Python Reference Manual

Guido van Rossum
Corporation for National Research Initiatives (CNRI)
1895 Preston White Drive, Reston, Va 20191, USA
E-mail: guido@cnri.reston.va.us, guido@python.org

October 25, 1996
Release 1.4

ABSTRACT

Python is a simple, yet powerful, interpreted programming language that bridges the gap between C and shell programming, and is thus ideally suited for "throw-away programming" and rapid prototyping. Its syntax is put together from constructs borrowed from a variety of other languages; most prominent are influences from ABC, C, Modula-3 and Icon.

The Python interpreter is easily extended with new functions and data types implemented in C. Python is also suitable as an extension language for highly customizable C applications such as editors or window managers.

Python is available for various systems, amongst which most common flavors of UNIX (including Linux), the Apple Macintosh, MS-DOS, MS-Windows 3.1(1), Windows 95, Windows NT, and OS/2.

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*.

Table of Contents

Table of Contents

Table of Contents

CHAPTER 1: INTRODUCTION

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

While I am trying to be as precise as possible, I have chosen 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 definitition 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, 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 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_letters` 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 used in lexical and grammar rules in this document.)

Each rule begins with a name (which is the name defined by the rule) and a colon. 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 in 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 2: LEXICAL ANALYSIS

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

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

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

2.1 Line structure

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

2.1.1 Logical lines

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

2.1.2 Physical lines

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

2.1.3 Comments

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

2.1.4 Explicit line joining

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

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

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

2.1.5 Implicit line joining

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

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

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

2.1.6 Blank lines

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

2.1.7 Indentation

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

First, tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to there is a multiple of eight (this is intended to be the same rule as used by UNIX). The total number of spaces preceding the first non-blank character then determines the line's indentation. Indentation cannot be split over multiple physical lines using backslashes; the whitespace up to the first backslash determines the indentation.

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

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

Before the first line of the file is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one INDENT token is generated. If it is smaller, it *must* be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a DEDENT token is generated. At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero.

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

```
def perm(l):
        # Compute the list of all permutations of l
    if len(l) <= 1:
                  return [l]
    r = []
    for i in range(len(l)):
```

```
            s = l[:i] + l[i+1:]
            p = perm(s)
            for x in p:
             r.append(l[i:i+1] + x)
    return r
```

The following example shows various indentation errors:

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

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

2.1.8 Whitespace between tokens

Except at the beginning of a logical line or in string literals, the whitespace characters space, tab and formfeed can be used interchangeably to separate tokens. Whitespace is needed between two tokens only if their concatenation could otherwise be interpreted as a different token (e.g., `ab` is one token, but `a b` is two tokens).

2.2 Other tokens

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

2.3 Identifiers and keywords

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

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

Identifiers are unlimited in length. Case is significant.

2.3.1 Keywords

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

```
and         elif        global      not         try
break       else        if          or          while
class       except      import      pass
continue    finally     in          print
```

```
def         for         is          raise
del         from        lambda      return
```

2.3.2 Reserved classes of identifiers

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

Table 1: Special Meanings of Identifiers

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

(XXX need section references here.)

2.4 Literals

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

2.4.1 String literals

String literals are described by the following lexical definitions:

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

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

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

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

Table 2: Escape Sequences

Escape Sequence	Meaning
`\`*newline*	Ignored
`\\`	Backslash (`\`)
`\'`	Single quote (`'`)
`\"`	Double quote (`"`)
`\a`	ASCII Bell (BEL)
`\b`	ASCII Backspace (BS)
`\f`	ASCII Formfeed (FF)
`\n`	ASCII Linefeed (LF)
`\r`	ASCII Carriage Return (CR)
`\t`	ASCII Horizontal Tab (TAB)
`\v`	ASCII Vertical Tab (VT)
`\`*ooo*	ASCII character with octal value *ooo*
`\x`*xx...*	ASCII character with hex value *xx...*

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

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

2.4.1.1 String literal concatenation

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

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

Note that this feature is defined at the syntactical level, but implemented at compile time. The '+' operator must be used to concatenate string expressions at run time.

2.4.2 Numeric literals

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

2.4.2.1 Integer and long integer literals

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

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

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

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

Some examples of plain and long integer literals:

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

2.4.2.2 Floating point literals

Floating point literals are described by the following lexical definitions:

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

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

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

2.4.2.3 Imaginary literals

Imaginary literals are described by the following lexical definitions:

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

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

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

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

2.5 Operators

The following tokens are operators:

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

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

2.6 Delimiters

The following tokens serve as delimiters in the grammar:

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

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

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

```
'       "       #       \
```

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

```
@       $       ?
```

CHAPTER 3: DATA MODEL

3.1 Objects, values and types

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In 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. It determines the operations that an 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. The '==' operator compares the value of two objects. Objects whose value can change are said to be *mutable*; objects whose value is unchangeable once they are created are called *immutable*. An object's (im)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 which collects most objects as soon as they become unreachable, but never collects garbage containing circular references.)

