# The Python Language Reference

发布 *2.7.18*

**Guido van Rossum**
**and the Python development team**

**五月 20, 2020**

**Python Software Foundation**
**Email: docs@python.org**

# Contents

本参考手册描述了 Python 的语法和“核心语义”。本参考是简洁的，但试图做到准确和完整。非必要的内建对象类型和内建函数、模块的语义描述在 library-index 中。有关该语言的非正式介绍，请参阅 tutorial-index 。对 C 或 C++ 程序员，还有两个额外的手册：extending-index 概述了如何编写一个 Python 扩展模块，c-api-index 详细介绍了 C/C++ 中可用的接口。

CHAPTER 1

# 概述

本参考手册是对 Python 编程语言的描述。并不适宜作为教程使用。

我希望尽可能地保证内容精确无误，但还是选择使用自然词句进行描述，正式的规格定义仅用于句法和词法解析。这样应该能使文档对于普通人来说更易理解，但也可能导致一些歧义。因此，如果你是来自火星并且想凭借这份文档把 Python 重新实现一遍，也许有时需要自行猜测，实际上最终大概会得到一个十分不同的语言。而在另一方面，如果你正在使用 Python 并且想了解有关该语言特定领域的精确规则，你应该能够在这里找到它们。如果你希望查看对该语言更正式的定义，也许你可以花些时间自己写上一份—或者发明一台克隆机器:-)

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

每种 Python 实现都带有一些内置和标准的模块。相关的文档可参见 library-index 索引。少数内置模块也会在此提及，如果它们同语言描述存在明显的关联。

## 1.1 其他实现

虽然官方 Python 实现差不多得到最广泛的欢迎，但也有一些其他实现对特定领域的用户来说更具吸引力。

知名的实现包括:

**CPython** 这是最早出现并持续维护的 Python 实现，以 C 语言编写。新的语言特性通常在此率先添加。

**Jython** 以 Java 语言编写的 Python 实现。此实现可以作为 Java 应用的一个脚本语言，或者可以用来创建需要 Java 类库支持的应用。想了解更多信息可访问 Jython 网站。

**Python for .NET** 此实现实际上使用了 CPython 实现，但是属于.NET 托管应用并且可以引入.NET 类库。它的创造者是 Brian Lloyd。想了解详情可访问 Python for .NET 主页。

**IronPython** 另一个.NET 的 Python 实现，与 Python.NET 不同点在于它是生成 IL 的完全 Python 实现，并且将 Python 代码直接编译为.NET 程序集。它的创造者就是当初创造 Jython 的 Jim Hugunin。想了解详情可访问 IronPython 网站。

**PyPy** 完全使用 Python 语言编写的 Python 实现。它支持多个其他实现所没有的高级特性，例如非栈式支持和 JIT 编译器等。此项目的目标之一是通过允许方便地修改解释器 (因为它是用 Python 编写的)，鼓励该对语言本身进行试验。想了解详情可访问 PyPy 项目主页。

以上这些实现都可能在某些方面与此参考文档手册的描述有所差异，或是引入了超出标准 Python 文档范围的特定信息。请参考它们各自的专门文档，以确定你正在使用的这个实现有哪些你需要了解的东西。

## 1.2 标注

句法和词法解析的描述采用经过改进的 BNF 语法标注。这包含以下定义样式:

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

第一行表示 `name` 是一个 `lc_letter` 之后跟零个或多个 `lc_letter` 和下划线。而一个 `lc_letter` 则是任意单个 `'a'` 至 `'z'` 字符。(实际上在本文档中始终采用此规则来定义词法和语法规则的名称。)

每条规则的开头是一个名称 (即该规则所定义的名称) 加上 `::=`。竖线 (`|`) 被用来分隔可选项；它是此标注中最灵活的操作符。星号 (`*`) 表示前一项的零次或多次重复；类似地，加号 (`+`) 表示一次或多次重复，而由方括号括起的内容 (`[ ]`) 表示出现零次或一次 (或者说，这部分内容是可选的)。`*` 和 `+` 操作符的绑定是最紧密的；圆括号用于分组。固定字符串包含在引号内。空格的作用仅限于分隔形符。每条规则通常为一行；有许多个可选项的规则可能会以竖线为界分为多行。

