

---

# Python Reference Manual

*Release 2.2*

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

December 21, 2001

**PythonLabs**  
Email: [python-docs@python.org](mailto:python-docs@python.org)

Copyright © 2001 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

See the end of this document for complete license and permissions information.

## Abstract

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

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



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Notation . . . . .	1
<b>2</b>	<b>Lexical analysis</b>	<b>3</b>
2.1	Line structure . . . . .	3
2.2	Other tokens . . . . .	5
2.3	Identifiers and keywords . . . . .	6
2.4	Literals . . . . .	6
2.5	Operators . . . . .	9
2.6	Delimiters . . . . .	9
<b>3</b>	<b>Data model</b>	<b>11</b>
3.1	Objects, values and types . . . . .	11
3.2	The standard type hierarchy . . . . .	12
3.3	Special method names . . . . .	17
<b>4</b>	<b>Execution model</b>	<b>25</b>
4.1	Code blocks, execution frames, and namespaces . . . . .	25
4.2	Exceptions . . . . .	26
<b>5</b>	<b>Expressions</b>	<b>29</b>
5.1	Arithmetic conversions . . . . .	29
5.2	Atoms . . . . .	29
5.3	Primaries . . . . .	31
5.4	The power operator . . . . .	34
5.5	Unary arithmetic operations . . . . .	35
5.6	Binary arithmetic operations . . . . .	35
5.7	Shifting operations . . . . .	35
5.8	Binary bit-wise operations . . . . .	36
5.9	Comparisons . . . . .	36
5.10	Boolean operations . . . . .	37
5.11	Expression lists . . . . .	38
5.12	Summary . . . . .	38
<b>6</b>	<b>Simple statements</b>	<b>41</b>
6.1	Expression statements . . . . .	41
6.2	Assert statements . . . . .	41
6.3	Assignment statements . . . . .	42
6.4	The <code>pass</code> statement . . . . .	44
6.5	The <code>del</code> statement . . . . .	44
6.6	The <code>print</code> statement . . . . .	44
6.7	The <code>return</code> statement . . . . .	44
6.8	The <code>yield</code> statement . . . . .	45
6.9	The <code>raise</code> statement . . . . .	45

6.10	The <code>break</code> statement	46
6.11	The <code>continue</code> statement	46
6.12	The <code>import</code> statement	46
6.13	The <code>global</code> statement	47
6.14	The <code>exec</code> statement	47
<b>7</b>	<b>Compound statements</b>	<b>49</b>
7.1	The <code>if</code> statement	49
7.2	The <code>while</code> statement	50
7.3	The <code>for</code> statement	50
7.4	The <code>try</code> statement	51
7.5	Function definitions	52
7.6	Class definitions	52
<b>8</b>	<b>Top-level components</b>	<b>55</b>
8.1	Complete Python programs	55
8.2	File input	55
8.3	Interactive input	55
8.4	Expression input	56
<b>A</b>	<b>Future statements and nested scopes</b>	<b>57</b>
A.1	Future statements	57
A.2	<code>__future__</code> — Future statement definitions	58
A.3	Nested scopes	59
<b>B</b>	<b>History and License</b>	<b>61</b>
B.1	History of the software	61
B.2	Terms and conditions for accessing or otherwise using Python	61
	<b>Index</b>	<b>65</b>

# Introduction

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

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

It is dangerous to add too many implementation details to a language reference document — the implementation may change, and other implementations of the same language may work differently. On the other hand, there is currently only one Python implementation in widespread use (although a second one now exists!), 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 defined in lexical and grammar rules in this document.)

Each rule begins with a name (which is the name defined by the rule) and a 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 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.

# 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.

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

## 2.1 Line structure

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

### 2.1.1 Logical lines

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

### 2.1.2 Physical lines

A physical line 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. Implicitly 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). During interactive input of statements, handling of a blank line may differ depending on the implementation of the read-eval-print loop. In the standard implementation, an entirely blank logical line (i.e. one containing not even whitespace or a comment) terminates a multi-line statement.

### 2.1.7 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 and including the replacement is a multiple of eight (this is intended to be the same rule as used by UNIX). The total number of spaces preceding the first non-blank character then determines the line's indentation. Indentation cannot be split over multiple physical lines using backslashes; the whitespace up to the first backslash determines the indentation.

**Cross-platform compatibility note:** because of the nature of text editors on non-UNIX platforms, it is unwise to use a mixture of spaces and tabs for the indentation in a single source file.

A formfeed character may be present at the start of the line; it will be ignored for the indentation

calculations above. Formfeed characters occurring elsewhere in the leading whitespace have an undefined effect (for instance, they may reset the space count to zero).

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

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

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

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

The following example shows various indentation errors:

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

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

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

### 2.3.2 Reserved classes of identifiers

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

Form	Meaning	Notes
<code>_*</code>	Not imported by 'from <i>module</i> import *'	(1)
<code>__*__</code>	System-defined name	
<code>__*</code>	Class-private name mangling	

(XXX need section references here.)

Note:

- (1) The special identifier `'_'` is used in the interactive interpreter to store the result of the last evaluation; it is stored in the `__builtin__` module. When not in interactive mode, `'_'` has no special meaning and is not defined.

## 2.4 Literals

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

### 2.4.1 String literals

String literals are described by the following lexical definitions:

```

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

```

One syntactic restriction not indicated by these productions is that whitespace is not allowed between the `stringprefix` and the rest of the string literal.

In plain English: String literals can be enclosed in matching single quotes (') or double quotes ("). They can also be enclosed in matching groups of three single or double quotes (these are generally referred to as *triple-quoted strings*). The backslash (\) character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character. String literals may optionally be prefixed with a letter 'r' or 'R'; such strings are called *raw strings* and use different rules for interpreting backslash escape sequences. A prefix of 'u' or 'U' makes the string a Unicode string. Unicode strings use the Unicode character set as defined by the Unicode Consortium and ISO 10646. Some additional escape sequences, described below, are available in Unicode strings. The two prefix characters may be combined; in this case, 'u' must appear before 'r'.

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

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

Escape Sequence	Meaning
<code>\newline</code>	Ignored
<code>\\</code>	Backslash (\)
<code>\'</code>	Single quote (')
<code>\"</code>	Double quote (")
<code>\a</code>	ASCII Bell (BEL)
<code>\b</code>	ASCII Backspace (BS)
<code>\f</code>	ASCII Formfeed (FF)
<code>\n</code>	ASCII Linefeed (LF)
<code>\N{name}</code>	Character named <i>name</i> in the Unicode database (Unicode only)
<code>\r</code>	ASCII Carriage Return (CR)
<code>\t</code>	ASCII Horizontal Tab (TAB)
<code>\uxxxx</code>	Character with 16-bit hex value <i>xxxx</i> (Unicode only)
<code>\Uxxxxxxxx</code>	Character with 32-bit hex value <i>xxxxxxxx</i> (Unicode only)
<code>\v</code>	ASCII Vertical Tab (VT)
<code>\ooo</code>	ASCII character with octal value <i>ooo</i>
<code>\xhh</code>	ASCII character with hex value <i>hh</i>

As in Standard C, up to three octal digits are accepted. However, exactly two hex digits are taken in hex escapes.

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

When an 'r' or 'R' prefix is present, a character following a backslash is included in the string without change, and *all backslashes are left in the string*. For example, the string literal `r"\n"` consists of two characters: a backslash and a lowercase 'n'. String quotes can be escaped with a backslash, but the backslash remains in the string; for example, `r"\"` is a valid string literal consisting of two characters:

a backslash and a double quote; `r"\` is not a valid string literal (even a raw string cannot end in an odd number of backslashes). Specifically, *a raw string cannot end in a single backslash* (since the backslash would escape the following quote character). Note also that a single backslash followed by a newline is interpreted as those two characters as part of the string, *not* as a line continuation.