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 always 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 (im)mutability of a container, only the identities of the immediately contained objects are implied. So, if an immutable container (like a tuple) contains a reference to a mutable object, its value changes if that mutable object is changed.

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

3.2 The standard type hierarchy

Below is a list of the types that are built into Python. Extension modules written in C 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. There are also some 'generic' special attributes, not listed with the individual objects: `__methods__` is a list of the method names of a built-in object, if it has any; `__members__` is a list of the data attribute names of a built-in object, if it has any.

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.

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 and floating point numbers:

Integers These represent elements from the mathematical set of whole numbers

There are two 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 falls outside this range, the exception `OverflowError` is raised. 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.

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. For any operation except left shift, if it yields a result in the plain integer domain without causing overflow, it 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 implementation for the accepted range and handling of overflow. Python does not support single-precision floating point numbers; the savings in CPU 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 value of a complex number `z` can be retrieved through the attributes `z.real` and `z.imag`.

Sequences These represent finite ordered sets indexed by natural 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`. 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 again.

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 array 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.

(What should be done on systems whose native character set is not ASCII???)

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 enclosing 'nothing' in 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 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 optional module `array` provides an additional example of a mutable sequence type.

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 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 are created by the `{...}` notation. (See "Dictionary displays" on page 28.)

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

Callable types These are the types to which the function call operation (for invocation, See "Calls" on page 31.) is applied:

User-defined functions A user-defined function object is created by a function definition. (See "Function definitions" on page 48.)

Special read-only attributes: `func_doc` or `__doc__` is the function's documentation string, or `None` if unavailable; `func_name` or `__name__` is the function's name; `func_defaults` is a tuple containing default argument values for those arguments that have defaults, or `None` if no arguments have a default value; `func_code` is the code object representing the compiled function body; `func_globals` is (a reference to) the dictionary that holds the function's global variables — it defines the global name space of the module in which the function was defined. 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 (a.k.a. *object closure*) combines a class, a class instance (or `None`) and a user-defined function.

Special read-only attributes: `im_self` is the instance object; `im_func` is the function object; `im_class` is the class that defined the method (which may be a base class of the class of which `im_self` is an instance); `__doc__` is the method's documentation (same as `im_func.__doc__`); `__name__` is the method name (same as `im_func.__name__`).

User-defined method objects are created in two ways: when getting an attribute of a class that is a user-defined function object, or when getting an attributes of a class instance that is a user-defined function object. In the former case (class attribute), the `im_self` attribute is `None`, and the method object is said to be *unbound*; in the latter case (instance attribute), `im_self` is the instance, and the method object is said to be *bound*. For instance, when `C` is a class which contains a definition for a function `f`, `C.f` does not yield the function object `f`; rather, it yields an unbound method object m where `m.im_class` is `C`, `m.im_function` is `f`, and `m.im_self` is `None`. When `x` is a `C` instance, `x.f` yields a bound method object `m` where `m.im_class` is `C`, `m.im_function` is `f`, and `m.im_self` is `x`.

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)`.

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.

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 paragraph).

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 `list.append`, assuming `list` is a list object. In this case, the special read-only attribute `__self__` is set to the object denoted by `list`.

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 can be called as a function only when the class has a `__call__` method; in this case, `x(arguments)` is a shorthand for `x.__call__(arguments)`.

Modules Modules are imported by the `import` statement. (See "The import statement" on page 43.) A module object has a name space 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 update the module's name space dictionary, e.g. ``m.x = 1'' is equivalent to ``m.__dict__["x"] = 1''.

Special read-only attribute: `__dict__` is the dictionary object that is the module's name space.

Predefined (writable) attributes: `__name__` is the module 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 "Class definitions" on page 49.). A class has a name space 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 their occurrence in the base class list. When a class attribute reference would yield a user-defined function object, it is transformed into an unbound user-defined method object (see above). The `im_class` attribute of this method object is the class in which the function object was found, not necessarily the class for which the attribute reference was initiated.

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

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

Special read-only attributes: `__dict__` is the dictionary that is the class's name space; `__name__` is the class name; `__bases__` is a tuple (possibly empty or a singleton) containing the base classes, in the order of their occurrence in the base class list.

