dictionary containing the excess keyword arguments (using the keywords as keys and the argument values as corresponding values), or a (new) empty dictionary if there were no excess keyword arguments.

If the syntax '`*expression`' appears in the function call, '`expression`' must evaluate to a sequence. Elements from this sequence are treated as if they were additional positional arguments; if there are postional arguments *x1*,...,*xN*

, and '`expression`' evaluates to a sequence *y1*,...,*yM*, this is equivalent to a call with M+N positional arguments *x1*,...,*xN*,*y1*,...,*yM*.

A consequence of this is that although the '`*expression`' syntax appears *after* any keyword arguments, it is processed *before* the keyword arguments (and the '`**expression`' argument, if any – see below). So:

```
>>> def f(a, b):
...  print a, b
...
>>> f(b=1, *(2,))
2 1
>>> f(a=1, *(2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
1 2
```

It is unusual for both keyword arguments and the '`*expression`' syntax to be used in the same call, so in practice this confusion does not arise.

If the syntax '`**expression`' appears in the function call, '`expression`' must evaluate to a (subclass of) dictionary, the contents of which are treated as additional keyword arguments. In the case of a keyword appearing in both '`expression`' and as an explicit keyword argument, a `TypeError` exception is raised.

Formal parameters using the syntax '`*identifier`' or '`**identifier`' cannot be used as positional argument slots or as keyword argument names. Formal parameters using the syntax '`(sublist)`' cannot be used as keyword argument names; the outermost sublist corresponds to a single unnamed argument slot, and the argument value is assigned to the sublist using the usual tuple assignment rules after all other parameter processing is done.

A call always returns some value, possibly `None`, unless it raises an exception. How this value is computed depends on the type of the callable object.

If it is—

a user-defined function: The code block for the function is executed, passing it the argument list. The first thing the code block will do is bind the formal parameters to the arguments; this is described in section 7.6. When the code block executes a `return` statement, this specifies the return value of the function call.

a built-in function or method: The result is up to the interpreter; see the *Python Library Reference* for the descriptions of built-in functions and methods.

a class object: A new instance of that class is returned.

a class instance method: The corresponding user-defined function is called, with an argument list that is one longer than the argument list of the call: the instance becomes the first argument.

a class instance: The class must define a `__call__()` method; the effect is then the same as if that method was called.

5.4 The power operator

The power operator binds more tightly than unary operators on its left; it binds less tightly than unary operators on its right. The syntax is:

```
power  ::=  primary ["**" u_expr]
```

Thus, in an unparenthesized sequence of power and unary operators, the operators are evaluated from right to left (this does not constrain the evaluation order for the operands).

The power operator has the same semantics as the built-in `pow()` function, when called with two arguments: it yields its left argument raised to the power of its right argument. The numeric arguments are first converted to a common type. The result type is that of the arguments after coercion.

With mixed operand types, the coercion rules for binary arithmetic operators apply. For int and long int operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns `100`, but `10**-2` returns `0.01`. (This last feature was added in Python 2.2. In Python 2.1 and before, if both arguments were of integer types and the second argument was negative, an exception was raised).

Raising `0.0` to a negative power results in a `ZeroDivisionError`. Raising a negative number to a fractional power results in a `ValueError`.

5.5 Unary arithmetic operations

All unary arithmetic (and bit-wise) operations have the same priority:

```
u_expr  ::=  power | "-" u_expr | "+" u_expr | "~" u_expr
```

The unary `-` (minus) operator yields the negation of its numeric argument.

The unary `+` (plus) operator yields its numeric argument unchanged.

The unary `~` (invert) operator yields the bit-wise inversion of its plain or long integer argument. The bit-wise inversion of `x` is defined as `-(x+1)`. It only applies to integral numbers.

In all three cases, if the argument does not have the proper type, a `TypeError` exception is raised.

5.6 Binary arithmetic operations

The binary arithmetic operations have the conventional priority levels. Note that some of these operations also apply to certain non-numeric types. Apart from the power operator, there are only two levels, one for multiplicative operators and one for additive operators:

```
m_expr  ::=  u_expr | m_expr "*" u_expr | m_expr "//" u_expr | m_expr "/" u_expr
             | m_expr "%" u_expr
a_expr  ::=  m_expr | a_expr "+" m_expr | a_expr "-" m_expr
```

The `*` (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be an integer (plain or long) and the other must be a sequence. In the former case, the numbers are converted to a common type and then multiplied together. In the latter case, sequence repetition is performed; a negative repetition factor yields an empty sequence.

The `/` (division) and `//` (floor division) operators yield the quotient of their arguments. The numeric arguments are first converted to a common type. Plain or long integer division yields an integer of the same type; the result is that of mathematical division with the 'floor' function applied to the result. Division by zero raises the `ZeroDivisionError` exception.

The `%` (modulo) operator yields the remainder from the division of the first argument by the second. The numeric arguments are first converted to a common type. A zero right argument raises the `ZeroDivisionError` exception. The arguments may be floating point numbers, e.g., `3.14%0.7` equals `0.34` (since `3.14` equals `4*0.7 + 0.34`.) The modulo operator always yields a result with the same sign as its second operand (or zero); the absolute value of the result is strictly smaller than the absolute value of the second operand[2].

The integer division and modulo operators are connected by the following identity: `x == (x/y)*y + (x%y)`. Integer division and modulo are also connected with the built-in function `divmod()`: `divmod(x, y) == (x/y, x%y)`. These identities don't hold for floating point numbers; there similar identities hold approximately where `x/y` is replaced by `floor(x/y)` or `floor(x/y) - 1`[3].

In addition to performing the modulo operation on numbers, the `%` operator is also overloaded by string and unicode objects to perform string formatting (also known as interpolation). The syntax for string formatting is described in the *Python Library Reference*, section "Sequence Types".

Deprecated since release 2.3. The floor division operator, the modulo operator, and the `divmod()` function are no longer defined for complex numbers. Instead, convert to a floating point number using the `abs()` function if appropriate.

The + (addition) operator yields the sum of its arguments. The arguments must either both be numbers or both sequences of the same type. In the former case, the numbers are converted to a common type and then added together. In the latter case, the sequences are concatenated.

The - (subtraction) operator yields the difference of its arguments. The numeric arguments are first converted to a common type.

5.7 Shifting operations

The shifting operations have lower priority than the arithmetic operations:

```
shift_expr ::= a_expr | shift_expr ( "<<" | ">>" ) a_expr
```

These operators accept plain or long integers as arguments. The arguments are converted to a common type. They shift the first argument to the left or right by the number of bits given by the second argument.

A right shift by *n* bits is defined as division by `pow(2,n)`. A left shift by *n* bits is defined as multiplication with `pow(2,n)`; for plain integers there is no overflow check so in that case the operation drops bits and flips the sign if the result is not less than `pow(2,31)` in absolute value. Negative shift counts raise a `ValueError` exception.

5.8 Binary bit-wise operations

Each of the three bitwise operations has a different priority level:

```
and_expr ::= shift_expr | and_expr "&" shift_expr
xor_expr ::= and_expr | xor_expr "^" and_expr
or_expr  ::= xor_expr | or_expr "|" xor_expr
```

The `&` operator yields the bitwise AND of its arguments, which must be plain or long integers. The arguments are converted to a common type.