## 2.4.2 String literal concatenation

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

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

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

## 2.4.3 Numeric literals

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

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

## 2.4.4 Integer and long integer literals

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

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

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

Plain integer 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
```

## 2.4.5 Floating point literals

Floating point literals are described by the following lexical definitions:

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

Note that the integer and exponent parts of floating point numbers can look like octal integers, but are interpreted using radix 10. For example, '077e010' is legal, and denotes the same number as '77e10'. The allowed range of floating point literals is implementation-dependent. Some examples of floating point literals:

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

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

## 2.4.6 Imaginary literals

Imaginary literals are described by the following lexical definitions:

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

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

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

## 2.5 Operators

The following tokens are operators:

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

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

## 2.6 Delimiters

The following tokens serve as delimiters in the grammar:

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

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

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

’        “        #        \

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

@        \$        ?

## Data model

### 3.1 Objects, values and types

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

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

Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected. An implementation is allowed to postpone garbage collection or omit it altogether — it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable. (Implementation note: the current implementation uses a reference-counting scheme with (optional) delayed detection of cyclicly linked garbage, which collects most objects as soon as they become unreachable, but is not guaranteed to collect garbage containing circular references. See the [Python Library Reference](#) for information on controlling the collection of cyclic garbage.)

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

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

Some objects contain references to other objects; these are called *containers*. Examples of containers are tuples, lists and dictionaries. The references are part of a container’s value. In most cases, when we talk about the value of a container, we imply the values, not the identities of the contained objects; however, when we talk about the mutability 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.

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

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

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

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

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

**Integers** These represent elements from the mathematical set of 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 would fall 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 processor and memory usage that are usually the reason for

using these is dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating point numbers.

**Complex numbers** These represent complex numbers as a pair of machine-level double precision floating point numbers. The same caveats apply as for floating point numbers. The real and imaginary 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 non-negative numbers. The built-in function `len()` returns the number of items of a sequence. When the length of a sequence is  $n$ , the index set contains the numbers  $0, 1, \dots, n-1$ . Item  $i$  of sequence  $a$  is selected by `a[i]`.

Sequences also support slicing: `a[i:j]` selects all items with index  $k$  such that  $i \leq k < j$ . 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.

Sequences are distinguished according to their mutability:

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

The following types are immutable sequences:

**Strings** The items of a string are characters. There is no separate character type; a character is represented by a string of one item. Characters represent (at least) 8-bit bytes. The built-in functions `chr()` and `ord()` convert between characters and nonnegative integers representing the byte values. Bytes with the values 0-127 usually represent the corresponding ASCII values, but the interpretation of values is up to the program. The string data type is also used to represent arrays of bytes, e.g., to hold data read from a file.

(On systems whose native character set is not ASCII, strings may use EBCDIC in their internal representation, provided the functions `chr()` and `ord()` implement a mapping between ASCII and EBCDIC, and string comparison preserves the ASCII order. Or perhaps someone can propose a better rule?)

**Unicode** The items of a Unicode object are Unicode characters. A Unicode character is represented by a Unicode object of one item and can hold a 16-bit value representing a Unicode ordinal. The built-in functions `unichr()` and `ord()` convert between characters and nonnegative integers representing the Unicode ordinals as defined in the Unicode Standard 3.0. Conversion from and to other encodings are possible through the Unicode method `encode` and the built-in function `unicode()`.

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

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

There is currently a single mutable sequence type:

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

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

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

There is currently a single intrinsic mapping type:

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

Dictionaries are mutable; they are created by the `{...}` notation (see section 5.2.5, “Dictionary Displays”).

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

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

**User-defined functions** A user-defined function object is created by a function definition (see section 7.5, “Function definitions”). It should be called with an argument list containing the same number of items as the function's formal parameter list.

Special attributes: `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 namespace of the module in which the function was defined; `func_dict` or `__dict__` contains the namespace supporting arbitrary function attributes; `func_closure` is `None` or a tuple of cells that contain binding for the function's free variables. Of these, `func_code`, `func_defaults`, `func_closure`, `func_doc`/`__doc__`, and `func_dict`/`__dict__` may be writable; the others can never be changed. Additional information about a function's definition can be retrieved from its code object; see the description of internal types below.

In Python 2.1, the `func_closure` slot is always `None` unless nested scopes are enabled. (See the appendix.)

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

Special read-only attributes: `im_self` is the class instance object, `im_func` is the function object; `im_class` is the class of `im_self` for bound methods, or the class that asked for the method for unbound methods); `__doc__` is the method's documentation (same as `im_func.__doc__`); `__name__` is the method name (same as `im_func.__name__`). Changed in version 2.2: `im_self` used to refer to the class that defined the method.

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

User-defined method objects are created in two ways: when getting an attribute of a class that is a user-defined function object, or when getting an attribute of a class instance that is a user-defined function object defined by the class of the instance. 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 has a method `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_func` 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_func` 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. It is also important to note that user-defined functions which are attributes of a class instance are not converted to bound methods; this *only* happens when the function is an attribute of the class.

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

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

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

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

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

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

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

**Classes** Class objects are created by class definitions (see section 7.6, “Class definitions”). A class has a namespace implemented by a dictionary object. Class attribute references are translated to lookups in this dictionary, e.g., `'C.x'` is translated to `'C.__dict__["x"]'`. When the attribute name is not found there, the attribute search continues in the base classes. The search is depth-first, left-to-right in the order of occurrence in the base class list. When a class attribute reference 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 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 (see above) to yield a class instance (see below).

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

**Class instances** A class instance is created by calling a class object (see above). A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes. If a class attribute is found that is a user-defined function object (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 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, or mappings if they have methods with certain special names. See section 3.3, "Special method names."

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

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

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

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

Special read-only attributes: `co_name` gives the function name; `co_argcount` is the number of positional arguments (including arguments with default values); `co_nlocals` is the number of local variables used by the function (including arguments); `co_varnames` is a tuple containing the names of the local variables (starting with the argument names); `co_cellvars` is a tuple containing the names of local variables that are referenced by nested functions; `co_freevars` is a tuple containing the names of local variables that are neither local nor global; `co_code` is a string representing the sequence of bytecode instructions; `co_consts` is a tuple containing the literals used by the bytecode; `co_names` is a tuple containing the names used by the bytecode; `co_filename` is the filename from which the code was compiled; `co_firstlineno` is the first line number of the function; `co_lnotab` is a string encoding the mapping from byte code offsets to line numbers (for details see the source code of the interpreter); `co_stacksize` is the required stack size (including local variables); `co_flags` is an integer encoding a number of flags for the interpreter.

The `co_cellvars` and `co_freevars` are present in Python 2.1 when nested scopes are not enabled, but the code itself does not use or create cells.

The following flag bits are defined for `co_flags`: bit 0x04 is set if the function uses the `*arguments` syntax to accept an arbitrary number of positional arguments; bit 0x08 is set if the function uses the `**keywords` syntax to accept arbitrary keyword arguments; other bits are used internally or reserved for future use; bit 0x10 is set if the function was compiled with nested scopes enabled. If a code object represents a function, the first item in `co_consts` is the documentation string of the function, or None if undefined.

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

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

Special writable attributes: `f_trace`, if not `None`, is a function called at the start of each source code line (this is used by the debugger); `f_exc_type`, `f_exc_value`, `f_exc_traceback` represent the most recent exception caught in this frame.

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

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

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

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

## 3.3 Special method names

A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. 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 — 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.

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

### 3.3.1 Basic customization

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

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

### `__del__(self)`

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

**Note:** `del x` doesn't directly call `x.__del__()` — the former decrements the reference count for `x` by one, and the latter is only called when 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`. Circular references which are garbage are detected when the option cycle detector is enabled (it's on by default), but can only be cleaned up if there are no Python-level `__del__()` methods involved. Refer to the documentation for the [gc module](#) for more information about how `__del__()` methods are handled by the cycle detector, particularly the description of the `garbage` value.

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

### `__repr__(self)`

Called by the `repr()` built-in function and by string conversions (reverse quotes) to compute the “official” string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value (given an appropriate environment). If this is not possible, a string of the form `<...some useful description...>` should be returned. The return value must be a string object.

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

### `__str__(self)`

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

### `__lt__(self, other)`

### `__le__(self, other)`

### `__eq__(self, other)`

### `__ne__(self, other)`

### `__gt__(self, other)`

### `__ge__(self, other)`

New in version 2.1. These are the so-called “rich comparison” methods, and are called for comparison operators in preference to `__cmp__()` below. The correspondence between operator symbols and method names is as follows: `x < y` calls `x.__lt__(y)`, `x <= y` calls `x.__le__(y)`, `x == y` calls `x.__eq__(y)`, `x != y` and `x > y` call `x.__ne__(y)`, `x > y` calls `x.__gt__(y)`, and `x >= y` calls `x.__ge__(y)`. These methods can return any value, but if the comparison operator is used in a Boolean context, the return value should be interpretable as a Boolean value, else a `TypeError`

will be raised. By convention, 0 is used for false and 1 for true.

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

Arguments to rich comparison methods are never coerced. A rich comparison method may return `NotImplemented` if it does not implement the operation for a given pair of arguments.

`__cmp__(self, other)`

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

`__rcmp__(self, other)`

Changed in version 2.1: No longer supported.

`__hash__(self)`

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

`__nonzero__(self)`

Called to 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 intentional asymmetry between `__getattr__()` and `__setattr__()`.) This is done both for efficiency reasons and because otherwise `__setattr__()` would have no way to access other attributes of the instance. Note that at least for instance variables, you can fake total control by not inserting any values in the instance attribute dictionary (but instead inserting them in another object).

`__setattr__(self, name, value)`

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

If `__setattr__()` wants to assign to an instance attribute, it should not simply execute

`'self.name = value'` — this would cause a recursive call to itself. Instead, it should insert the value in the dictionary of instance attributes, e.g., `'self.__dict__[name] = value'`.

`__delattr__(self, name)`

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

### 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 container types

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

`__len__(self)`

Called to implement the built-in function `len()`. Should return the length of the object, an integer  $\geq 0$ . Also, an object that doesn’t define a `__nonzero__()` method and whose `__len__()` method returns zero is considered to be false in a Boolean context.

`__getitem__(self, key)`

Called to implement evaluation of `self[key]`. For sequence types, the accepted keys should be integers and slice objects. Note that the special interpretation of negative indexes (if the class wishes to emulate a sequence type) is up to the `__getitem__()` method. If `key` is of an inappropriate type, `TypeError` may be raised; if of a value outside the set of indexes for the sequence (after any special interpretation of negative values), `IndexError` should be raised. **Note:** for loops expect that an `IndexError` will be raised for illegal indexes to allow proper detection of the end of the sequence.

`__setitem__(self, key, value)`

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

`__delitem__(self, key)`

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

`__iter__(self)`

This method is called when an iterator is required for a container. This method should return a

new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container, and should also be made available as the method `iterkeys()`.

Iterator objects also need to implement this method; they are required to return themselves. For more information on iterator objects, see “[Iterator Types](#)” in the *Python Library Reference*.

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

`__contains__(self, item)`

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

### 3.3.5 Additional methods for emulation of sequence types

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

`__getslice__(self, i, j)`

**Deprecated since release 2.0.** Support slice objects as parameters to the `__getitem__()` method.

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

`__setslice__(self, i, j, sequence)`

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

This method is deprecated. If no `__setslice__()` is found, a slice object is created instead, and passed to `__setitem__()` instead.

`__delslice__(self, i, j)`

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

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

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