Predefined (writable) attribute: `__doc__` is the class's documentation string, or `None` if undefined.

Class instances A class instance is created by calling a class object as a function (see above). A class instance has a name space implemented as a dictionary, which is the first place where instance attributes are searched. When an attribute is not found there, the search continues with the class attributes. If a class attribute is found that is a user-defined function object (and in no other case), it is transformed into an unbound user-defined method object (see above). The `im_class` attribute of this method object is the class in which the function object was found, not necessarily the class of the instance for which the attribute reference was initiated. 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, mappings, or callable objects, and override various other special operations, if they have methods with certain special names. See "Special method names" on page 18.

Special read-only attributes: `__dict__` yields the attribute dictionary; `__class__` yields the instance's class.

Files A file object represents an open file. It is a wrapper around a C standard I/O (*stdio*) file pointer. File objects are created by the `open()` built-in function, and also by `posix.popen()`, `posix.fdopen()` and the `makefile` method of socket objects. 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 methods of file objects and other details.

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-compile* 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 name space dictionary of 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_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_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_flags is an integer encoding a number of flags for the interpreter. The following flag bits are defined: bit 2 is set if the function uses the "*arguments'' syntax to accept an arbitrary number of positional arguments; bit 3 is set if the function uses the ''**keywords'' syntax to accept arbitrary keyword arguments; other bits are used internally or reserved for future use. The first item in co_consts is the documentation string of the function, or None if undefined. To find out the first line number of a function, you have to disassemble the bytecode instructions; the standard library module codehack defines a function getlineno() that returns the first line number of a code object.

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 locals 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_owner is the class or module that defined the code, if any; f_lineno gives the current line number and f_lasti gives the precise instruction (this is an index into the instruction 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).

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, (See "The try statement" on page 47.), the stack trace is made available to the program as sys.exc_traceback. 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 lowerbound; `stop` is the upperbound; `step` is the step value; each is `None` if omitted. These attributes can have any type.

3.3 Special method names

This section describes how user-defined classes can customize their behavior or emulate the behavior of other object types. In the following, if a class defines a particular method, any class derived from it is also understood to define that method (implicitly).

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. For instance, if a class defines a method named `__getitem__`, and `x` is an instance of this class, then `x[i]` is equivalent to `x.__getitem__(i)`. (The reverse is not true; e.g. if `x` is a list object, `x.__getitem__(i)` is not equivalent to `x[i]`.) Except where mentioned, attempts to execute an operation raise an exception when no appropriate method is defined.

3.3.1 Basic customization

`__init__(self, [args...])` Called when the instance is created. The arguments are those that were passed to the class constructor expression. If a base class has an `__init__` method the derived class's `__init__` method must explicitly call it to ensure proper initialization of the base class part of the instance, e.g.
''`BaseClass.__init__(self, [args...])`''.

`__del__(self)` Called when the instance is about to be destroyed. If a base class has a `__del__` method the derived class's `__del__` method must explicitly call it to ensure proper deletion of the base class part of the instance. e.g. ''`BaseClass.__del__(self)`''. 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 that ''`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 its reference count reaches zero. Some common situations that may prevent the reference count of an object to go 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`.

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 is 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.

`__repr__(self)` Called by the `repr()` built-in function and by string conversions (reverse quotes) to compute the "official" string representation of an object. This should normally look like a valid Python expression that can be used to recreate an object with the same value.

`__str__(self)` Called by the `str()` built-in function and by the `print` statement compute the ''informal'' string representation of an object. This differs from `__repr__` in that it doesn't have to look like a valid Python expression: a more convenient or concise representation may be used instead.

`__cmp__(self, other)` Called by all comparison operations. Should return a negative integer if `self` < `other`, zero if `self` == `other`, a positive integer if `self` > `other`. If no `__cmp__` method is defined, class instances are compared by object identity ("address"). (Implementation note: due to limitations in the interpreter, exceptions raised by comparisons are *ignored*, and the outcome will be random in this case.)

`__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 no `__hash__` method is defined, class instances are hashed by object identity (''address''). If a class does not define a `__cmp__` method it should not define a `__hash__` method either; if it defines `__cmp__` but not `__hash__` its instances will not be usable as dictionary keys. If a class defines mutable objects and implements a `__cmp__` 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).`__nonzero__(self)` Calledto implement truth value testing; should return 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.

3.3.2 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. For performance reasons, these methods are cached in the class object at class definition time; therefore, they cannot be changed after the class definition is executed.