在词法定义中 (如上述示例)，还额外使用了两个约定: 由三个点号分隔的两个字符字面值表示在指定 (闭) 区间范围内的任意单个 ASCII 字符。由尖括号 (`<...>`) 括起来的内容是对于所定义符号的非正式描述；即可以在必要时用来说明‘控制字符’的意图。

虽然所用的标注方式几乎相同，但是词法定义和句法定义是存在很大区别的: 词法定义作用于输入源中单独的字符，而句法定义则作用于由词法分析所生成的形符流。在下一章节 (“词法分析”) 中使用的 BNF 全部都是词法定义；在之后的章节中使用的则是句法定义。

CHAPTER 2

# 词法分析

Python 程序由一个 *解析器* 读取。输入到解析器的是一个由 *词法分析器* 所生成的 *形符* 流，本章将描述词法分析器是如何将一个文件拆分为一个个形符的。

Python uses the 7-bit ASCII character set for program text.

2.3 新版功能: An encoding declaration can be used to indicate that string literals and comments use an encoding different from ASCII.

For compatibility with older versions, Python only warns if it finds 8-bit characters; those warnings should be corrected by either declaring an explicit encoding, or using escape sequences if those bytes are binary data, instead of characters.

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

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

## 2.1 行结构

一个 Python 程序可分为许多 *逻辑行*。

### 2.1.1 逻辑行

逻辑行的结束是以 NEWLINE 形符表示的。语句不能跨越逻辑行的边界，除非其语法允许包含 NEWLINE (例如复合语句可由多行子语句组成)。一个逻辑行可由一个或多个 *物理行* 按照明确或隐含的 *行拼接* 规则构成。

### 2.1.2 物理行

物理行是以一个行终止序列结束的字符序列。在源文件和字符串中，可以使用任何标准平台上的行终止序列 - Unix 所用的 ASCII 字符 LF (换行), Windows 所用的 ASCII 字符序列 CR LF (回车加换行), 或者旧 Macintosh 所用的 ASCII 字符 CR (回车)。所有这些形式均可使用，无论具体平台。输入的结束也会被作为最后一个物理行的隐含终止标志。

当嵌入 Python 时，源码字符串传入 Python API 应使用标准 C 的传统换行符 (即 `\n`，表示 ASCII 字符 LF 作为行终止标志)。

### 2.1.3 注释

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 编码声明

如果一条注释位于 Python 脚本的第一或第二行，并且匹配正则表达式 `coding[=:]\s*([-\w.]+)`，这条注释会被作为编码声明来处理；上述表达式的第一组指定了源码文件的编码。编码声明必须独占一行。如果它是在第二行，则第一行也必须是注释。推荐的编码声明形式如下

```
# -*- coding: <encoding-name> -*-
```

这也是 GNU Emacs 认可的形式，以及

```
# vim:fileencoding=<encoding-name>
```

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

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

### 2.1.5 显式的行拼接

两个或更多个物理行可使用反斜杠字符 (\) 拼接为一个逻辑行，规则如下: 当一个物理行以一个不在字符串或注释内的反斜杠结尾时，它将与下一行拼接构成一个单独的逻辑行，反斜杠及其后的换行符会被删除。例如:

```
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
```

以反斜杠结束的行不能带有注释。反斜杠不能用来拼接注释。反斜杠不能用来拼接形符，字符串除外 (即原文字符串以外的形符不能用反斜杠分隔到两个物理行)。不允许有原文字符串以外的反斜杠存在于物理行的其他位置。

### 2.1.6 隐式的行拼接

圆括号、方括号或花括号以内的表达式允许分成多个物理行，无需使用反斜杠。例如:

```
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
```

隐式的行拼接可以带有注释。后续行的缩进不影响程序结构。后续行也允许为空白行。隐式拼接的行之间不会有 NEWLINE 形符。隐式拼接的行也可以出现于三引号字符串中 (见下)；此情况下这些行不允许带有注释。

### 2.1.7 空白行

A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored (i.e., no NEWLINE token is generated). During interactive input of statements, handling of a blank line may differ depending on the implementation of the read-eval-print loop. In the standard implementation, an entirely blank logical line (i.e. one containing not even whitespace or a comment) terminates a multi-line statement.

### 2.1.8 缩进

一个逻辑行开头处的空白 (空格符和制表符) 被用来计算该行的缩进等级，以决定语句段落的组织结构。

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.