```

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

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

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

```

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

### 3.3.6 Emulating numeric types

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

```

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

```

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

```

__div__(self, other)
__truediv__(self, other)

```

The division operator (`/`) is implemented by these methods. The `__truediv__()` method is used

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

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

These methods are called to implement the binary arithmetic operations (+, -, \*, /, %, `divmod()`, `pow()`, \*\*, <<, >>, &, ^, |) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation. For instance, to evaluate the expression `x-y`, where `y` is an instance of a class that has an `__rsub__()` method, `y.__rsub__(x)` is called. Note that ternary `pow()` will not try calling `__rpow__()` (the coercion rules would become too complicated).

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

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

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

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

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

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

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

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

```
__coerce__(self, other)
```

Called to implement “mixed-mode” numeric arithmetic. Should either return a 2-tuple containing

*self* and *other* converted to a common numeric type, or `None` if conversion is impossible. When the common type would be the type of `other`, it is sufficient to return `None`, since the interpreter will also ask the other object to attempt 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  $x \text{ op } y$ , 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 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.

---

## Execution model

### 4.1 Code blocks, execution frames, and namespaces

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 file read by the built-in function `execfile()` is a code block. The string argument passed to the built-in function `eval()` and to the `exec` statement is a code block. 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 namespaces, the local and the global namespace, that affect execution of the code block.

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

The *local namespace* of an execution frame determines the default place where names are defined and searched. The *global namespace* 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 selected 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 namespace; global names are searched only in the global and built-in namespace.<sup>1</sup>

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

When a global name is not found in the global namespace, it is searched in the built-in namespace (which is actually the global namespace of the module `__builtin__`). The built-in namespace associated with

---

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

the execution of a code block is actually found by looking up the name `__builtins__` in its global namespace; this should be a dictionary or a module (in the latter case 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. When a name is not found at all, a `NameError` exception is raised.

The following table lists the meaning of the local and global namespace for various types of code blocks. The namespace for a particular module is automatically created when the module is first imported (i.e., when it is loaded). Note that in almost all cases, the global namespace is the namespace of the containing module — scopes in Python do not nest!

Code block type	Global namespace	Local namespace	Notes
Module	n.s. for this module	same as global	
Script (file or command)	n.s. for <code>__main__</code>	same as global	(1)
Interactive command	n.s. for <code>__main__</code>	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.	(2)
String passed to <code>exec</code> statement	global n.s. of containing block	local n.s. of containing block	(2), (3)
String passed to <code>eval()</code>	global n.s. of caller	local n.s. of caller	(2), (3)
File read by <code>execfile()</code>	global n.s. of caller	local n.s. of caller	(2), (3)
Expression read by <code>input()</code>	global n.s. of caller	local n.s. of caller	

Notes:

**n.s.** means *namespace*

- (1) The main module for a script is always called `__main__`; “the filename don’t enter into it.”
- (2) The global and local namespace 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 namespace, respectively. The effect of modifications to this dictionary on the namespace are undefined.<sup>2</sup>

## 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 offending piece of code from the top).

<sup>2</sup>The current implementations return the dictionary actually used to implement the namespace, *except* for functions, where the optimizer may cause the local namespace to be implemented differently, and `locals()` returns a read-only dictionary.

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

Exceptions are identified by 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` statement in section 7.4 and `raise` statement in section 6.9.



---

# 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).<sup>1</sup>

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

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

## 5.2.2 Literals

Python supports string literals and various numeric literals:

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

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

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

## 5.2.3 Parenthesized forms

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

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

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

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

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

## 5.2.4 List displays

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

```
list_display ::= "[" [listmaker] "]"
listmaker   ::= expression ( list_for | ( "," expression)* [ "," ] )
list_iter   ::= list_for | list_if
list_for    ::= "for" expression_list "in" testlist [list_iter]
list_if     ::= "if" test [list_iter]
```

A list display yields a new list object. Its contents are specified by providing either a list of expressions or a list comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and placed into the list object in that order. When a list comprehension is

---

<sup>1</sup>The Python interpreter provides a useful set of predefined built-in functions. It is not recommended to reuse (hide) these names with self defined objects. See the [Python Library Reference](#) for the descriptions of built-in functions and methods.

supplied, it consists of a single expression followed by at least one `for` clause and zero or more `for` or `if` clauses. In this case, the elements of the new list are those that would be produced by considering each of the `for` or `if` clauses a block, nesting from left to right, and evaluating the expression to produce a list element each time the innermost block is reached.

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

## 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, e.g., a module, list, or an instance. This object is then asked to produce the attribute whose name is the identifier. If this attribute is not available, the exception `AttributeError` is raised. Otherwise, the type and value of the

object produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

### 5.3.2 Subscriptions

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

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

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

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

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

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

### 5.3.3 Slicings

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

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

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

The semantics for a simple slicing are as follows. The primary must evaluate to a sequence object. The lower and upper bound expressions, if present, must evaluate to plain integers; defaults are zero and the `sys.maxint`, respectively. If either bound is negative, the sequence's length is added to it. The slicing now selects all items with index  $k$  such that  $i \leq k < j$  where  $i$  and  $j$  are the specified lower and upper bounds. This may be an empty sequence. It is not an error if  $i$  or  $j$  lie outside the range of valid indexes (such items don't exist so they aren't selected).

The semantics for an extended slicing are as follows. The primary must evaluate to a mapping object, and it is indexed with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of an ellipsis slice item is the built-in `Ellipsis` object. The conversion of a proper slice is a slice object (see section 3.2) whose `start`, `stop` and `step` attributes are the values of the expressions

given as lower bound, upper bound and stride, respectively, substituting `None` for missing expressions.

### 5.3.4 Calls

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

```
call          ::= primary "(" [argument_list [","]] ")"
argument_list ::= positional_arguments [", " keyword_arguments [", " "*" expression [", "]]
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 section 7.5 for the syntax of formal parameter lists.