`__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 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).

`__setattr__(self, name, value)` Called whenever an attribute assignment is attempted. This is called instead of the normal mechanism (i.e. instead of storing 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`’’.

`__delattr__(self, name)` Like `__setattr__` but for attribute deletion instead of assignment.

3.3.3 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, ...)`.

3.3.4 Emulating sequence and mapping types

The following methods can be defined to emulate sequence or mapping objects. 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, and the method `__getslice__` (see below) should be defined. It is also recommended that mappings provide methods `keys`, `values` and `items` behaving similar to those for Python's standard dictionary objects; mutable sequences should provide methods `append`, `count`, `index`, `insert`, `sort`, `remove` and `reverse` like Python standard list objects. Finally, sequence types should implement addition (meaning concatenation) and multiplication (meaning repetition) by defining the methods `__add__`, `__radd__`, `__mul__` and `__rmul__` described below; they should not define `__coerce__` or other numerical operators.

`__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]`. Note that the special interpretation of negative keys (if the class wishes to emulate a sequence type) is up to the `__getitem__` method.

`__setitem__(self, key, value)` Called to implement assignment to `self[key]`. Same note as for `__getitem__`.

`__delitem__(self, key)` Called to implement deletion of `self[key]`. Same note as for `__getitem__`.

3.3.4.1 Additional methods for emulation of sequence types

The following methods can be defined to further emulate sequence objects. For immutable sequences methods, only `__getslice__` should be defined; for mutable sequences, all three methods should be defined.

`__getslice__(self, i, j)` 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 0 or `len(self)`, respectively, and `len(self)` has been added (once) to originally negative `i` or `j` by the time this function is called (unlike for `__getitem__`).

`__setslice__(self, i, j, sequence)` Called to implement assignment to `self[i:j]`. The `sequence` argument can have any type. The return value should be `None`. Same notes for `i` and `j` as for `__getslice__`.

`__delslice__(self, i, j)` Called to implement deletion of `self[i:j]`. Same notes for `i` and `j` as for `__getslice__`.

Notice that these methods are only invoked when a single slice with a single colon is used. For slice operations involving *extended slice notation*, `__getitem__`, `__setitem__` or `__delitem__` is called.

3.3.5 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, right)`
`__sub__(self, right)`
`__mul__(self, right)`
`__div__(self, right)`
`__mod__(self, right)`
`__divmod__(self, right)`
`__pow__(self, right)`
`__lshift__(self, right)`
`__rshift__(self, right)`
`__and__(self, right)`
`__xor__(self, right)`
`__or__(self, right)`

These functions 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.

`__radd__(self, left)`
`__rsub__(self, left)`
`__rmul__(self, left)`
`__rdiv__(self, left)`
`__rmod__(self, left)`
`__rdivmod__(self, left)`
`__rpow__(self, left)`
`__rlshift__(self, left)`
`__rrshift__(self, left)`
`__rand__(self, left)`
`__rxor__(self, left)`
`__ror__(self, left)` These functions are called to implement the binary arithmetic operations (`+`, `-`, `*`, `/`, `%`, `divmod()`, `pow()`, <<, >>, `&`, `^`, |) with reversed operands. These functions are only called if the left operand does not support the corresponding operation (possibly after coercion). For instance: to evaluate the expression x+y, where x is an instance of a class that does not have an `__add__` method, `y.__radd(x)` is called. If the class defines a __coerce__ method that coerces its arguments to a common type, these methods will never be called and thus needn't be defined. They are useful for classes that implement semi-numerical data types (types that have some numerical behavior but don't adhere to all invariants usually assumed about numbers).

`__neg__(self)`
`__pos__(self)`

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

`__int__(self)`
`__long__(self)`
`__float__(self)` Called to implement the built-in functions `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.

`__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 no conversion is possible. 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 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).

Coercion rules: to evaluate `x` *op* `y`, the following steps are taken (where `__op__` and `__rop__` are the method names corresponding to *op*, e.g. if *op* is '`+`', `__add__` and `__radd__` are used). If an exception occurs at any point, the evaluation is abandoned and exception handling takes over.

0. If x is a string object and *op* is the modulo operator (`%`), the string formatting operation (see [Ref:XXX]) is invoked and the remaining steps are skipped.

1. If x is a class instance:

   1a. If x has a `__coerce__` method: replace x and y with the 2-tuple returned by `x.__coerce__(y)`; skip to step 2 if the coercion returns `None`.

   1b. If neither x nor y is a class instance after coercion, go to step 3.