The `^` operator yields the bitwise XOR (exclusive OR) of its arguments, which must be plain or long integers. The arguments are converted to a common type.

The | operator yields the bitwise (inclusive) OR of its arguments, which must be plain or long integers. The arguments are converted to a common type.

[2] While `abs(x%y) < abs(y)` is true mathematically, for floats it may not be true numerically due to roundoff. For example, and assuming a platform on which a Python float is an IEEE 754 double-precision number, in order that `-1e-100 %1e100` have the same sign as `1e100`, the computed result is `-1e-100 + 1e100`, which is numerically exactly equal to `1e100`. Function `fmod()` in the `math` module returns a result whose sign matches the sign of the first argument instead, and so returns `-1e-100` in this case. Which approach is more appropriate depends on the application.

[3] If x is very close to an exact integer multiple of y, it's possible for `floor(x/y)` to be one larger than `(x-x%y)/y` due to rounding. In such cases, Python returns the latter result, in order to preserve that `divmod(x,y)[0] * y + x % y` be very close to `x`.

5.9 Comparisons

Unlike C, all comparison operations in Python have the same priority, which is lower than that of any arithmetic, shifting or bitwise operation. Also unlike C, expressions like `a < b < c` have the interpretation that is conventional in mathematics:

```
comparison     ::=  or_expr ( comp_operator or_expr )*
comp_operator  ::=  "<" | ">" | "==" | ">=" | "<=" | "<>" | "!="
                    | "is" ["not"] | ["not"] "in"
```

Comparisons yield boolean values: `True` or `False`.

Comparisons can be chained arbitrarily, e.g., `x < y <= z` is equivalent to `x < y and y <= z`, except that `y` is evaluated only once (but in both cases `z` is not evaluated at all when `x < y` is found to be false).

Formally, if *a*, *b*, *c*, ..., *y*, *z* are expressions and *opa*, *opb*, ..., *opy* are comparison operators, then *a opa b opb c* ... *y opy z* is equivalent to *a opa b* `and` *b opb c* `and` ... *y opy z*, except that each expression is evaluated at most once.

Note that *a opa b opb c* doesn't imply any kind of comparison between *a* and *c*, so that, e.g., `x < y > z` is perfectly legal (though perhaps not pretty).

The forms <> and `!=` are equivalent; for consistency with C, `!=` is preferred; where `!=` is mentioned below <> is also accepted. The <> spelling is considered obsolescent.

The operators <, >, ==, >=, <=, and `!=` compare the values of two objects. The objects need not have the same type. If both are numbers, they are converted to a common type. Otherwise, objects of different types *always* compare unequal, and are ordered consistently but arbitrarily. You can control comparison behavior of objects of non-builtin types by defining a `__cmp__` method or rich comparison methods like `__gt__`, described in section 3.4.

(This unusual definition of comparison was used to simplify the definition of operations like sorting and the `in` and `not in` operators. In the future, the comparison rules for objects of different types are likely to change.)

Comparison of objects of the same type depends on the type:

- Numbers are compared arithmetically.
- Strings are compared lexicographically using the numeric equivalents (the result of the built-in function `ord()`) of their characters. Unicode and 8-bit strings are fully interoperable in this behavior.
- Tuples and lists are compared lexicographically using comparison of corresponding elements. This means that to compare equal, each element must compare equal and the two sequences must be of the same type and have the same length.

  If not equal, the sequences are ordered the same as their first differing elements. For example, `cmp([1,2,x], [1,2,y])` returns the same as `cmp(x,y)`. If the corresponding element does not exist, the shorter sequence is ordered first (for example, `[1,2] < [1,2,3]`).
- Mappings (dictionaries) compare equal if and only if their sorted (key, value) lists compare equal.[4] Outcomes other than equality are resolved consistently, but are not otherwise defined.[5]
- Most other objects of builtin types compare unequal unless they are the same object; the choice whether one object is considered smaller or larger than another one is made arbitrarily but consistently within one execution of a program.

The operators `in` and `not in` test for set membership. *x* `in` *s* evaluates to true if *x* is a member of the set *s*, and false otherwise. *x* `not in` *s* returns the negation of *x* `in` *s*. The set membership test has traditionally been bound

[4] The implementation computes this efficiently, without constructing lists or sorting.

[5] Earlier versions of Python used lexicographic comparison of the sorted (key, value) lists, but this was very expensive for the common case of comparing for equality. An even earlier version of Python compared dictionaries by identity only, but this caused surprises because people expected to be able to test a dictionary for emptiness by comparing it to `{}`.

to sequences; an object is a member of a set if the set is a sequence and contains an element equal to that object. However, it is possible for an object to support membership tests without being a sequence. In particular, dictionaries support membership testing as a nicer way of spelling *key* `in` *dict*; other mapping types may follow suit.

For the list and tuple types, *x* `in` *y* is true if and only if there exists an index *i* such that *x* `==` *y*`[`*i*`]` is true.

For the Unicode and string types, *x* `in` *y* is true if and only if *x* is a substring of *y*. An equivalent test is `y.find(x) != -1`. Note, *x* and *y* need not be the same type; consequently, `u'ab' in 'abc'` will return `True`. Empty strings are always considered to be a substring of any other string, so `"" in "abc"` will return `True`. Changed in version 2.3: Previously, *x* was required to be a string of length `1`.

For user-defined classes which define the `__contains__()` method, *x* `in` *y* is true if and only if *y*`.__contains__(`*x*`)` is true.

For user-defined classes which do not define `__contains__()` and do define `__getitem__()`, *x* `in` *y* is true if and only if there is a non-negative integer index *i* such that *x* `==` *y*`[`*i*`]`, and all lower integer indices do not raise `IndexError` exception. (If any other exception is raised, it is as if `in` raised that exception).

The operator `not in` is defined to have the inverse true value of `in`.

The operators `is` and `is not` test for object identity: *x* `is` *y* is true if and only if *x* and *y* are the same object. *x* `is not` *y* yields the inverse truth value.

5.10 Boolean operations

Boolean operations have the lowest priority of all Python operations:

```
expression              ::=  conditional_expression | lambda_form
old_expression          ::=  or_test | old_lambda_form
conditional_expression  ::=  or_test ["if" or_test "else" expression]
or_test                 ::=  and_test | or_test "or" and_test
and_test                ::=  not_test | and_test "and" not_test
not_test                ::=  comparison | "not" not_test
```

In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: `False`, `None`, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true.

The operator `not` yields `True` if its argument is false, `False` otherwise.

The expression *x* `if` *C* `else` *y* first evaluates *C* (*not x*); if *C* is true, *x* is evaluated and its value is returned; otherwise, *y* is evaluated and its value is returned. New in version 2.5.

The expression *x* `and` *y* first evaluates *x*; if *x* is false, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

The expression *x* `or` *y* first evaluates *x*; if *x* is true, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

(Note that neither `and` nor `or` restrict the value and type they return to `False` and `True`, but rather return the last evaluated argument. This is sometimes useful, e.g., if `s` is a string that should be replaced by a default value if it is empty, the expression `s or 'foo'` yields the desired value. Because `not` has to invent a value anyway, it does not bother to return a value of the same type as its argument, so e.g., `not 'foo'` yields `False`, not `''`.)