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

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

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

If the syntax `*expression` appears in the function call, `expression` must evaluate to a sequence. Elements from this sequence are treated as if they were additional positional arguments; if there are positional arguments  $x_1, \dots, x_N$ , and `expression` evaluates to a sequence  $y_1, \dots, y_M$ , this is equivalent to a call with  $M+N$  positional arguments  $x_1, \dots, x_N, y_1, \dots, y_M$ .

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

```

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

```

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

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

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

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

If it is—

**a user-defined function:** The code block for the function is executed, passing it the argument list. The first thing the code block will do is bind the formal parameters to the arguments; this is described in section 7.5. When the code block executes a `return` statement, this specifies the return value of the function call.

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

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

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

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

## 5.4 The power operator

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

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

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

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

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

The `/` (division) 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 similar identities hold approximately where `x/y` is replaced by `floor(x/y)` or `floor(x/y) - 1` (for floats),<sup>2</sup> or `floor((x/y).real)` (for complex).

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:

---

<sup>2</sup>If `x` is very close to an exact integer multiple of `y`, it’s possible for `floor(x/y)` to be one larger than `(x-x%y)/y` due to rounding. In such cases, Python returns the latter result, in order to preserve that `divmod(x,y)[0] * y + x % y` be very close to `x`.

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

## 5.8 Binary bit-wise operations

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

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

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

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

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

## 5.9 Comparisons

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

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

Comparisons yield 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  $a, b, c, \dots, y, z$  are expressions and  $opa, opb, \dots, opy$  are comparison operators, then  $a opa b opb c \dots y opy z$  is equivalent to  $a opa b$  and  $b opb c$  and  $\dots y opy z$ , except that each expression is evaluated at most once.

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

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

The operators `<`, `>`, `==`, `>=`, `<=`, and `!=` compare the values of two objects. The objects need not have the same type. If both are numbers, they are converted to a common type. Otherwise, objects of different types *always* compare unequal, and are ordered consistently but arbitrarily.

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

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

- Numbers are compared arithmetically.
- Strings are compared lexicographically using the numeric equivalents (the result of the built-in function `ord()`) of their characters. Unicode and 8-bit strings are fully interoperable in this behavior.
- Tuples and lists are compared lexicographically using comparison of corresponding items.
- Mappings (dictionaries) compare equal if and only if their sorted (key, value) lists compare equal.<sup>3</sup> Outcomes other than equality are resolved consistently, but are not otherwise defined.<sup>4</sup>
- 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 execution of a program.

The operators `in` and `not in` test for set membership. `x in s` evaluates to true if `x` is a member of the set `s`, and false otherwise. `x not in s` returns the negation of `x in s`. The set membership test has traditionally been bound to sequences; an object is a member of a set if the set is a sequence and contains an element equal to that object. However, it is possible for an object to support membership tests without being a sequence. In particular, dictionaries support membership testing as a nicer way of spelling `key in dict`; other mapping types may follow suit.

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

For the Unicode and string types, `x in y` is true if and only if there exists an index `i` such that `x == y[i]` is true. If `x` is not a string or Unicode object of length 1, a `TypeError` exception is raised.

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

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

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

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

## 5.10 Boolean operations

Boolean operations have the lowest priority of all Python operations:

```
expression ::= 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.

<sup>3</sup>The implementation computes this efficiently, without constructing lists or sorting.

<sup>4</sup>Earlier versions of Python used lexicographic comparison of the sorted (key, value) lists, but this was very expensive for the common case of comparing for equality. An even earlier version of Python compared dictionaries by identity only, but this caused surprises because people expected to be able to test a dictionary for emptiness by comparing it to `{}`.

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
```

See section 7.5 for the syntax of parameter lists. Note that functions created with lambda forms cannot contain statements.

**Programmer's note:** Prior to Python 2.1, a lambda form defined inside a function has no access to names defined in the function's namespace. This is because Python had only two scopes: local and global. A common work-around was to use default argument values to pass selected variables into the lambda's namespace, e.g.:

```
def make_incrementor(increment):
    return lambda x, n=increment: x+n
```

As of Python 2.1, nested scopes were introduced, and this work-around has not been necessary. Python 2.1 supports nested scopes in modules which include the statement `'from __future__ import nested_scopes'`, and more recent versions of Python enable nested scopes by default. This version works starting with Python 2.1:

```
from __future__ import nested_scopes

def make_incrementor(increment):
    return lambda x: x+increment
```

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

## 5.12 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, and exponentiation, which groups from

right to left).

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



---

## Simple statements

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

```
simple_stmt ::= expression_stmt | assert_stmt | assignment_stmt | augmented_assignment_stmt
```

### 6.1 Expression statements

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

```
expression_stmt ::= expression_list
```

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

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

### 6.2 Assert statements

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

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

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

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

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

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

These equivalences assume that `__debug__` and `AssertionError` refer to the built-in variables with those names. In the current implementation, the built-in variable `__debug__` is 1 under normal circumstances, 0 when optimization is requested (command line option `-O`). The current code generator emits no code for an assert statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be

displayed as part of the stack trace.

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

## 6.3 Assignment statements

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

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

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

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

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

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

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

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

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

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

- If the target is a target list enclosed in parentheses or in square brackets: The object must be a sequence with the same number of items as there are targets in the target list, and its items are assigned, from left to right, to the corresponding targets.
- If the target is an attribute reference: The primary expression in the reference is evaluated. It should yield an object with assignable attributes; if this is not the case, `TypeError` is raised. That object is then asked to assign the assigned object to the given attribute; if it cannot perform the assignment, it raises an exception (usually but not necessarily `AttributeError`).
- If the target is a subscription: The primary expression in the reference is evaluated. It should yield either a mutable sequence object (e.g., a list) or a mapping object (e.g., a dictionary). Next, the subscript expression is evaluated.

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

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

- If the target is a slicing: The primary expression in the reference is evaluated. It should yield a mutable sequence object (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 sequence's length is added to it. The resulting bounds are clipped to lie between zero and the sequence's length, inclusive. Finally, the sequence object is asked to replace the slice with the items of the assigned sequence. The length of the slice may be different from the length of the assigned sequence, thus changing the length of the target sequence, if the object allows it.

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

WARNING: Although the definition of assignment implies that overlaps between the left-hand side and the right-hand side are 'safe' (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
```

### 6.3.1 Augmented Assignment statements

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

```
augmented_assignment_stmt ::= target augop expression_list
augop                      ::= "+=" | "-=" | "*=" | "/=" | "%=" | "**=" | ">>=" | "<<=" | "&=" |
target                    ::= identifier | "(" target_list ")" | "[" target_list "]" | attri
```

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

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

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

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

## 6.4 The pass statement

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

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

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

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

## 6.5 The del statement

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

Deletion is recursively defined very similar to the way assignment is defined. Rather than 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 namespace, 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).

## 6.6 The print statement

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

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

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

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

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

## 6.7 The return statement

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

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

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

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

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

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

## 6.8 The `yield` statement

```
yield_stmt ::= "yield" expression_list
```

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

When a generator function is called, it returns an iterator known as a generator iterator, or more commonly, a generator. The body of the generator function is executed by calling the generator's `next()` method repeatedly until it raises an exception.

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

The `yield` statement is not allowed in the `try` clause of a `try ... finally` construct. The difficulty is that there's no guarantee the generator will ever be resumed, hence no guarantee that the `finally` block will ever get executed.

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