   1c. If x has a method `__op__`, return `x.__op__(y)`; otherwise, restore x and y to their value before step 1a.

2. If y is a class instance:

   2a. If y has a `__coerce__` method: replace y and x with the 2-tuple returned by `y.__coerce__(x)`; skip to step 3 if the coercion returns None.

   2b. If neither x nor y is a class instance after coercion, go to step 3.

   2b. If y has a method `__rop__`, return `y.__rop__(x)`; otherwise, restore x and y to their value before step 2a.

3. We only get here if neither x nor y is a class instance.

   3a. If *op* is '`+`' and x is a sequence, sequence concatenation is invoked.

   3b. If *op* is '`*`' and one operand is a sequence and the other an integer, sequence repetition is invoked.

   3c. Otherwise, both operands must be numbers; they are coerced to a common type if possible, and the numeric operation is invoked for that type.

CHAPTER 4: EXECUTION MODEL

4.1 Code blocks, execution frames, and name spaces

A *code block* is a piece of Python program text that can be executed as a unit, such as a module, a class definition or a function body. Some code blocks (like modules) are normally executed only once, others (like function bodies) may be executed many times. Code blocks may textually contain other code blocks. Code blocks may invoke other code blocks (that may or may not be textually contained in them) as part of their execution, e.g. by invoking (calling) a function.

The following are code blocks: A module is a code block. A function body is a code block. A class definition is a code block. Each command typed interactively is a separate code 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 string argument passed to the built-in function `eval` and to the `exec` statement are code blocks. The file read by the built-in function `execfile` is a code block. And finally, the expression read and evaluated by the built-in function `input` is a code block. A code block is executed in an execution frame. An *execution frame* contains some administrative information (used for debugging), determines where and how execution continues after the code block's execution has completed, and (perhaps most importantly) defines two name spaces, the local and the global name space, that affect execution of the code block.

A *name space* is a mapping from names (identifiers) to objects. A particular name space may be referenced by more than one execution frame, and from other places as well. Adding a name to a name space is called *binding* a name (to an object); changing the mapping of a name is called *rebinding*; removing a name is *unbinding*. Name spaces are functionally equivalent to dictionaries (and often implemented as dictionaries).

The *local name space* of an execution frame determines the default place where names are defined and searched. The *global name space* determines the place where names listed in `global` statements are defined and searched, and where names that are not bound anywhere in the current code block are searched.

Whether a name is local or global in a code block is determined by static inspection of the source text for the code block: in the absence of `global` statements, a name that is bound anywhere in the code block is local in the entire code block; all other names are considered global. The `global` statement forces global interpretation of specified names throughout the code block. 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. Local names are searched only on the local name space; global names are searched only in the global and built-in namespace.[1]

A target occurring in a `del` statement is also considered bound for this purpose (though the actual semantics are to "unbind" the name).

---

1. If the code block contains `exec` statements or the construct ''`from ... import *`'', the semantics of local names change subtly: local name lookup first searches in the local namespace, then in the global namespace and in the built-in namespace.

When a global name is not found in the global name space, it is searched in the built-in namespace. The built-in namespace associated with the execution of a code block is actually found by looking up the name `__builtins__` is its global name space; this should be a dictionary or a module (in the latter case its dictionary is used). Normally, the `__builtins__` namespace is the dictionary of the built-in module `__builtin__` (note: no 's'); if it isn't, *restricted execution mode* is in effect, see [Ref:XXX]. When a name is not found at all, a `NameError` exception is raised.

The following table lists the local and global name space used for all types of code blocks. The name space for a particular module is automatically created when the module is first imported. Note that in almost all cases, the global name space is the name space of the containing module — scopes in Python do not nest!

Table 3: Name Spaces for Various Code Blocks

Code block type	Global name space	Local name space	Notes
Module	n.s. for this module	same as global	
Script (file or command)	n.s. for `__main__`	same as global	(1)
Interactive command	n.s. for `__main__`	same as global	
Class definition	global n.s. of containing block	new n.s.	
Function body	global n.s. of containing block	new n.s.	
String passed to `exec` statement	global n.s. of containing block	local n.s. of containing block	(2), (3)
String passed to `eval()`	global n.s. of caller	local n.s. of caller	(2), (3)
File read by `execfile()`	global n.s. of caller	local n.s. of caller	(2), (3)
Expression read by `input`	global n.s. of caller	local n.s. of caller	

Notes:

n.s. means *name space*

(1) The main module for a script is always called `__main__`; ''the filename don't enter into it.''

(2) The global and local name space for these can be overridden with optional extra arguments.