5.11 Lambdas

```
lambda_form      ::=  "lambda" [parameter_list]:  expression
old_lambda_form  ::=  "lambda" [parameter_list]:  old_expression
```

Lambda forms (lambda expressions) have the same syntactic position as expressions. They are a shorthand to create anonymous functions; the expression `lambda` *arguments*`:` *expression* yields a function object. The unnamed object behaves like a function object defined with

```
def name(arguments):
    return expression
```

See section 7.6 for the syntax of parameter lists. Note that functions created with lambda forms cannot contain statements.

5.12 Expression lists

```
expression_list  ::=  expression ( "," expression )* [","]
```

An expression list containing at least one comma yields a tuple. The length of the tuple is the number of expressions in the list. The expressions are evaluated from left to right.

The trailing comma is required only to create a single tuple (a.k.a. a *singleton*); it is optional in all other cases. A single expression without a trailing comma doesn't create a tuple, but rather yields the value of that expression. (To create an empty tuple, use an empty pair of parentheses: `()`.)

5.13 Evaluation order

Python evaluates expressions from left to right. Notice that while evaluating an assignment, the right-hand side is evaluated before the left-hand side.

In the following lines, expressions will be evaluated in the arithmetic order of their suffixes:

```
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
func(expr1, expr2, *expr3, **expr4)
expr3, expr4 = expr1, expr2
```

5.14 Summary

The following table summarizes the operator precedences in Python, from lowest precedence (least binding) to highest precedence (most binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for comparisons, including tests, which all have the same precedence and chain from left to right — see section 5.9 – and exponentiation, which groups from right to left).

Operator	Description
`lambda`	Lambda expression
`or`	Boolean OR
`and`	Boolean AND
`not` *x*	Boolean NOT
`in`, `not in`	Membership tests
`is`, `is not`	Identity tests
`<`, `<=`, `>`, `>=`, `<>`, `!=`, `==`	Comparisons
`|`	Bitwise OR
`^`	Bitwise XOR
`&`	Bitwise AND
`<<`, `>>`	Shifts
`+`, `-`	Addition and subtraction
`*`, `/`, `%`	Multiplication, division, remainder
+*x*, −*x*	Positive, negative
`~`*x*	Bitwise not
`**`	Exponentiation
x`.`*attribute*	Attribute reference
x`[`*index*`]`	Subscription
x`[`*index*`:`*index*`]`	Slicing
f`(`*arguments*`...)`	Function call
`(`*expressions*`...)`	Binding or tuple display
`[`*expressions*`...]`	List display
`{`*key*`:`*datum*`...}`	Dictionary display
`` ` ``*expressions*`` ...` ``	String conversion

CHAPTER

SIX

Simple statements

Simple statements are comprised within a single logical line. Several simple statements may occur on a single line separated by semicolons. The syntax for simple statements is:

```
simple_stmt  ::=  expression_stmt
                  | assert_stmt
                  | assignment_stmt
                  | augmented_assignment_stmt
                  | pass_stmt
                  | del_stmt
                  | print_stmt
                  | return_stmt
                  | yield_stmt
                  | raise_stmt
                  | break_stmt
                  | continue_stmt
                  | import_stmt
                  | global_stmt
                  | exec_stmt
```

6.1 Expression statements

Expression statements are used (mostly interactively) to compute and write a value, or (usually) to call a procedure (a function that returns no meaningful result; in Python, procedures return the value `None`). Other uses of expression statements are allowed and occasionally useful. The syntax for an expression statement is:

```
expression_stmt  ::=  expression_list
```

An expression statement evaluates the expression list (which may be a single expression).

In interactive mode, if the value is not `None`, it is converted to a string using the built-in `repr()` function and the resulting string is written to standard output (see section 6.6) on a line by itself. (Expression statements yielding `None` are not written, so that procedure calls do not cause any output.)

6.2 Assert statements

Assert statements are a convenient way to insert debugging assertions into a program:

```
assert_stmt  ::=  "assert" expression ["," expression]
```

The simple form, '`assert expression`', is equivalent to

```
if __debug__:
   if not expression: raise AssertionError
```

The extended form, '`assert expression1, expression2`', is equivalent to

```
if __debug__:
   if not expression1: raise AssertionError, expression2
```

These equivalences assume that `__debug__` and `AssertionError` refer to the built-in variables with those names. In the current implementation, the built-in variable `__debug__` is `True` under normal circumstances, `False` when optimization is requested (command line option -O). The current code generator emits no code for an assert statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be displayed as part of the stack trace.

Assignments to `__debug__` are illegal. The value for the built-in variable is determined when the interpreter starts.

6.3 Assignment statements

Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects:

```
assignment_stmt  ::=  (target_list "=")+ (expression_list | yield_expression)
target_list      ::=  target ("," target)* [","]
target           ::=  identifier
                      | "(" target_list ")"
                      | "[" target_list "]"
                      | attributeref
                      | subscription
                      | slicing
```

(See section 5.3 for the syntax definitions for the last three symbols.)

An assignment statement evaluates the expression list (remember that this can be a single expression or a comma-separated list, the latter yielding a tuple) and assigns the single resulting object to each of the target lists, from left to right.

Assignment is defined recursively depending on the form of the target (list). When a target is part of a mutable object (an attribute reference, subscription or slicing), the mutable object must ultimately perform the assignment and decide about its validity, and may raise an exception if the assignment is unacceptable. The rules observed by various types and the exceptions raised are given with the definition of the object types (see section 3.2).

Assignment of an object to a target list is recursively defined as follows.

- If the target list is a single target: The object is assigned to that target.
- If the target list is a comma-separated list of targets: The object must be a sequence with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets. (This rule is relaxed as of Python 1.5; in earlier versions, the object had to be a tuple. Since strings are sequences, an assignment like '`a, b = "xy"`' is now legal as long as the string has the right length.)

Assignment of an object to a single target is recursively defined as follows.

- If the target is an identifier (name):

  - If the name does not occur in a `global` statement in the current code block: the name is bound to the object in the current local namespace.
  - Otherwise: the name is bound to the object in the current global namespace.

  The name is rebound if it was already bound. This may cause the reference count for the object previously bound to the name to reach zero, causing the object to be deallocated and its destructor (if it has one) to be called.

- If the target is a target list enclosed in parentheses or in square brackets: The object must be a sequence with the same number of items as there are targets in the target list, and its items are assigned, from left to right, to the corresponding targets.

- If the target is an attribute reference: The primary expression in the reference is evaluated. It should yield an object with assignable attributes; if this is not the case, `TypeError` is raised. That object is then asked to assign the assigned object to the given attribute; if it cannot perform the assignment, it raises an exception (usually but not necessarily `AttributeError`).

- If the target is a subscription: The primary expression in the reference is evaluated. It should yield either a mutable sequence object (such as a list) or a mapping object (such as a dictionary). Next, the subscript expression is evaluated.

  If the primary is a mutable sequence object (such as a list), the subscript must yield a plain integer. If it is negative, the sequence's length is added to it. The resulting value must be a nonnegative integer less than the sequence's length, and the sequence is asked to assign the assigned object to its item with that index. If the index is out of range, `IndexError` is raised (assignment to a subscripted sequence cannot add new items to a list).