```
from __future__ import generators
```

**See Also:**

PEP 0255, “*Simple Generators*”

The proposal for adding generators and the `yield` statement to Python.

## 6.9 The `raise` statement

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

If no expressions are present, `raise` re-raises the last expression that was raised in the current scope.

Otherwise, `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 may be an instance of that class or one of its derivatives, and then that instance is raised. If the second expression is not such an instance, the given class is instantiated. The argument list for the instantiation is determined as follows: if the second expression is a tuple, it is used as the argument list; if it is `None`, the argument list is empty; otherwise, the argument list consists of a single argument which is the second expression. If the first expression is an instance object, the second expression must be `None`.

If the first object is a 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 a class or instance, it raises the exception identified by the class of the instance determined in the previous step, with the instance as its parameter.

If a third object is present, and it is not `None`, it should be a traceback object (see section 3.2), 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.

## 6.10 The `break` statement

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

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

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

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

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

## 6.11 The `continue` statement

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

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

## 6.12 The `import` statement

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

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

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

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

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

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

---

<sup>1</sup>It may occur within an `except` or `else` clause. The restriction on occurring in the `try` clause is implementor’s laziness and will eventually be lifted.

name following `as` is used as the local name for the module. To avoid confusion, you cannot import modules with dotted names `as` a different local name. So `import module as m` is legal, but `import module.submod as s` is not. The latter should be written as `from module import submod as s`; see below.

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

The *public names* defined by a module are determined by checking the module's namespace for a variable named `__all__`; if defined, it must be a sequence of strings which are names defined or imported by that module. The names given in `__all__` are all considered public and are required to exist. If `__all__` is not defined, the set of public names includes all names found in the module's namespace which do not begin with an underscore character (`'_'`).

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.

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

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

## 6.13 The global statement

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

The `global` statement is a declaration which holds for the entire current code block. It means that the listed identifiers are to be interpreted as globals. 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 textually preceding that `global` statement.

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

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

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

## 6.14 The exec statement

```
exec_stmt ::= "exec" 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.

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

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

Also, in the current implementation, multi-line compound statements must end with a newline: `exec "for v in seq:\n\tprint v\n"` works, but `exec "for v in seq:\n\tprint v"` fails with `SyntaxError`.

## 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 5.10 for the definition of true and false); then that suite is executed (and no other part of the `if` statement is executed or evaluated). If all expressions are false, the suite of the `else` clause, if present, is executed.

## 7.2 The while statement

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

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

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

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

## 7.3 The for statement

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

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

The expression list is evaluated once; it should yield 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" ":"
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 search inspects the `except` clauses in turn until one is found that matches the exception. An expression-less `except` clause, if present, must be last; it matches any exception. For an `except` clause with an expression, that expression is evaluated, and the clause matches the exception if the resulting object is “compatible” with the exception. An object is compatible with an exception if it is 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 canceled and a search starts for the new exception in the surrounding code and on the call stack (it is treated as if the entire `try` statement raised the exception).

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

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

The optional `else` clause is executed if and when control flows off the end of the `try` clause.<sup>1</sup> 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` or `break` statement, the saved exception is lost. A `continue` statement is illegal in the `finally` clause. (The reason is a problem with the current implementation – this restriction may be lifted in the future). The exception information is not available to the program during execution of the `finally` clause.

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

---

<sup>1</sup>Currently, control “flows off the end” except in the case of an exception or the execution of a `return`, `continue`, or `break` statement.

## 7.5 Function definitions

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

```
funcdef          ::= "def" funcname "(" [parameter_list] ")" ":" suite
parameter_list  ::= (defparameter ",")* ("*" identifier [, "**" identifier] | "*" identifier
defparameter    ::= parameter ["=" expression]
sublist         ::= parameter ("," parameter)* [","]
parameter       ::= identifier | "(" sublist ")"
funcname        ::= identifier
```

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

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

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

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

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

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

It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions. This uses lambda forms, described in section 5.10. Note that the lambda form is merely a shorthand for a simplified function definition; a function defined in a “`def`” statement can be passed around or assigned to another name just like a function defined by a lambda form. The “`def`” form is actually more powerful since it allows the execution of multiple statements.

**Programmer’s note:** a “`def`” form executed inside a function definition defines a local function that can be returned or passed around. The semantics of name resolution in the nested function will change in Python 2.2. See the appendix for a description of the new semantics.

## 7.6 Class definitions

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

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

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

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



## Top-level components

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

### 8.1 Complete Python programs

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

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

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

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

### 8.2 File input

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

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

This syntax is used in the following situations:

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

### 8.3 Interactive input

Input in interactive mode is parsed using the following grammar:

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

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

## 8.4 Expression input

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

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

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

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

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

---

## Future statements and nested scopes

The semantics of Python's static scoping will change in version 2.2 to support resolution of unbound local names in enclosing functions' namespaces. The new semantics will be available in Python 2.1 through the use of a future statement. This appendix documents these two features for Python 2.1; it will be removed in Python 2.2 and the features will be documented in the main sections of this manual.

### A.1 Future statements

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

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

feature: identifier
name: identifier
```

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

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

The features recognized by Python 2.2 are 'generators', 'division' and 'nested\_scopes'. 'nested\_scopes' is redundant in 2.2 as the nested scopes feature is active by default.

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

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

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

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

Note that there is nothing special about the statement:

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

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

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

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

## A.2 `__future__` — Future statement definitions

`__future__` is a real module, and serves three purposes:

- To avoid confusing existing tools that analyze import statements and expect to find the modules they're importing.
- To ensure that future statements run under releases prior to 2.1 at least yield runtime exceptions (the import of `__future__` will fail, because there was no module of that name prior to 2.1).
- To document when incompatible changes were introduced, and when they will be — or were — made mandatory. This is a form of executable documentation, and can be inspected programatically via importing `__future__` and examining its contents.

Each statement in '`__future__.py`' is of the form:

```
FeatureName = "_Feature(" OptionalRelease "," MandatoryRelease ","  
                CompilerFlag ")"
```

where, normally, `OptionalRelease` is less than `MandatoryRelease`, and both are 5-tuples of the same form as `sys.version_info`:

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int  
 PY_MINOR_VERSION, # the 1; an int  
 PY_MICRO_VERSION, # the 0; an int  
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string  
 PY_RELEASE_SERIAL # the 3; an int  
)
```

`OptionalRelease` records the first release in which the feature was accepted.

In the case of `MandatoryReleases` that have not yet occurred, `MandatoryRelease` predicts the release in which the feature will become part of the language.

Else `MandatoryRelease` records when the feature became part of the language; in releases at or after that, modules no longer need a future statement to use the feature in question, but may continue to use such imports.

`MandatoryRelease` may also be `None`, meaning that a planned feature got dropped.

Instances of class `_Feature` have two corresponding methods, `getOptionalRelease()` and `getMandatoryRelease()`.

`CompilerFlag` is the (bitfield) flag that should be passed in the fourth argument to the builtin function `compile()` to enable the feature in dynamically compiled code. This flag is stored in the `compiler_flag` attribute on `_Future` instances.

No feature description will ever be deleted from `__future__`.

## A.3 Nested scopes

This section defines the new scoping semantics that will be introduced in Python 2.2. They are available in Python 2.1 by using the future statement `'nested_scopes'`. This section begins with a bit of terminology.

### A.3.1 Definitions and rules

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

A *block* is a piece of Python program text that is executed as a unit. The following are blocks: a module, a function body, and a class definition.

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

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

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

The name binding operations are assignment, class and function definition, import statements, for statements, and except statements. Each assignment or import statement occurs within a block defined by a class or function definition or at the module level (the top-level code block).

If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block. This can lead to errors when a name is used within a block before it is bound.

The previous rule is a subtle. Python lacks declarations and allows name binding operations to occur anywhere within a code block. The local variables of a code block can be determined by scanning the entire text of the block for name binding operations.

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

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

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

### A.3.2 Interaction with dynamic features

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

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

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

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

The builtin functions `eval()` and `input()` can not access free variables unless the variables are also referenced by the program text of the block that contains the call to `eval()` or `input()`.

*Compatibility note:* The compiler for Python 2.1 will issue warnings for uses of nested functions that will behave differently with nested scopes. The warnings will not be issued if nested scopes are enabled via a future statement. If a name bound in a function scope and the function contains a nested function scope that uses the name, the compiler will issue a warning. The name resolution rules will result in different bindings under Python 2.1 than under Python 2.2. The warning indicates that the program may not run correctly with all versions of Python.

# History and License

## B.1 History of the software

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

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

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

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

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes

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

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

## B.2 Terms and conditions for accessing or otherwise using Python

### PSF LICENSE AGREEMENT FOR PYTHON 2.2

1. This LICENSE AGREEMENT is between the Python Software Foundation (“PSF”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 2.2 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.2 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001 Python Software Foundation; All Rights Reserved" are retained in Python 2.2 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.2 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.2.
4. PSF is making Python 2.2 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.2 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.2 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.2, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.2, Licensee agrees to be bound by the terms and conditions of this License Agreement.