(3) 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.

The built-in functions `globals()` and `locals()` returns a dictionary representing the current global and local name space, respectively. The effect of modifications to this dictionary on the name space are undefined.[1]

1. The current implementations return the dictionary actually used to implement the name space, *except* for functions, where the optimizer may cause the local name space to be implemented differently, and `locals()` returns a dictionary that is a shadow copy of the actual local name space.

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 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 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 this case, the interpreter normally prints a stack backtrace.

Exceptions are identified by string objects or class instances. Selection of a matching `except` clause is based on object identity (i.e. two different string objects with the same value represent different exceptions). For string exceptions, the except clause must reference the same string object. For class exceptions, the except clause must reference the same class or a base class of it.

When an exception is raised, an object (maybe `None`) is passed as the exception's "parameter" or ''value''; this object does not affect the selection of an exception handler, but is passed to the selected exception handler as additional information. For class exceptions, this object must be an instance of the exception class being raised.

See also the description of the `try` and `raise` statements in "Compound statements" on page 45.

CHAPTER 5: 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 the end of chapter 3. 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|dict_display|string_conversion
```

5.2.1 Identifiers (Names)

An identifier occurring as an atom is a reference to a local, global or built-in name binding. If a name is assigned to anywhere in a code block (even in unreachable code), and is not mentioned in a `global` statement in that code block, then it refers to a local name throughout that code block. When it is not assigned to anywhere in the block, or when it is assigned to but also explicitly listed in a `global` statement, it refers to a global name if one exists, else to a built-in name (and this binding may dynamically change).

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

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 “Literals” on page 6 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.

(In the original implementation, all literals in the same code block with the same type and value yield the same object.)

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] "]"
```

A list display yields a new list object. If it has no expression list, the list object has no items. Otherwise, the elements of the expression list are evaluated from left to right and inserted in the list object in that order.

5.2.5 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 "The standard type hierarchy" on page 12 (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.

5.2.6 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.)

It is illegal to attempt to convert recursive objects (e.g. lists or dictionaries that contain a reference to themselves, directly or indirectly.)

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.

5.3 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. 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.

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 (string, tuple or list) object. 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 | ellipses
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
ellipses:         "..."
```

There's an 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 declaring 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 sequence's length, 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 ellipses slice item is the built-in `Ellipses` object. The conversion of a proper slice is a slice object (see page 17) 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 [","]] ")"
argument_list:         positional_arguments ["," keyword_arguments]
                     | keyword_arguments
positional_arguments: expression ("," expression)*
keyword_arguments:    keyword_item ("," keyword_item)*
keyword_item:         identifier "=" expression
```

A trailing comma may be present after an argument list 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 “Function definitions” on page 48 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.

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“Function definitions” on page 48. 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 library reference manual 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.

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: 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; if the result is not expressible in that type (as in raising an integer to a negative power, or a negative floating point number to a broken power), a `TypeError` exception is raised.

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 remaining 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
a_expr:         m_expr | aexpr "+" m_expr | aexpr "-" m_expr
```

The "*" (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be a plain integer 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) operator yields the quotient of its 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 second operand.

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 and complex numbers; there a similar identity holds where `x/y` is replaced by `floor(x/y)`) or `floor((x/y).real)`, respectively.

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 this 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 bit-wise 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.

5.9 Comparisons

Contrary to C, all comparison operations in Python have the same priority, which is lower than that of any arithmetic, shifting or bitwise operation. Also contrary to 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 integer values: 1 for true, 0 for 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 implied.

The operators `"<", ">", "==", ">=", "<="`, and `"!="` compare the values of two objects. The objects needn't 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. (This unusual definition of comparison is done to simplify the definition of operations like sorting and the `in` and `not in` operators.)

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.
- Tuples and lists are compared lexicographically using comparison of corresponding items.
- Mappings (dictionaries) are compared through lexicographic comparison of their sorted (key, value) lists.[1]
- Most other 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 ex-

ecution of a program.

The operators `in` and `not in` test for sequence membership: if *y* is a sequence, *x* in *y* is true if and only if there exists an index *i* such that *x* = *y*[*i*]. *x* not in *y* yields the inverse truth value. The exception `TypeError` is raised when *y* is not a sequence, or when *y* is a string and *x* is not a string of length one.[1]

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:      or_test | lambda_form
or_test:        and_test | or_test "or" and_test
and_test:       not_test | and_test "and" not_test
not_test:       comparison | "not" not_test
lambda_form:"lambda" [parameter_list]: expression
```

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

The operator `not` yields 1 if its argument is false, 0 otherwise.