  If the primary is a mapping object (such as a dictionary), the subscript must have a type compatible with the mapping's key type, and the mapping is then asked to create a key/datum pair which maps the subscript to the assigned object. This can either replace an existing key/value pair with the same key value, or insert a new key/value pair (if no key with the same value existed).

- If the target is a slicing: The primary expression in the reference is evaluated. It should yield a mutable sequence object (such as a list). The assigned object should be a sequence object of the same type. Next, the lower and upper bound expressions are evaluated, insofar they are present; defaults are zero and the sequence's length. The bounds should evaluate to (small) integers. If either bound is negative, the sequence's length is added to it. The resulting bounds are clipped to lie between zero and the sequence's length, inclusive. Finally, the sequence object is asked to replace the slice with the items of the assigned sequence. The length of the slice may be different from the length of the assigned sequence, thus changing the length of the target sequence, if the object allows it.

(In the current implementation, the syntax for targets is taken to be the same as for expressions, and invalid syntax is rejected during the code generation phase, causing less detailed error messages.)

WARNING: Although the definition of assignment implies that overlaps between the left-hand side and the right-hand side are 'safe' (for example '`a, b = b, a`' swaps two variables), overlaps *within* the collection of assigned-to variables are not safe! For instance, the following program prints '`[0, 2]`':

```
x = [0, 1]
i = 0
i, x[i] = 1, 2
print x
```

6.3.1 Augmented assignment statements

Augmented assignment is the combination, in a single statement, of a binary operation and an assignment statement:

```
augmented_assignment_stmt  ::=  target augop (expression_list | yield_expression)
augop                      ::=  "+=" | "-=" | "*=" | "/=" | "%=" | "**="
                                | ">>=" | "<<=" | "&=" | "^=" | "|="
```

(See section 5.3 for the syntax definitions for the last three symbols.)

An augmented assignment evaluates the target (which, unlike normal assignment statements, cannot be an unpacking) and the expression list, performs the binary operation specific to the type of assignment on the two operands, and assigns the result to the original target. The target is only evaluated once.

An augmented assignment expression like `x += 1` can be rewritten as `x = x + 1` to achieve a similar, but not exactly equal effect. In the augmented version, `x` is only evaluated once. Also, when possible, the actual operation is performed *in-place*, meaning that rather than creating a new object and assigning that to the target, the old object is modified instead.

With the exception of assigning to tuples and multiple targets in a single statement, the assignment done by augmented assignment statements is handled the same way as normal assignments. Similarly, with the exception of the possible *in-place* behavior, the binary operation performed by augmented assignment is the same as the normal binary operations.

For targets which are attribute references, the initial value is retrieved with a `getattr()` and the result is assigned with a `setattr()`. Notice that the two methods do not necessarily refer to the same variable. When `getattr()` refers to a class variable, `setattr()` still writes to an instance variable. For example:

```
class A:
    x = 3    # class variable
a = A()
a.x += 1     # writes a.x as 4 leaving A.x as 3
```

6.4 The `pass` statement

```
pass_stmt  ::=  "pass"
```

`pass` is a null operation — when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed, for example:

```
def f(arg): pass    # a function that does nothing (yet)

class C: pass       # a class with no methods (yet)
```

6.5 The `del` statement

```
del_stmt  ::=  "del" target_list
```

Deletion is recursively defined very similar to the way assignment is defined. Rather that spelling it out in full details, here are some hints.

Deletion of a target list recursively deletes each target, from left to right.

Deletion of a name removes the binding of that name from the local or global namespace, depending on whether the name occurs in a `global` statement in the same code block. If the name is unbound, a `NameError` exception will be raised.

It is illegal to delete a name from the local namespace if it occurs as a free variable in a nested block.

Deletion of attribute references, subscriptions and slicings is passed to the primary object involved; deletion of a slicing is in general equivalent to assignment of an empty slice of the right type (but even this is determined by the sliced object).

6.6 The `print` statement

```
print_stmt  ::=  "print" ([expression ("," expression)* [","]
                 | ">>" expression [("," expression)+ [","]])
```

`print` evaluates each expression in turn and writes the resulting object to standard output (see below). If an object is not a string, it is first converted to a string using the rules for string conversions. The (resulting or original) string is then written. A space is written before each object is (converted and) written, unless the output system believes it is positioned at the beginning of a line. This is the case (1) when no characters have yet been written to standard output, (2) when the last character written to standard output is '`\n`', or (3) when the last write operation on standard output was not a `print` statement. (In some cases it may be functional to write an empty string to standard output for this reason.) **Note:** Objects which act like file objects but which are not the built-in file objects often do not properly emulate this aspect of the file object's behavior, so it is best not to rely on this.

A '`\n`' character is written at the end, unless the `print` statement ends with a comma. This is the only action if the statement contains just the keyword `print`.

Standard output is defined as the file object named `stdout` in the built-in module `sys`. If no such object exists, or if it does not have a `write()` method, a `RuntimeError` exception is raised.

`print` also has an extended form, defined by the second portion of the syntax described above. This form is sometimes referred to as "`print` chevron." In this form, the first expression after the >> must evaluate to a "file-like" object, specifically an object that has a `write()` method as described above. With this extended form, the subsequent expressions are printed to this file object. If the first expression evaluates to `None`, then `sys.stdout` is used as the file for output.

6.7 The `return` statement

```
return_stmt  ::=  "return" [expression_list]
```

`return` may only occur syntactically nested in a function definition, not within a nested class definition.

If an expression list is present, it is evaluated, else `None` is substituted.

`return` leaves the current function call with the expression list (or `None`) as return value.

When `return` passes control out of a `try` statement with a `finally` clause, that `finally` clause is executed before really leaving the function.

In a generator function, the `return` statement is not allowed to include an `expression_list`. In that context, a bare `return` indicates that the generator is done and will cause `StopIteration` to be raised.

6.8 The `yield` statement

```
yield_stmt  ::=  yield_expression
```

The `yield` statement is only used when defining a generator function, and is only used in the body of the generator function. Using a `yield` statement in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

When a generator function is called, it returns an iterator known as a generator iterator, or more commonly, a generator.

The body of the generator function is executed by calling the generator's `next()` method repeatedly until it raises an exception.

When a `yield` statement is executed, the state of the generator is frozen and the value of `expression_list` is returned to `next()`'s caller. By "frozen" we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, and the internal evaluation stack: enough information is saved so that the next time `next()` is invoked, the function can proceed exactly as if the `yield` statement were just another external call.

As of Python version 2.5, the `yield` statement is now allowed in the `try` clause of a `try` ... `finally` construct. If the generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), the generator-iterator's `close()` method will be called, allowing any pending `finally` clauses to execute.

Note: In Python 2.2, the `yield` statement is only allowed when the `generators` feature has been enabled. It will always be enabled in Python 2.3. This `__future__` import statement can be used to enable the feature:

```
from __future__ import generators
```

See Also:

PEP 0255, "*Simple Generators*"
: The proposal for adding generators and the `yield` statement to Python.

PEP 0342, "*Coroutines via Enhanced Generators*"
: The proposal that, among other generator enhancements, proposed allowing `yield` to appear inside a `try` ... `finally` block.