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

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the “BeOpen Python” logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

#### **CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1**

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

8. By clicking on the “ACCEPT” button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

**CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2**

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

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

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

# INDEX

## Symbols

- `__abs__()` (numeric object method), 23
- `__add__()` (numeric object method), 22
- `__add__()` (sequence object method), 20
- `__and__()` (numeric object method), 22
- `__bases__` (class attribute), 16
- `__builtin__` (built-in module), 25, 48, 55
- `__builtins__`, 48
- `__call__()` (object method), 20
- `__call__()` (object method), 34
- `__class__` (instance attribute), 16
- `__cmp__()` (object method), 19
- `__cmp__()` (object method), 19
- `__coerce__()` (numeric object method), 23
- `__coerce__()` (numeric object method), 20
- `__complex__()` (numeric object method), 23
- `__contains__()` (container object method), 21
- `__contains__()` (mapping object method), 20
- `__contains__()` (sequence object method), 20
- `__debug__`, 41
- `__del__()` (object method), 18
- `__delattr__()` (object method), 20
- `__delitem__()` (container object method), 20
- `__delslice__()` (sequence object method), 21
- `__dict__` (class attribute), 16
- `__dict__` (function attribute), 14
- `__dict__` (instance attribute), 16, 20
- `__dict__` (module attribute), 15
- `__div__()` (numeric object method), 22
- `__divmod__()` (numeric object method), 22
- `__doc__` (class attribute), 16
- `__doc__` (function attribute), 14
- `__doc__` (module attribute), 15
- `__eq__()` (object method), 18
- `__file__` (module attribute), 15
- `__float__()` (numeric object method), 23
- `__floordiv__()` (numeric object method), 22
- `__future__` (standard module), 58
- `__ge__()` (object method), 18
- `__getattr__()` (object method), 19
- `__getitem__()` (container object method), 20
- `__getitem__()` (mapping object method), 17
- `__getslice__()` (sequence object method), 21
- `__gt__()` (object method), 18
- `__hash__()` (object method), 19
- `__hex__()` (numeric object method), 23
- `__iadd__()` (numeric object method), 23
- `__iadd__()` (sequence object method), 20
- `__iand__()` (numeric object method), 23
- `__idiv__()` (numeric object method), 23
- `__ilshift__()` (numeric object method), 23
- `__imod__()` (numeric object method), 23
- `__import__()` (built-in function), 47
- `__imul__()` (numeric object method), 23
- `__imul__()` (sequence object method), 20
- `__init__()` (object method), 17
- `__init__()` (object method), 15
- `__init__.py`, 47
- `__int__()` (numeric object method), 23
- `__invert__()` (numeric object method), 23
- `__ior__()` (numeric object method), 23
- `__ipow__()` (numeric object method), 23
- `__irshift__()` (numeric object method), 23
- `__isub__()` (numeric object method), 23
- `__iter__()` (container object method), 20
- `__ixor__()` (numeric object method), 23
- `__le__()` (object method), 18
- `__len__()` (container object method), 20
- `__len__()` (mapping object method), 19
- `__long__()` (numeric object method), 23
- `__lshift__()` (numeric object method), 22
- `__lt__()` (object method), 18
- `__main__` (built-in module), 26, 55
- `__mod__()` (numeric object method), 22
- `__module__` (class attribute), 16
- `__mul__()` (numeric object method), 22
- `__mul__()` (sequence object method), 20
- `__name__` (class attribute), 16
- `__name__` (function attribute), 14
- `__name__` (module attribute), 15
- `__ne__()` (object method), 18
- `__neg__()` (numeric object method), 23
- `__nonzero__()` (object method), 19
- `__nonzero__()` (object method), 20
- `__oct__()` (numeric object method), 23
- `__or__()` (numeric object method), 22
- `__pos__()` (numeric object method), 23
- `__pow__()` (numeric object method), 22
- `__radd__()` (numeric object method), 23
- `__radd__()` (sequence object method), 20
- `__rand__()` (numeric object method), 23
- `__rcmp__()` (object method), 19

- `__rdiv__()` (numeric object method), 23
- `__rdivmod__()` (numeric object method), 23
- `__repr__()` (object method), 18
- `__rlshift__()` (numeric object method), 23
- `__rmod__()` (numeric object method), 23
- `__rmul__()` (numeric object method), 23
- `__rmul__()` (sequence object method), 20
- `__ror__()` (numeric object method), 23
- `__rpow__()` (numeric object method), 23
- `__rrshift__()` (numeric object method), 23
- `__rshift__()` (numeric object method), 22
- `__rsub__()` (numeric object method), 23
- `__rxor__()` (numeric object method), 23
- `__setattr__()` (object method), 19
- `__setattr__()` (object method), 19
- `__setitem__()` (container object method), 20
- `__setslice__()` (sequence object method), 21
- `__str__()` (object method), 18
- `__sub__()` (numeric object method), 22
- `__truediv__()` (numeric object method), 22
- `__xor__()` (numeric object method), 22

## A

- `abs()` (built-in function), 23
- addition, 35
- and
  - bit-wise, 36
- and
  - operator, 37
- anonymous
  - function, 38
- `append()` (sequence object method), 20
- argument
  - function, 14
- arithmetic
  - conversion, 29
  - operation, binary, 35
  - operation, unary, 35
- `array` (standard module), 13
- ASCII, 2, 6, 7, 10, 13
- `assert`
  - statement, 41
- `AssertionError`
  - exception, 41
- assertions
  - debugging, 41
- assignment
  - attribute, 42
  - augmented, 43
  - class attribute, 15
  - class instance attribute, 16
  - slicing, 43
  - statement, 13, 42
  - subscription, 42
  - target list, 42
- atom, 29
- attribute, 12
  - assignment, 42

- assignment, class, 15
- assignment, class instance, 16
- class, 15
- class instance, 16
- deletion, 44
- generic special, 12
- reference, 31
- special, 12

## AttributeError

- exception, 31
- augmented
  - assignment, 43

## B

- back-quotes, 18, 31
- backslash character, 4
- backward
  - quotes, 18, 31
- binary
  - arithmetic operation, 35
  - bit-wise operation, 36
- binding
  - global name, 47
  - name, 25, 30, 42, 46, 47, 52, 53
- bit-wise
  - and, 36
  - operation, binary, 36
  - operation, unary, 35
  - or, 36
  - xor, 36
- blank line, 4
- block
  - code, 25
- BNF, 1, 29
- Boolean
  - operation, 37
- `break`
  - statement, 46, 50, 51
- `bsddb` (standard module), 14
- built-in
  - method, 15
  - module, 46
  - name, 30
- built-in function
  - call, 34
  - object, 15, 34
- built-in method
  - call, 34
  - object, 15, 34
- byte, 13
- bytecode, 16

## C

- C, 7
  - language, 12, 13, 15, 36
- call, 33
  - built-in function, 34
  - built-in method, 34

- class instance, 34
- class object, 15, 34
- function, 14, 34
- instance, 20, 34
- method, 34
- procedure, 41
- user-defined function, 34
- callable
  - object, 14, 33
- chaining
  - comparisons, 36
- character, 13, 32
- character set, 13
- `chr()` (built-in function), 13
- class
  - attribute, 15
  - attribute assignment, 15
  - constructor, 17
  - definition, 45, 52
  - instance, 16
  - name, 53
  - object, 15, 34, 52
- class instance
  - attribute, 16
  - attribute assignment, 16
  - call, 34
  - object, 15, 16, 34
- class object
  - call, 15, 34
- clause, 49
- `clear()` (mapping object method), 20
- `cmp()` (built-in function), 19
- `co_argcount` (code object attribute), 16
- `co_cellvars` (code object attribute), 16
- `co_code` (code object attribute), 16
- `co_consts` (code object attribute), 16
- `co_filename` (code object attribute), 16
- `co_firstlineno` (code object attribute), 16
- `co_flags` (code object attribute), 16
- `co_freevars` (code object attribute), 16
- `co_lnotab` (code object attribute), 16
- `co_name` (code object attribute), 16
- `co_names` (code object attribute), 16
- `co_nlocals` (code object attribute), 16
- `co_stacksize` (code object attribute), 16
- `co_varnames` (code object attribute), 16
- code
  - block, 25
  - object, 16
- code block, 25, 30, 46
- comma, 30
  - trailing, 38, 44
- command line, 55
- comment, 3
- comparison, 36
  - string, 13
- comparisons, 19
  - chaining, 36
- `compile()` (built-in function), 47
- complex
  - number, 13
  - object, 13
- `complex()` (built-in function), 23
- complex literal, 8
- compound
  - statement, 49
- comprehensions
  - list, 30
- constant, 6
- constructor
  - class, 17
- container, 11, 15
- `continue`
  - statement, 46, 50, 51
- conversion
  - arithmetic, 29
  - string, 18, 31, 41
- `copy()` (mapping object method), 20
- `count()` (sequence object method), 20