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 0 and 1, 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 `0`, not `''`.)

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 that behaves virtually identical to one defined with

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

1. This is expensive since it requires sorting the keys first, but about the only sensible definition. An 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 `{}`.
1. The latter restriction is sometimes a nuisance.

See “Function definitions” on page 48 for the syntax of parameter lists. Note that functions created with lambda forms cannot contain statements.

5.11 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 *single*); 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: `()`.)

5.12 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, which chain from left to right — see above).

Table 4: Operator Precedence

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

CHAPTER 6: 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
              | assignment_stmt
              | pass_stmt
              | del_stmt
              | print_stmt
              | return_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 “The print statement” on page 41) on a line by itself. (Expression statements yielding `None` are not written, so that procedure calls do not cause any output.)

6.2 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
target_list:     target ("," target)* [","]
target:          identifier | "(" target_list ")" | "[" target_list "]"
               | attributeref | subscription | slicing
```

(See “Primaries” on page 29 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 “The standard type hierarchy” on page 12.)

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 tuple 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.

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 name space.
    - Otherwise: the name is bound to the object in the current global name space.

  The name is rebound if it was already bound. This can cause the reference count for the object previously bound to the name to reach zero, causing the object to be deallocated and its
- If the target is a target list enclosed in parentheses: the object is assigned to that target list as described above.
- If the target is a target list enclosed in square brackets: the object must be a list with the same number of items as the target list contains targets, 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 (list) object or a mapping (dictionary) object. Next, the subscript expression is evaluated.

  If the primary is a mutable sequence object (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 (dictionary) object, 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 (e.g. 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 se-

quence'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' (e.g. ``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 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)
```

6.4 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 (which must exist) from the local or global name space, depending on whether the name occurs in a `global` statement in the same code 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).

6.5 The `print` statement

```
print_stmt:     "print" [ 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; or (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.)

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 is not a writable file, a RuntimeError exception is raised. (The original implementation attempts to write to the system's original standard output instead, but this is not safe, and should be fixed.)

6.6 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.

6.7 The raise statement

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

raise evaluates its first expression, which must yield a string, class, or instance object. If there is a second expression, this is evaluated, else None is substituted. If the first expression is a class object, then the second expression must be an instance of that class or one of its derivatives. If the first expression is an instance object, the second expression must be None.

If the first object is a class or string, it then raises the exception identified by the first object, with the second one (or None) as its parameter. If the first object is an instance, it raises the exception identified by the class of the object, with the instance as its parameter (and there should be no second object, or the second object should be None).

If a third object is present, and it is not None, it should be a traceback object (see page 17 traceback objects), and it is substituted instead of the current location as the place where the exception occurred. This is useful to re-raise an exception transparently in an except clause.

6.8 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.

6.9 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 `try` statement within that loop.[1] It continues with the next cycle of the nearest enclosing loop.

6.10 The `import` statement

```
import_stmt:     "import" identifier ("," identifier)*
               | "from" identifier "import" identifier ("," identifier)*
             | "from" identifier "import" "*"
```

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 name space (of the scope where the `import` statement occurs). The first form (without `from`) repeats these steps for each identifier in the list, the `from` form performs them once, with the first identifier specifying the module name.

The system maintains a table of modules that have been initialized, indexed by module name. (The current implementation makes this table 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. This first looks for a built-in module definition, and if no built-in module if the given name is found, it searches a user-specified list of directories for a file whose name is the module name with extension `".py"`. (The current implementation uses the list of strings `sys.path` as the search path; it is initialized from the shell environment variable `$PYTHONPATH`, with an installation-dependent default.)

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).

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 name space to the module object, and then goes on to import the next identifier, if any. 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 name space to the object thus found. If a name is not found, `ImportError` is raised. If the list of identifiers is replaced by a star (`*`), all names defined in the module are bound, except those beginning with an underscore(_).

Names bound by import statements may not occur in `global` statements in the same scope.

The `from` form with `*` may only occur in a module scope.

---

1. Except that it may currently occur within an `except` clause.

(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.)

6.11 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. While *using* global names is automatic if they are not defined in the local scope, *assigning* to global names would be impossible without `global`.

Names listed in a `global` statement must not be used in the same code block before that `global` statement is executed.

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.)

Note: the `global` is a directive to the parser. Therefore, 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.

6.12 The `exec` statement

```
exec_stmt:    "exec" expression ["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, 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, both must be dictionaries and they are used for the global and local variables, respectively.