6.9 The `raise` statement

```
raise_stmt  ::=  "raise" [expression ["," expression ["," expression]]]
```

If no expressions are present, `raise` re-raises the last exception that was active in the current scope. If no exception is active in the current scope, a `TypeError` exception is raised indicating that this is an error (if running under IDLE, a `Queue.Empty` exception is raised instead).

Otherwise, `raise` evaluates the expressions to get three objects, using `None` as the value of omitted expressions. The first two objects are used to determine the *type* and *value* of the exception.

If the first object is an instance, the type of the exception is the class of the instance, the instance itself is the value, and the second object must be `None`.

If the first object is a class, it becomes the type of the exception. The second object is used to determine the exception value: If it is an instance of the class, the instance becomes the exception value. If the second object is a tuple, it is used as the argument list for the class constructor; if it is `None`, an empty argument list is used, and any other object is treated as a single argument to the constructor. The instance so created by calling the constructor is used as the exception value.

If a third object is present and not `None`, it must be a traceback object (see section 3.2), and it is substituted instead of the current location as the place where the exception occurred. If the third object is present and not a traceback object or `None`, a `TypeError` exception is raised. The three-expression form of `raise` is useful to re-raise an exception transparently in an except clause, but `raise` with no expressions should be preferred if the exception to be re-raised was the most recently active exception in the current scope.

Additional information on exceptions can be found in section 4.2, and information about handling exceptions is in section 7.4.

6.10 The `break` statement

```
break_stmt  ::=  "break"
```

`break` may only occur syntactically nested in a `for` or `while` loop, but not nested in a function or class definition within that loop.

It terminates the nearest enclosing loop, skipping the optional `else` clause if the loop has one.

If a `for` loop is terminated by `break`, the loop control target keeps its current value.

When `break` passes control out of a `try` statement with a `finally` clause, that `finally` clause is executed before really leaving the loop.

6.11 The `continue` statement

```
continue_stmt  ::=  "continue"
```

`continue` may only occur syntactically nested in a `for` or `while` loop, but not nested in a function or class definition or `finally` statement within that loop.[1] It continues with the next cycle of the nearest enclosing loop.

6.12 The `import` statement

```
import_stmt      ::=  "import" module ["as" name] ( "," module ["as" name] )*
                      | "from" relative_module "import" identifier ["as" name]
                       ( "," identifier ["as" name] )*
                      | "from" relative_module "import" "(" identifier ["as" name]
                       ( "," identifier ["as" name] )* [","] ")"
                      | "from" module "import" "*"
module           ::=  (identifier ".")* identifier
relative_module  ::=  "."* module | "."+
name             ::=  identifier
```

Import statements are executed in two steps: (1) find a module, and initialize it if necessary; (2) define a name or names in the local namespace (of the scope where the `import` statement occurs). The first form (without `from`) repeats these steps for each identifier in the list. The form with `from` performs step (1) once, and then performs step (2) repeatedly.

In this context, to "initialize" a built-in or extension module means to call an initialization function that the module must provide for the purpose (in the reference implementation, the function's name is obtained by prepending string "init" to the module's name); to "initialize" a Python-coded module means to execute the module's body.

The system maintains a table of modules that have been or are being initialized, indexed by module name. This table is accessible as `sys.modules`. When a module name is found in this table, step (1) is finished. If not, a search for a module definition is started. When a module is found, it is loaded. Details of the module searching and loading process are implementation and platform specific. It generally involves searching for a "built-in" module with the given name and then searching a list of locations given as `sys.path`.

If a built-in module is found, its built-in initialization code is executed and step (1) is finished. If no matching file is found, `ImportError` is raised. If a file is found, it is parsed, yielding an executable code block. If a syntax error occurs, `SyntaxError` is raised. Otherwise, an empty module of the given name is created and inserted in the module table, and then the code block is executed in the context of this module. Exceptions during this execution terminate step (1).

[1] It may occur within an `except` or `else` clause. The restriction on occurring in the `try` clause is implementor's laziness and will eventually be lifted.

When step (1) finishes without raising an exception, step (2) can begin.

The first form of `import` statement binds the module name in the local namespace to the module object, and then goes on to import the next identifier, if any. If the module name is followed by `as`, the name following `as` is used as the local name for the module.

The `from` form does not bind the module name: it goes through the list of identifiers, looks each one of them up in the module found in step (1), and binds the name in the local namespace to the object thus found. As with the first form of `import`, an alternate local name can be supplied by specifying "`as` localname". If a name is not found, `ImportError` is raised. If the list of identifiers is replaced by a star ('`*`'), all public names defined in the module are bound in the local namespace of the `import` statement..

The *public names* defined by a module are determined by checking the module's namespace for a variable named `__all__`; if defined, it must be a sequence of strings which are names defined or imported by that module. The names given in `__all__` are all considered public and are required to exist. If `__all__` is not defined, the set of public names includes all names found in the module's namespace which do not begin with an underscore character ('_'). `__all__` should contain the entire public API. It is intended to avoid accidentally exporting items that are not part of the API (such as library modules which were imported and used within the module).

The `from` form with '`*`' may only occur in a module scope. If the wild card form of import — '`import *`' — is used in a function and the function contains or is a nested block with free variables, the compiler will raise a `SyntaxError`.

Hierarchical module names: when the module names contains one or more dots, the module search path is carried out differently. The sequence of identifiers up to the last dot is used to find a "package"; the final identifier is then searched inside the package. A package is generally a subdirectory of a directory on `sys.path` that has a file '__init__-.py'. [XXX Can't be bothered to spell this out right now; see the URL http://www.python.org/doc/essays/packages.html for more details, also about how the module search works from inside a package.]

The built-in function `__import__()` is provided to support applications that determine which modules need to be loaded dynamically; refer to Built-in Functions in the *Python Library Reference* for additional information.

6.12.1 Future statements

A *future statement* is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python. The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language. It allows use of the new features on a per-module basis before the release in which the feature becomes standard.

```
future_statement  ::=  "from" "__future__" "import" feature ["as" name]
                        ("," feature ["as" name])*
                       | "from" "__future__" "import" "(" feature ["as" name]
                        ("," feature ["as" name])* [","] ")"
feature           ::=  identifier
name              ::=  identifier
```

A future statement must appear near the top of the module. The only lines that can appear before a future statement are:

- the module docstring (if any),
- comments,
- blank lines, and
- other future statements.

The features recognized by Python 2.5 are 'absolute_import', 'division', 'generators', 'nested_-scopes' and 'with_statement'. 'generators' and 'nested_scopes' are redundant in Python version 2.3 and above because they are always enabled.

A future statement is recognized and treated specially at compile time: Changes to the semantics of core constructs are often implemented by generating different code. It may even be the case that a new feature introduces new incompatible syntax (such as a new reserved word), in which case the compiler may need to parse the module differently. Such decisions cannot be pushed off until runtime.

For any given release, the compiler knows which feature names have been defined, and raises a compile-time error if a future statement contains a feature not known to it.

The direct runtime semantics are the same as for any import statement: there is a standard module __future__, described later, and it will be imported in the usual way at the time the future statement is executed.

The interesting runtime semantics depend on the specific feature enabled by the future statement.

Note that there is nothing special about the statement:

```
import __future__ [as name]
```

That is not a future statement; it's an ordinary import statement with no special semantics or syntax restrictions.