## D

- dangling
  - else, 49
- data, 11
  - type, 12
  - type, immutable, 30
- datum, 31
- `dbm` (standard module), 14
- debugging
  - assertions, 41
- decimal literal, 8
- DEDENT token, 5, 49
- default
  - parameter value, 52
- definition
  - class, 45, 52
  - function, 45, 52
- `del`
  - statement, 13, 18, 44
- delete, 13
- deletion
  - attribute, 44
  - target, 44
  - target list, 44
- delimiters, 9
- destructor, 18, 42
- dictionary
  - display, 31
  - object, 14, 15, 19, 31, 32, 43
- display
  - dictionary, 31
  - list, 30
  - tuple, 30
- division, 35
- `divmod()` (built-in function), 22, 23
- documentation string, 16

## E

- EBCDIC, 13
- elif
  - keyword, 50
- Ellipsis, 12
  - object, 12
- else
  - dangling, 49
- else
  - keyword, 46, 50, 51
- empty
  - list, 31
  - tuple, 13, 30
- error handling, 26
- errors, 26
- escape sequence, 7
- eval() (built-in function), 47, 48, 56
- exc\_info (in module sys), 17
- exc\_traceback (in module sys), 17, 51
- exc\_type (in module sys), 51
- exc\_value (in module sys), 51
- except
  - keyword, 51
- exception, 26, 45
  - AssertionError, 41
  - AttributeError, 31
  - handler, 17
  - ImportError, 46, 47
  - NameError, 30
  - raising, 45
  - RuntimeError, 44
  - StopIteration, 45
  - SyntaxError, 46, 48
  - TypeError, 35
  - ValueError, 36
  - ZeroDivisionError, 35
- exception handler, 26
- exclusive
  - or, 36
- exec
  - statement, 26, 47
- execfile() (built-in function), 47
- execution
  - frame, 25, 53
  - restricted, 26
  - stack, 17
- execution model, 25
- expression, 29
  - lambda, 38
  - list, 38, 41, 42
  - statement, 41
- extended
  - slicing, 32
- extended print statement, 44
- extension
  - filename, 46
  - module, 12

## F

- f\_back (frame attribute), 17
- f\_builtins (frame attribute), 17
- f\_code (frame attribute), 17
- f\_exc\_traceback (frame attribute), 17
- f\_exc\_type (frame attribute), 17
- f\_exc\_value (frame attribute), 17
- f\_globals (frame attribute), 17
- f\_lasti (frame attribute), 17
- f\_lineno (frame attribute), 17
- f\_locals (frame attribute), 17
- f\_restricted (frame attribute), 17
- f\_trace (frame attribute), 17
- file
  - object, 16, 56
- filename
  - extension, 46
- finally
  - keyword, 45, 46, 51
- float() (built-in function), 23
- floating point
  - number, 13
  - object, 13
- floating point literal, 8
- for
  - statement, 46, 50
- form
  - lambda, 38, 52
- frame
  - execution, 25, 53
  - object, 17
- from
  - keyword, 46, 47
  - statement, 26, 47
- func\_code (function attribute), 14
- func\_defaults (function attribute), 14
- func\_dict (function attribute), 14
- func\_doc (function attribute), 14
- func\_globals (function attribute), 14
- function
  - anonymmous, 38
  - argument, 14
  - call, 14, 34
  - call, user-defined, 34
  - definition, 45, 52
  - generator, 45
  - name, 52
  - object, 14, 15, 34, 52
  - user-defined, 14
- future
  - statement, 57

## G

- garbage collection, 11
- gdbm (standard module), 14
- generator
  - function, 15, 45
  - iterator, 15, 45

- generic
  - special attribute, 12
- `get()` (mapping object method), 20
- global
  - name, 30
  - name binding, 47
  - namespace, 14, 25
- global
  - statement, 25, 26, 30, 42, 44, 47
- `globals()` (built-in function), 48
- grammar, 1
- grouping, 4

## H

- handle an exception, 26
- handler
  - exception, 17
- `has_key()` (mapping object method), 20
- `hash()` (built-in function), 19
- hash character, 3
- `hex()` (built-in function), 23
- hexadecimal literal, 8
- hierarchical
  - module names, 47
- hierarchy
  - type, 12

## I

- `id()` (built-in function), 11
- identifier, 6, 29
- identity
  - test, 37
- identity of an object, 11
- `if`
  - statement, 49
- `im_class` (method attribute), 14
- `im_func` (method attribute), 14
- `im_self` (method attribute), 14
- imaginary literal, 8
- immutable
  - data type, 30
  - object, 13, 30, 31
- immutable object, 11
- immutable sequence
  - object, 13
- `import`
  - statement, 15, 46
- `ImportError`
  - exception, 46, 47
- importing
  - module, 46
- `in`
  - keyword, 50
  - operator, 37
- inclusive
  - or, 36
- `INDENT` token, 5
- indentation, 4
- index operation, 13
- `index()` (sequence object method), 20
- inheritance, 53
- initialization
  - module, 46
- input, 56
  - raw, 56
- `input()` (built-in function), 56
- `insert()` (sequence object method), 20
- instance
  - call, 20, 34
  - class, 16
  - object, 15, 16, 34
- `int()` (built-in function), 23
- integer, 13
  - object, 12
  - representation, 12
- integer literal, 8
- interactive mode, 55
- internal type, 16
- interpreter, 55
- inversion, 35
- invocation, 14
- `is`
  - operator, 37
- `is not`
  - operator, 37
- item
  - sequence, 32
  - string, 32
- item selection, 13
- `items()` (mapping object method), 20

## K

- key, 31
- key/datum pair, 31
- `keys()` (mapping object method), 20
- keyword, 6
  - `elif`, 50
  - `else`, 46, 50, 51
  - `except`, 51
  - `finally`, 45, 46, 51
  - `from`, 46, 47
  - `in`, 50

## L

- `lambda`
  - expression, 38
  - form, 38, 52
- language
  - C, 12, 13, 15, 36
  - Pascal, 50
- `last_traceback` (in module `sys`), 17
- leading whitespace, 4
- `len()` (built-in function), 13, 20
- lexical analysis, 3
- lexical definitions, 2
- line continuation, 4

- line joining, 3, 4
- line structure, 3
- list
  - assignment, target, 42
  - comprehensions, 30
  - deletion target, 44
  - display, 30
  - empty, 31
  - expression, 38, 41, 42
  - object, 13, 31, 32, 43
  - target, 42, 50
- literal, 6, 30
- local
  - namespace, 25
- `locals()` (built-in function), 48
- logical line, 3
- `long()` (built-in function), 23
- long integer
  - object, 12
- long integer literal, 8
- loop
  - over mutable sequence, 50
  - statement, 46, 50
- loop control
  - target, 46

## M

- `makefile()` (socket method), 16
- mangling
  - name, 30
- mapping
  - object, 13, 16, 32, 43
- membership
  - test, 37
- method
  - built-in, 15
  - call, 34
  - object, 14, 15, 34
  - user-defined, 14
- minus, 35
- module
  - built-in, 46
  - extension, 12
  - importing, 46
  - initialization, 46
  - name, 46
  - names, hierarchical, 47
  - namespace, 15
  - object, 15, 32
  - search path, 46
  - user-defined, 46
- `modules` (in module `sys`), 46
- modulo, 35
- multiplication, 35
- mutable
  - object, 13, 42
- mutable object, 11
- mutable sequence