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 7: 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 | 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 "Boolean operations" on page 35 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 (string, tuple or list):

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

The expression list is evaluated once; it should yield a sequence. The suite is then executed once for each item in the sequence, 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:        try_exc_stmt | try_fin_stmt
try_exc_stmt:    "try" ":" suite
                ("except" [expression ["," target]] ":" suite)+
                ["else" ":" suite]
try_fin_stmt:    "try" ":" suite
                "finally" ":" suite
```

There are two forms of `try` statement: `try...except` and `try...finally`. These forms cannot be mixed (but they can be nested in each other).

The `try...except` form specifies one or more exception handlers (the `except` clauses). 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 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 either the object that identifies the exception, or (for exceptions that are classes) it is a base class of the exception, or it is a tuple containing an item that is compatible with the exception. Note that the object identities must match, i.e. it must be the same object, not just an object 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.

If the evaluation of an expression in the header of an except clause raises an exception, the original search for a handler is cancelled 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's parameter is assigned to the target specified in that except clause, if present, and the except clause's suite is executed. When the end of this suite 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 page 17) identifying the point in the program where the exception occurred.

The optional `else` clause is executed when no exception occurs in the `try` clause. Exceptions in the `else` clause are not handled by the preceding `except` clauses.

The `try...finally` form specifies a 'cleanup' handler. The `try` clause is executed. When no exception occurs, the `finally` clause is executed. When an exception occurs in the `try` clause, the exception is temporarily saved, the `finally` clause is executed, and then the saved exception is re-raised. If the `finally` clause raises another exception or executes a `return`, `break` or `continue` statement, the saved exception is lost.

When a `return` or `break` 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 `try` clause. (The reason is a problem with the current implementation — this restriction may be lifted in the future).

7.5 Function definitions

A function definition defines a user-defined function object (see "The standard type hierarchy" on page 12)[1]:

```
funcdef:        "def" funcname "(" [parameter_list] ")" ":" suite
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 name space to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global name space as the global name space 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.

When one or more top-level parameters have the form *parameter = expression*, the function is said to have "default parameter values". Default parameter values are evaluated when the function definition is executed. For a parameter with a default value, the correponding 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.[2]

Function call semantics are described in section "Calls" on page 31. When a user-defined function is called, first missing arguments for which a default value exists are supplied; then the arguments (a.k.a. actual parameters) are bound to the (formal) parameters, as follows:

- If there are no formal parameters, there must be no arguments.
- If the formal parameter list does not end in a star followed by an identifier, there must be exactly as many arguments as there are parameters in the formal parameter list (at the top level); the arguments are assigned to the formal parameters one by one. Note that the presence or absence of a trailing comma at the top level in either the formal or the actual parameter list makes no difference. The assignment to a formal parameter is performed as if the parameter occurs on the left hand side of an assignment statement whose right hand side's value is that of the argument.
- If the formal parameter list ends in a star followed by an identifier, preceded by zero or more comma-followed parameters, there must be at least as many arguments as there are parameters preceding the star.

1. The new syntax to receive arbitrary keyword arguments is not yet documented in this manual. See chapter 12 of the Tutorial.
2. Currently this is not checked; instead, `def f(a=1,b)` is interpreted as `def f(a=1,b=None)`.

Call this number *N*. The first *N* arguments are assigned to the corresponding formal parameters in the way descibed above. A tuple containing the remaining arguments, if any, is then assigned to the identifier following the star. This variable will always be a tuple: if there are no extra arguments, its value is `()`, if there is just one extra argument, it is a singleton tuple.

Note that the 'variable length parameter list' feature only works at the top level of the parameter list; individual parameters use a model corresponding more closely to that of ordinary assignment. While the latter model is generally preferable, because of the greater type safety it offers (wrong-sized tuples aren't silently mistreated), variable length parameter lists are a sufficiently accepted practice in most programming languages that a compromise has been worked out. (And anyway, assignment has no equivalent for empty argument lists.)

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 "Boolean operations" on page 35.

7.6 Class definitions

A class definition defines a class object (see section "The standard type hierarchy" on page 12):

```
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. The class's suite is then executed in a new execution frame (see section "Code blocks, execution frames, and name spaces" on page 23), using a newly created local name space and the original global name space. (Usually, the suite contains only function definitions.) When the class's suite finishes execution, its execution frame is discarded but its local name space is saved. A class object is then created using the inheritance list for the base classes and the saved local name space for the attribute dictionary. The class name is bound to this class object in the original local name space.

CHAPTER 8: 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 name space 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 name space 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.

INDEX

I

K

L

M