Code compiled by an exec statement or calls to the builtin functions compile() and execfile() that occur in a module M containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can, starting with Python 2.2 be controlled by optional arguments to compile() — see the documentation of that function in the *Python Library Reference* for details.

A future statement typed at an interactive interpreter prompt will take effect for the rest of the interpreter session. If an interpreter is started with the **-i** option, is passed a script name to execute, and the script includes a future statement, it will be in effect in the interactive session started after the script is executed.

6.13 The global statement

```
global_stmt  ::=  "global" identifier ("," identifier)*
```

The global statement is a declaration which holds for the entire current code block. It means that the listed identifiers are to be interpreted as globals. It would be impossible to assign to a global variable without global, although free variables may refer to globals without being declared global.

Names listed in a global statement must not be used in the same code block textually preceding that global statement.

Names listed in a global statement must not be defined as formal parameters or in a for loop control target, class definition, function definition, or import statement.

(The current implementation does not enforce the latter two restrictions, but programs should not abuse this freedom, as future implementations may enforce them or silently change the meaning of the program.)

Programmer's note: the global is a directive to the parser. It applies only to code parsed at the same time as the global statement. In particular, a global statement contained in an exec statement does not affect the code block *containing* the exec statement, and code contained in an exec statement is unaffected by global statements in the code containing the exec statement. The same applies to the eval(), execfile() and compile() functions.

6.14 The `exec` statement

```
exec_stmt  ::=  "exec" or_expr ["in" expression ["," expression]]
```

This statement supports dynamic execution of Python code. The first expression should evaluate to either a string, an open file object, or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs). If it is an open file, the file is parsed until EOF and executed. If it is a code object, it is simply executed. In all cases, the code that's executed is expected to be valid as file input (see section 8.2, "File input"). Be aware that the `return` and `yield` statements may not be used outside of function definitions even within the context of code passed to the `exec` statement.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only the first expression after `in` is specified, it should be a dictionary, which will be used for both the global and the local variables. If two expressions are given, they are used for the global and local variables, respectively. If provided, *locals* can be any mapping object. Changed in version 2.4: formerly *locals* was required to be a dictionary.

As a side effect, an implementation may insert additional keys into the dictionaries given besides those corresponding to variable names set by the executed code. For example, the current implementation may add a reference to the dictionary of the built-in module `__builtin__` under the key `__builtins__` (!).

Programmer's hints: dynamic evaluation of expressions is supported by the built-in function `eval()`. The built-in functions `globals()` and `locals()` return the current global and local dictionary, respectively, which may be useful to pass around for use by `exec`.

CHAPTER

SEVEN

Compound statements

Compound statements contain (groups of) other statements; they affect or control the execution of those other statements in some way. In general, compound statements span multiple lines, although in simple incarnations a whole compound statement may be contained in one line.

The `if`, `while` and `for` statements implement traditional control flow constructs. `try` specifies exception handlers and/or cleanup code for a group of statements. Function and class definitions are also syntactically compound statements.

Compound statements consist of one or more 'clauses.' A clause consists of a header and a 'suite.' The clause headers of a particular compound statement are all at the same indentation level. Each clause header begins with a uniquely identifying keyword and ends with a colon. A suite is a group of statements controlled by a clause. A suite can be one or more semicolon-separated simple statements on the same line as the header, following the header's colon, or it can be one or more indented statements on subsequent lines. Only the latter form of suite can contain nested compound statements; the following is illegal, mostly because it wouldn't be clear to which `if` clause a following `else` clause would belong:

```
if test1: if test2: print x
```

Also note that the semicolon binds tighter than the colon in this context, so that in the following example, either all or none of the `print` statements are executed:

```
if x < y < z: print x; print y; print z
```

Summarizing:

```
compound_stmt  ::=  if_stmt
                    | while_stmt
                    | for_stmt
                    | try_stmt
                    | with_stmt
                    | funcdef
                    | classdef
suite          ::=  stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement      ::=  stmt_list NEWLINE | compound_stmt
stmt_list      ::=  simple_stmt (";" simple_stmt)* [";"]
```

Note that statements always end in a `NEWLINE` possibly followed by a `DEDENT`. Also note that optional continuation clauses always begin with a keyword that cannot start a statement, thus there are no ambiguities (the 'dangling `else`' problem is solved in Python by requiring nested `if` statements to be indented).

The formatting of the grammar rules in the following sections places each clause on a separate line for clarity.

7.1 The `if` statement

The `if` statement is used for conditional execution:

```
if_stmt  ::=  "if" expression ":" suite
              ( "elif" expression ":" suite )*
              ["else" ":" suite]
```

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true (see section 5.10 for the definition of true and false); then that suite is executed (and no other part of the `if` statement is executed or evaluated). If all expressions are false, the suite of the `else` clause, if present, is executed.

7.2 The `while` statement

The `while` statement is used for repeated execution as long as an expression is true:

```
while_stmt  ::=  "while" expression ":" suite
                 ["else" ":" suite]
```

This repeatedly tests the expression and, if it is true, executes the first suite; if the expression is false (which may be the first time it is tested) the suite of the `else` clause, if present, is executed and the loop terminates.

A `break` statement executed in the first suite terminates the loop without executing the `else` clause's suite. A `continue` statement executed in the first suite skips the rest of the suite and goes back to testing the expression.

7.3 The `for` statement

The `for` statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

```
for_stmt  ::=  "for" target_list "in" expression_list ":" suite
               ["else" ":" suite]
```

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the `expression_list`. The suite is then executed once for each item provided by the iterator, in the order of ascending indices. Each item in turn is assigned to the target list using the standard rules for assignments, and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty), the suite in the `else` clause, if present, is executed, and the loop terminates.

A `break` statement executed in the first suite terminates the loop without executing the `else` clause's suite. A `continue` statement executed in the first suite skips the rest of the suite and continues with the next item, or with the `else` clause if there was no next item.

The suite may assign to the variable(s) in the target list; this does not affect the next item assigned to it.

The target list is not deleted when the loop is finished, but if the sequence is empty, it will not have been assigned to at all by the loop. Hint: the built-in function `range()` returns a sequence of integers suitable to emulate the effect of Pascal's `for i := a to b do`; e.g., `range(3)` returns the list `[0, 1, 2]`.

Warning: There is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, i.e. lists). An internal counter is used to keep track of which item is used next, and this is incremented on each iteration. When this counter has reached the length of the sequence the loop terminates. This means that if the suite deletes the current (or a previous) item from the sequence, the next item will be skipped (since it gets the index of the current item which has already been treated). Likewise, if the suite inserts an item in the sequence before the current item, the current item will be treated again the next time through the loop. This can lead to nasty bugs that can be avoided by making a temporary copy using a slice of the whole sequence, e.g.,

```
for x in a[:]:
    if x < 0: a.remove(x)
```

7.4 The `try` statement

The `try` statement specifies exception handlers and/or cleanup code for a group of statements:

```
try_stmt   ::=  try1_stmt | try2_stmt
try1_stmt  ::=  "try" ":" suite
                ("except" [expression ["," target]] ":" suite)+
                ["else" ":" suite]
                ["finally" ":" suite]
try2_stmt  ::=  "try" ":" suite
                "finally" ":" suite
```

Changed in version 2.5: In previous versions of Python, `try`...`except`...`finally` did not work. `try`...`except` had to be nested in `try`...`finally`.