**跨平台兼容性注释:** 由于非 UNIX 平台上文本编辑器本身的特性，在一个源文件中混合使用制表符和空格符是不明智的。另外也要注意不同平台还可能会显式地限制最大缩进层级。

行首有时可能会有一个进纸符；它在上述缩进层级计算中会被忽略。处于行首空格内其他位置的进纸符的效果未定义 (例如它可能导致空格计数重置为零)。

多个连续行各自的缩进层级将会被放入一个堆栈用来生成 INDENT 和 DEDENT 形符，具体说明如下。

在读取文件的第一行之前，先向堆栈推入一个零值；它将不再被弹出。被推入栈的层级数值从底至顶持续增加。每个逻辑行开头的行缩进层级将与栈顶行比较。如果相同，则不做处理。如果新行层级较高，则会被推入栈顶，并生成一个 INDENT 形符。如果新行层级较低，则 *应当* 是栈中的层级数值之一；栈中高于该层级的所有数值都将被弹出，每弹出一级数值生成一个 DEDENT 形符。在文件末尾，栈中剩余的每个大于零的数值生成一个 DEDENT 形符。

这是一个正确 (但令人迷惑) 的 Python 代码缩进示例:

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

(下页继续)

(续上页)

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

以下示例显示了各种缩进错误:

```
 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
```

(实际上，前三个错误会被解析器发现；只有最后一个错误是由词法分析器发现的—return r 的缩进无法匹配弹出栈的缩进层级。)

### 2.1.9 形符之间的空白

除非是在逻辑行的开头或字符串内，空格符、制表符和进纸符等空白符都同样可以用来分隔形符。如果两个形符彼此相连会被解析为一个不同的形符，则需要使用空白来分隔 (例如 ab 是一个形符，而 a b 是两个形符)。

## 2.2 其他形符

除了 NEWLINE, INDENT 和 DEDENT，还存在以下类别的形符: 标识符, 关键字, 字面值, 运算符以及 分隔符。空白字符 (之前讨论过的行终止符除外) 不属于形符，而是用来分隔形符。如果存在二义性，将从左至右读取尽可能长的合法字符串组成一个形符。