- loop over, 50
- object, 13

## N

- name, 6, 29
  - binding, 25, 30, 42, 46, 47, 52, 53
  - binding, global, 47
  - built-in, 30
  - class, 53
  - function, 52
  - global, 30
  - mangling, 30
  - module, 46
  - rebinding, 25, 42
  - unbinding, 25, 44
- `NameError`
  - exception, 30
- `NameError` (built-in exception), 26
- names
  - hierarchical module, 47
  - private, 30
- namespace, 25
  - global, 14, 25
  - local, 25
  - module, 15
- negation, 35
- nested
  - scopes, 59
- newline
  - suppression, 44
- NEWLINE token, 3, 49
- `None`, 12, 41
  - object, 12
- `not`
  - operator, 37
- `not in`
  - operator, 37
- notation, 1
- `NotImplemented`, 12
  - object, 12
- null
  - operation, 44
- number, 8
  - complex, 13
  - floating point, 13
- numeric
  - object, 12, 16
- numeric literal, 8

## O

- object, 11
  - built-in function, 15, 34
  - built-in method, 15, 34
  - callable, 14, 33
  - class, 15, 34, 52
  - class instance, 15, 16, 34
  - code, 16
  - complex, 13

- dictionary, 14, 15, 19, 31, 32, 43
- Ellipsis**, 12
- file, 16, 56
- floating point, 13
- frame, 17
- function, 14, 15, 34, 52
- immutable, 13, 30, 31
- immutable sequence, 13
- instance, 15, 16, 34
- integer, 12
- list, 13, 31, 32, 43
- long integer, 12
- mapping, 13, 16, 32, 43
- method, 14, 15, 34
- module, 15, 32
- mutable, 13, 42
- mutable sequence, 13
- None**, 12
- NotImplemented**, 12
- numeric, 12, 16
- plain integer, 12
- recursive, 31
- sequence, 13, 16, 32, 37, 43, 50
- slice, 20
- string, 13, 32
- traceback, 17, 46, 51
- tuple, 13, 32, 38
- unicode, 13
- user-defined function, 14, 34, 52
- user-defined method, 14
- oct()** (built-in function), 23
- octal literal, 8
- open()** (built-in function), 16
- operation
  - binary arithmetic, 35
  - binary bit-wise, 36
  - Boolean, 37
  - null, 44
  - shifting, 35
  - unary arithmetic, 35
  - unary bit-wise, 35
- operator
  - and**, 37
  - in**, 37
  - is**, 37
  - is not**, 37
  - not**, 37
  - not in**, 37
  - or**, 38
  - precedence, 38
- operators, 9
- or
  - bit-wise, 36
  - exclusive, 36
  - inclusive, 36
- or
  - operator, 38
- ord()** (built-in function), 13
- output, 41, 44
  - standard, 41, 44
- OverflowError** (built-in exception), 12

## P

- packages, 47
- parameter
  - value, default, 52
- parenthesized form, 30
- parser, 3
- Pascal
  - language, 50
- pass**
  - statement, 44
- path
  - module search, 46
- physical line, 3, 4, 7
- plain integer
  - object, 12
- plain integer literal, 8
- plus, 35
- pop()** (sequence object method), 20
- popen()** (in module `os`), 16
- pow()** (built-in function), 22, 23
- precedence
  - operator, 38
- primary, 31
- print**
  - statement, 18, 44
- private
  - names, 30
- procedure
  - call, 41
- program, 55
- Python Enhancement Proposals
  - PEP 0255, 45

## Q

- quotes
  - backward, 18, 31
  - reverse, 18, 31

## R

- raise**
  - statement, 45
- raise an exception, 26
- raising
  - exception, 45
- range()** (built-in function), 50
- raw input, 56
- raw string, 7
- raw\_input()** (built-in function), 56
- readline()** (file method), 56
- rebinding
  - name, 25, 42
- recursive
  - object, 31
- reference

- attribute, 31
- reference counting, 11
- `remove()` (sequence object method), 20
- `repr()` (built-in function), 18, 31, 41
- representation
  - integer, 12
- reserved word, 6
- restricted
  - execution, 26
- `return`
  - statement, 44, 51
- reverse
  - quotes, 18, 31
- `reverse()` (sequence object method), 20
- `RuntimeError`
  - exception, 44

## S

- scopes
  - nested, 59
- search
  - path, module, 46
- sequence
  - item, 32
  - object, 13, 16, 32, 37, 43, 50
- shifting
  - operation, 35
- simple
  - statement, 41
- singleton
  - tuple, 13
- slice, 32
  - object, 20
- `slice()` (built-in function), 17
- slicing, 13, 32
  - assignment, 43
  - extended, 32
- `sort()` (sequence object method), 20
- space, 4
- special
  - attribute, 12
  - attribute, generic, 12
- stack
  - execution, 17
  - trace, 17
- standard
  - output, 41, 44
- Standard C, 7
- standard input, 55
- `start` (slice object attribute), 17, 33
- statement
  - `assert`, 41
  - assignment, 13, 42
  - assignment, augmented, 43
  - `break`, 46, 50, 51
  - compound, 49
  - `continue`, 46, 50, 51
  - `del`, 13, 18, 44

- `exec`, 26, 47
- expression, 41
- `for`, 46, 50
- `from`, 26, 47
- future, 57
- global, 25, 26, 30, 42, 44, 47
- `if`, 49
- `import`, 15, 46
- loop, 46, 50
- `pass`, 44
- `print`, 18, 44
- `raise`, 45
- `return`, 44, 51
- simple, 41
- `try`, 17, 51
- `while`, 46, 50
- `yield`, 45
- statement grouping, 4
- `stderr` (in module `sys`), 16
- `stdin` (in module `sys`), 16
- `stdio`, 16
- `stdout` (in module `sys`), 16, 44
- `step` (slice object attribute), 17, 33
- `stop` (slice object attribute), 17, 33
- `StopIteration`
  - exception, 45
- `str()` (built-in function), 18, 31
- string
  - comparison, 13
  - conversion, 18, 31, 41
  - item, 32
  - object, 13, 32
  - Unicode, 7
- string literal, 6
- subscription, 13, 32
  - assignment, 42
- subtraction, 35
- suite, 49
- suppression
  - newline, 44
- syntax, 1, 29
- `SyntaxError`
  - exception, 46, 48
- `sys` (built-in module), 44, 46, 51, 55
- `sys.exc_info`, 17
- `sys.exc_traceback`, 17
- `sys.last_traceback`, 17
- `sys.modules`, 46
- `sys.stderr`, 16
- `sys.stdin`, 16
- `sys.stdout`, 16
- `SystemExit` (built-in exception), 27

## T

- tab, 4
- target, 42
  - deletion, 44
  - list, 42, 50

- list assignment, 42
- list, deletion, 44
- loop control, 46
- `tb_frame` (traceback attribute), 17
- `tb_lasti` (traceback attribute), 17
- `tb_lineno` (traceback attribute), 17
- `tb_next` (traceback attribute), 17
- termination model, 26
- test
  - identity, 37
  - membership, 37
- token, 3
- trace
  - stack, 17
- traceback
  - object, 17, 46, 51
- trailing
  - comma, 38, 44
- triple-quoted string, 7
- try
  - statement, 17, 51
- tuple
  - display, 30
  - empty, 13, 30
  - object, 13, 32, 38
  - singleton, 13
- type, 12
  - data, 12
  - hierarchy, 12
  - immutable data, 30
- `type()` (built-in function), 11
- type of an object, 11
- TypeError**
  - exception, 35
- types, internal, 16

## U

- unary
  - arithmetic operation, 35
  - bit-wise operation, 35
- unbinding
  - name, 25, 44
- `unichr()` (built-in function), 13
- Unicode, 13
- unicode
  - object, 13
- `unicode()` (built-in function), 13
- Unicode Consortium, 7
- UNIX, 55
- unreachable object, 11
- unrecognized escape sequence, 7
- `update()` (mapping object method), 20
- user-defined
  - function, 14
  - function call, 34
  - method, 14
  - module, 46
- user-defined function

- object, 14, 34, 52
- user-defined method
  - object, 14

## V

- value
  - default parameter, 52
- value of an object, 11
- ValueError**
  - exception, 36
- values
  - writing, 41, 44
- `values()` (mapping object method), 20

## W

- while
  - statement, 46, 50
- whitespace, 4
- writing
  - values, 41, 44

## X

- xor
  - bit-wise, 36

## Y

- yield
  - statement, 45

## Z

- ZeroDivisionError**
  - exception, 35