The `except` clause(s) specify one or more exception handlers. When no exception occurs in the `try` clause, no exception handler is executed. When an exception occurs in the `try` suite, a search for an exception handler is started. This search inspects the except clauses in turn until one is found that matches the exception. An expression-less except clause, if present, must be last; it matches any exception. For an except clause with an expression, that expression is evaluated, and the clause matches the exception if the resulting object is "compatible" with the exception. An object is compatible with an exception if it is the class or a base class of the exception object, a tuple containing an item compatible with the exception, or, in the (deprecated) case of string exceptions, is the raised string itself (note that the object identities must match, i.e. it must be the same string object, not just a string with the same value).

If no except clause matches the exception, the search for an exception handler continues in the surrounding code and on the invocation stack. [1]

If the evaluation of an expression in the header of an except clause raises an exception, the original search for a handler is canceled and a search starts for the new exception in the surrounding code and on the call stack (it is treated as if the entire `try` statement raised the exception).

When a matching except clause is found, the exception is assigned to the target specified in that except clause, if present, and the except clause's suite is executed. All except clauses must have an executable block. When the end of this block is reached, execution continues normally after the entire try statement. (This means that if two nested handlers exist for the same exception, and the exception occurs in the try clause of the inner handler, the outer handler will not handle the exception.)

Before an except clause's suite is executed, details about the exception are assigned to three variables in the `sys` module: `sys.exc_type` receives the object identifying the exception; `sys.exc_value` receives the exception's parameter; `sys.exc_traceback` receives a traceback object (see section 3.2) identifying the point in the program where the exception occurred. These details are also available through the `sys.exc_info()` function, which returns a tuple `(`*exc_type*`, `*exc_value*`, `*exc_traceback*`)`. Use of the corresponding variables is deprecated in favor of this function, since their use is unsafe in a threaded program. As of Python 1.5, the variables are restored to their previous values (before the call) when returning from a function that handled an exception.

The optional `else` clause is executed if and when control flows off the end of the `try` clause.[2] Exceptions in the `else` clause are not handled by the preceding `except` clauses.

[1] The exception is propogated to the invocation stack only if there is no `finally` clause that negates the exception.

[2] Currently, control "flows off the end" except in the case of an exception or the execution of a `return`, `continue`, or `break` statement.

If `finally` is present, it specifies a 'cleanup' handler. The `try` clause is executed, including any `except` and `else` clauses. If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved. The `finally` clause is executed. If there is a saved exception, it is re-raised at the end of the `finally` clause. If the `finally` clause raises another exception or executes a `return` or `break` statement, the saved exception is lost. The exception information is not available to the program during execution of the `finally` clause.

When a `return`, `break` or `continue` statement is executed in the `try` suite of a `try`...`finally` statement, the `finally` clause is also executed 'on the way out.' A `continue` statement is illegal in the `finally` clause. (The reason is a problem with the current implementation — this restriction may be lifted in the future).

Additional information on exceptions can be found in section 4.2, and information on using the `raise` statement to generate exceptions may be found in section 6.9.

7.5 The `with` statement

New in version 2.5.

The `with` statement is used to wrap the execution of a block with methods defined by a context manager (see section 3.4.9). This allows common `try`...`except`...`finally` usage patterns to be encapsulated for convenient reuse.

```
with_stmt  ::=  "with" expression ["as" target] ":" suite
```

The execution of the `with` statement proceeds as follows:

1. The context expression is evaluated to obtain a context manager.
2. The context manager's `__enter__()` method is invoked.
3. If a target was included in the `with` statement, the return value from `__enter__()` is assigned to it.

   Note: The `with` statement guarantees that if the `__enter__()` method returns without an error, then `__exit__()` will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. See step 5 below.
4. The suite is executed.
5. The context manager's `__exit__()` method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to `__exit__()`. Otherwise, three `None` arguments are supplied.

   If the suite was exited due to an exception, and the return value from the `__exit__()` method was false, the exception is reraised. If the return value was true, the exception is suppressed, and execution continues with the statement following the `with` statement.

   If the suite was exited for any reason other than an exception, the return value from `__exit__()` is ignored, and execution proceeds at the normal location for the kind of exit that was taken.

Note: In Python 2.5, the `with` statement is only allowed when the `with_statement` feature has been enabled. It will always be enabled in Python 2.6. This `__future__` import statement can be used to enable the feature:

```
from __future__ import with_statement
```

See Also:

PEP 0343, "*The "with" statement*"
The specification, background, and examples for the Python `with` statement.

7.6 Function definitions

A function definition defines a user-defined function object (see section 3.2):

```
funcdef         ::=  [decorators] "def" funcname "(" [parameter_list] ")" ":" suite
decorators      ::=  decorator+
decorator       ::=  "@" dotted_name ["(" [argument_list [","]] ")"] NEWLINE
dotted_name     ::=  identifier ("." identifier)*
parameter_list  ::=  (defparameter ",")*
                     (  "*" identifier [, "**" identifier]
                      | "**" identifier
                      | defparameter [","] )
defparameter    ::=  parameter ["=" expression]
sublist         ::=  parameter ("," parameter)* [","]
parameter       ::=  identifier | "(" sublist ")"
funcname        ::=  identifier
```

A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global namespace as the global namespace to be used when the function is called.

The function definition does not execute the function body; this gets executed only when the function is called.

A function definition may be wrapped by one or more decorator expressions. Decorator expressions are evaluated when the function is defined, in the scope that contains the function definition. The result must be a callable, which is invoked with the function object as the only argument. The returned value is bound to the function name instead of the function object. Multiple decorators are applied in nested fashion. For example, the following code:

```
@f1(arg)
@f2
def func(): pass
```

is equivalent to:

```
def func(): pass
func = f1(arg)(f2(func))
```

When one or more top-level parameters have the form *parameter* = *expression*, the function is said to have "default parameter values." For a parameter with a default value, the corresponding argument may be omitted from a call, in which case the parameter's default value is substituted. If a parameter has a default value, all following parameters must also have a default value — this is a syntactic restriction that is not expressed by the grammar.

Default parameter values are evaluated when the function definition is executed. This means that the expression is evaluated once, when the function is defined, and that that same "pre-computed" value is used for each call. This is especially important to understand when a default parameter is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified. This is generally not what was intended. A way around this is to use `None` as the default, and explicitly test for it in the body of the function, e.g.:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section 5.3.4. A function call always assigns values to all parameters mentioned in the parameter list, either from position arguments, from keyword arguments, or from default values. If the form "`*identifier`" is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form "`**identifier`" is present, it is initialized to a new dictionary receiving any excess keyword arguments, defaulting to a new empty dictionary.

It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions. This uses lambda forms, described in section 5.11. Note that the lambda form is merely a shorthand for a simplified function definition; a function defined in a "`def`" statement can be passed around or assigned to another name just like a function defined by a lambda form. The "`def`" form is actually more powerful since it allows the execution of multiple statements.

Programmer's note: Functions are first-class objects. A "`def`" form executed inside a function definition defines a local function that can be returned or passed around. Free variables used in the nested function can access the local variables of the function containing the def. See section 4.1 for details.

7.7 Class definitions

A class definition defines a class object (see section 3.2):

```
classdef     ::=  "class" classname [inheritance] ":" suite
inheritance  ::=  "(" [expression_list] ")"
classname    ::=  identifier
```

A class definition is an executable statement. It first evaluates the inheritance list, if present. Each item in the inheritance list should evaluate to a class object or class type which allows subclassing. The class's suite is then executed in a new execution frame (see section 4.1), using a newly created local namespace and the original global namespace. (Usually, the suite contains only function definitions.) When the class's suite finishes execution, its execution frame is discarded but its local namespace is saved. A class object is then created using the inheritance list for the base classes and the saved local namespace for the attribute dictionary. The class name is bound to this class object in the original local namespace.

Programmer's note: Variables defined in the class definition are class variables; they are shared by all instances. To define instance variables, they must be given a value in the `__init__()` method or in another method. Both class and instance variables are accessible through the notation "`self.name`", and an instance variable hides a class variable with the same name when accessed in this way. Class variables with immutable values can be used as defaults for instance variables. For new-style classes, descriptors can be used to create instance variables with different implementation details.

CHAPTER

EIGHT

Top-level components

The Python interpreter can get its input from a number of sources: from a script passed to it as standard input or as program argument, typed in interactively, from a module source file, etc. This chapter gives the syntax used in these cases.

8.1 Complete Python programs

While a language specification need not prescribe how the language interpreter is invoked, it is useful to have a notion of a complete Python program. A complete Python program is executed in a minimally initialized environment: all built-in and standard modules are available, but none have been initialized, except for `sys` (various system services), `__builtin__` (built-in functions, exceptions and `None`) and `__main__`. The latter is used to provide the local and global namespace for execution of the complete program.

The syntax for a complete Python program is that for file input, described in the next section.

The interpreter may also be invoked in interactive mode; in this case, it does not read and execute a complete program but reads and executes one statement (possibly compound) at a time. The initial environment is identical to that of a complete program; each statement is executed in the namespace of `__main__`.

Under UNIX, a complete program can be passed to the interpreter in three forms: with the **-c** *string* command line option, as a file passed as the first command line argument, or as standard input. If the file or standard input is a tty device, the interpreter enters interactive mode; otherwise, it executes the file as a complete program.

8.2 File input

All input read from non-interactive files has the same form:

```
file_input  ::=  (NEWLINE | statement)*
```

This syntax is used in the following situations:

- when parsing a complete Python program (from a file or from a string);
- when parsing a module;
- when parsing a string passed to the `exec` statement;

8.3 Interactive input

Input in interactive mode is parsed using the following grammar:

```
interactive_input  ::=  [stmt_list] NEWLINE | compound_stmt NEWLINE
```

Note that a (top-level) compound statement must be followed by a blank line in interactive mode; this is needed to help the parser detect the end of the input.

8.4 Expression input

There are two forms of expression input. Both ignore leading whitespace. The string argument to `eval()` must have the following form:

```
eval_input  ::=  expression_list NEWLINE*
```

The input line read by `input()` must have the following form:

```
input_input  ::=  expression_list NEWLINE
```

Note: to read 'raw' input line without interpretation, you can use the built-in function `raw_input()` or the `readline()` method of file objects.

APPENDIX

A

History and License

A.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.2	2.3.1	2003	PSF	yes
2.3.3	2.3.2	2003	PSF	yes
2.3.4	2.3.3	2004	PSF	yes
2.3.5	2.3.4	2005	PSF	yes
2.4	2.3	2004	PSF	yes
2.4.1	2.4	2005	PSF	yes
2.4.2	2.4.1	2005	PSF	yes
2.4.3	2.4.2	2006	PSF	yes
2.4.4	2.4.3	2006	PSF	yes
2.5	2.4	2006	PSF	yes
2.5.1	2.5	2007	PSF	yes
2.5.2	2.5.1	2008	PSF	yes

Note: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

A.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.5.3

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.5.3 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.5.3 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2008 Python Software Foundation; All Rights Reserved" are retained in Python 2.5.3 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.5.3 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.5.3.

4. PSF is making Python 2.5.3 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.5.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.5.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.5.3, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.5.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0
BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013."
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT

SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

A.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

A.3.1 Mersenne Twister

The `_random` module includes code based on a download from http://www.math.keio.ac.jp/ matumoto/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.keio.ac.jp/matumoto/emt.html
email: matumoto@math.keio.ac.jp
```

A.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo`, and `getnameinfo`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/about/index.html.

A.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:

```
 ---------------------------------------------------------------------
/                       Copyright (c) 1996.                            \
|          The Regents of the University of California.                 |
|                        All rights reserved.                           |
|                                                                       |
|   Permission to use, copy, modify, and distribute this software for   |
|   any purpose without fee is hereby granted, provided that this en-   |
|   tire notice is included in all copies of any software which is or   |
|   includes  a  copy  or  modification  of  this software and in all   |
|   copies of the supporting documentation for such software.           |
|                                                                       |
|   This  work was produced at the University of California, Lawrence   |
|   Livermore National Laboratory under  contract  no.  W-7405-ENG-48   |
|   between  the  U.S.  Department  of  Energy and The Regents of the   |
|   University of California for the operation of UC LLNL.              |
|                                                                       |
|                              DISCLAIMER                               |
|                                                                       |
|   This  software was prepared as an account of work sponsored by an   |
|   agency of the United States Government. Neither the United States   |
|   Government  nor the University of California nor any of their em-   |
|   ployees, makes any warranty, express or implied, or  assumes  any   |
|   liability  or  responsibility  for the accuracy, completeness, or   |
|   usefulness of any information,  apparatus,  product,  or  process   |
|   disclosed,   or  represents  that  its  use  would  not  infringe   |
|   privately-owned rights. Reference herein to any specific  commer-   |
|   cial  products,  process,  or  service  by trade name, trademark,   |
|   manufacturer, or otherwise, does not  necessarily  constitute  or   |
|   imply  its endorsement, recommendation, or favoring by the United   |
|   States Government or the University of California. The views  and   |
|   opinions  of authors expressed herein do not necessarily state or   |
|   reflect those of the United States Government or  the  University   |
|   of  California,  and shall not be used for advertising or product   |
\  endorsement purposes.                                               /
 ---------------------------------------------------------------------
```

A.3.4 MD5 message digest algorithm

The source code for the md5 module contains the following notice:

A.3.5 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

A.3.6 Cookie management

The `Cookie` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

A.3.7 Profiling

The `profile` and `pstats` modules contain the following notice:

```
Copyright 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

Permission to use, copy, modify, and distribute this Python software
and its associated documentation for any purpose (subject to the
restriction in the following sentence) without fee is hereby granted,
provided that the above copyright notice appears in all copies, and
that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of InfoSeek not be used in
advertising or publicity pertaining to distribution of the software
without specific, written prior permission.  This permission is
explicitly restricted to the copying and modification of the software
to remain in Python, compiled Python, or other languages (such as C)
wherein the modified or derived code is exclusively imported into a
Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY
SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER
RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF
CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

A.3.8 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

A.3.9 UUencode and UUdecode functions

The `uu` module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with python standard
```

A.3.10 XML Remote Procedure Calls

The `xmlrpclib` module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

INDEX

Symbols

A

B

C

F

G

M

N

Q

R

S

T

U