## 2.3 标识符和关键字

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"
```

标识符的长度没有限制。对大小写敏感。

### 2.3.1 关键字

以下标识符被作为语言的保留字或称 关键字，不可被用作普通标识符。关键字的拼写必须与这里列出的完全一致。

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

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

在 2.5 版更改: Using *as* and *with* as identifiers triggers a warning. To use them as keywords, enable the with_statement future feature .

在 2.6 版更改: *as* and *with* are full keywords.

### 2.3.2 保留的标识符类

某些标识符类 (除了关键字) 具有特殊的含义。这些标识符类的命名模式是以下划线字符打头和结尾:

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

---

**注解:** _ 作为名称常用于连接国际化文本；请参看 gettext 模块文档了解有关此约定的详情。

---

**__*__** System-defined names. These names are defined by the interpreter and its implementation (including the standard library). Current system names are discussed in the 特殊方法名称 section and elsewhere. More will likely be defined in future versions of Python. *Any* use of __*__ names, in any context, that does not follow explicitly documented use, is subject to breakage without warning.

**__*** 类的私有名称。这种名称在类定义中使用时，会以一种混合形式重写以避免在基类及派生类的“私有”属性之间出现名称冲突。参见标识符（名称）。

## 2.4 字面值

字面值用于表示一些内置类型的常量。

### 2.4.1 String literals

字符串字面值由以下词法定义进行描述:

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

One syntactic restriction not indicated by these productions is that whitespace is not allowed between the *stringprefix* and the rest of the string literal. The source character set is defined by the encoding declaration; it is ASCII if no encoding declaration is given in the source file; see section 编码声明.

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. A prefix of 'b' or 'B' is ignored in Python 2; it indicates that the literal should become a bytes literal in Python 3 (e.g. when code is automatically converted with 2to3). A 'u' or 'b' prefix may be followed by an 'r' prefix.

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:

| 转义序列 | 含义 | 注释 |
|---|---|---|
| \newline | Ignored | |
| \\ | 反斜杠 (\) | |
| \' | 单引号 (') | |
| \" | 双引号 (") | |
| \a | ASCII 响铃 (BEL) | |
| \b | ASCII 退格 (BS) | |
| \f | ASCII 进纸 (FF) | |
| \n | ASCII 换行 (LF) | |
| \N{name} | Character named *name* in the Unicode database (Unicode only) | |
| \r | ASCII 回车 (CR) | |
| \t | ASCII 水平制表 (TAB) | |
| \uxxxx | Character with 16-bit hex value *xxxx* (Unicode only) | (1) |
| \Uxxxxxxxx | Character with 32-bit hex value *xxxxxxxx* (Unicode only) | (2) |
| \v | ASCII 垂直制表 (VT) | |
| \ooo | 八进制数 *ooo* 码位的字符 | (3,5) |
| \xhh | 十六进制数 *hh* 码位的字符 | (4,5) |

注释:

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

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

(3) 与标准 C 一致，接受最多三个八进制数码。

(4) 与标准 C 不同，要求必须为两个十六进制数码。

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

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

When an `'r'` or `'R'` prefix is present, a character following a backslash is included in the string without change, and *all backslashes are left in the string*. For example, the string literal `r"\n"` consists of two characters: a backslash and a lowercase `'n'`. String quotes can be escaped with a backslash, but the backslash remains in the string; for example, `r"\""` is a valid string literal consisting of two characters: a backslash and a double quote; `r"\"` is not a valid string literal (even a raw string cannot end in an odd number of backslashes). Specifically, *a raw string cannot end in a single backslash* (since the backslash would escape the following quote character). Note also that a single backslash followed by a newline is interpreted as those two characters as part of the string, *not* as a line continuation.

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

### 2.4.2 字符串字面值拼接

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 数字字面值

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).

注意数字字面值并不包含正负号；`-1` 这样的负数实际上是由单目运算符‘`-`‘和字面值 `1` 合成的。

### 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 | bininteger
decimalinteger  ::=   nonzerodigit digit* | "0"
octinteger      ::=   "0" ("o" | "O") octdigit+ | "0" octdigit+
hexinteger      ::=   "0" ("x" | "X") hexdigit+
bininteger      ::=   "0" ("b" | "B") bindigit+
nonzerodigit    ::=   "1"..."9"
octdigit        ::=   "0"..."7"
bindigit        ::=   "0" | "1"
hexdigit        ::=   digit | "a"..."f" | "A"..."F"
```

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

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

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

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

### 2.4.5 浮点数字面值

浮点数字面值由以下词法定义进行描述:

```
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

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

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

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

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

### 2.4.6 虚数字面值

虚数字面值由以下词法定义进行描述:

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

一个虚数字面值将生成一个实部为 0.0 的复数。复数是以一对浮点数来表示的，它们的取值范围相同。要创建一个实部不为零的复数，就加上一个浮点数，例如 `(3+4j)`。一些虚数字面值的示例如下:

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

## 2.5 运算符

以下形符属于运算符:

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

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

## 2.6 分隔符

以下形符在语法中归类为分隔符:

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

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.

以下可打印 ASCII 字符作为其他形符的组成部分时具有特殊含义，或是对词法分析器有重要意义:

```
'       "       #       \
```

以下可打印 ASCII 字符不在 Python 词法中使用。如果出现于字符串字面值和注释之外将无条件地引发错误:

```
$       ?
```

### 备注

CHAPTER 3

# 数据模型

## 3.1 对象、值与类型

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

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

对象绝不会被显式地销毁；然而，当无法访问时它们可能会被作为垃圾回收。允许具体的实现推迟垃圾回收或完全省略此机制—如何实现垃圾回收是实现的质量问题，只要可访问的对象不会被回收即可。

**CPython implementation detail:** CPython currently uses a reference-counting scheme with (optional) delayed detection of cyclically linked garbage, which collects most objects as soon as they become unreachable, but is not guaranteed to collect garbage containing circular references. See the documentation of the `gc` module for information on controlling the collection of cyclic garbage. Other implementations act differently and CPython may change. Do not depend on immediate finalization of objects when they become unreachable (ex: always close files).

注意：使用实现的跟踪或调试功能可能令正常情况下会被回收的对象继续存活。还要注意通过'`try`…`except`'语句捕捉异常也可能令对象保持存活。

[1] 在某些情况下 有可能基于可控的条件改变一个对象的类型。但这通常不是个好主意，因为如果处理不当会导致一些非常怪异的行为。

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.

有些对象包含对其他对象的引用；它们被称为 *容器*。容器的例子有元组、列表和字典等。这些引用是容器对象值的组成部分。在多数情况下，当谈论一个容器的值时，我们是指所包含对象的值而不是其编号；但是，当我们谈论一个容器的可变性时，则仅指其直接包含的对象的编号。因此，如果一个不可变容器 (例如元组) 包含对一个可变对象的引用，则当该可变对象被改变时容器的值也会改变。

类型会影响对象行为的几乎所有方面。甚至对象编号的重要性也在某种程度上受到影响: 对于不可变类型，会得出新值的运算实际上会返回对相同类型和取值的任一现有对象的引用，而对于可变类型来说这是不允许的。例如在 `a = 1; b = 1` 之后，`a` 和 `b` 可能会也可能不会指向同一个值为一的对象，这取决于具体实现，但是在 `c = []; d = []` 之后，`c` 和 `d` 保证会指向两个不同、单独的新建空列表。(请注意 `c = d = []` 则是将同一个对象赋值给 `c` 和 `d`。)

## 3.2 标准类型层级结构

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

以下部分类型的描述中包含有'特殊属性列表'段落。这些属性提供对具体实现的访问而非通常使用。它们的定义在未来可能会改变。

**None** 此类型只有一种取值。是一个具有此值的单独对象。此对象通过内置名称 `None` 访问。在许多情况下它被用来表示空值，例如未显式指明返回值的函数将返回 None。它的逻辑值为假。

**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.Number** 此类对象由数字字面值创建，并会被作为算术运算符和算术内置函数的返回结果。数字对象是不可变的；一旦创建其值就不再改变。Python 中的数字当然非常类似数学中的数字，但也受限于计算机中的数字表示方法。

Python 区分整型数、浮点型数和复数:

**numbers.Integral** 此类对象表示数学中整数集合的成员 (包括正数和负数)。

There are three types of integers:

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

**Long integers** 此类对象表示任意大小的数字，仅受限于可用的内存 (包括虚拟内存)。在变换和掩码运算中会以二进制表示，负数会以 2 的补码表示，看起来像是符号位向左延伸补满空位。

**Booleans** These represent the truth values False and True. The two objects representing the values `False` and `True` are the only Boolean objects. The Boolean type is a subtype of plain integers, and Boolean

values behave like the values 0 and 1, respectively, in almost all contexts, the exception being that when converted to a string, the strings `"False"` or `"True"` are returned, respectively.

The rules for integer representation are intended to give the most meaningful interpretation of shift and mask operations involving negative integers and the least surprises when switching between the plain and long integer domains. Any operation, if it yields a result in the plain integer domain, will yield the same result in the long integer domain or when using mixed operands. The switch between domains is transparent to the programmer.

**numbers.Real (float)** 此类对象表示机器级的双精度浮点数。其所接受的取值范围和溢出处理将受制于底层的机器架构 (以及 C 或 Java 实现)。Python 不支持单精度浮点数；支持后者通常的理由是节省处理器和内存消耗，但这点节省相对于在 Python 中使用对象的开销来说太过微不足道，因此没有理由包含两种浮点数而令该语言变得复杂。

**numbers.Complex** 此类对象以一对机器级的双精度浮点数来表示复数值。有关浮点数的附带规则对其同样有效。一个复数值 `z` 的实部和虚部可通过只读属性 `z.real` 和 `z.imag` 来获取。

**序列** 此类对象表示以非负整数作为索引的有限有序集。内置函数 `len()` 可返回一个序列的条目数量。当一个序列的长度为 *n* 时，索引集包含数字 0, 1, …, *n*-1。序列 *a* 的条目 *i* 可通过 `a[i]` 选择。

序列还支持切片: `a[i:j]` 选择索引号为 *k* 的所有条目，*i* <= *k* < *j*。当用作表达式时，序列的切片就是一个与序列类型相同的新序列。新序列的索引还是从 0 开始。

有些序列还支持带有第三个“step”形参的“扩展切片”：`a[i:j:k]` 选择 *a* 中索引号为 *x* 的所有条目，`x = i + n*k`, *n* >= 0 且 *i* <= *x* < *j*。

序列可根据其可变性来加以区分:

**不可变序列** 不可变序列类型的对象一旦创建就不能再改变。(如果对象包含对其他对象的引用，其中的可变对象就是可以改变的；但是，一个不可变对象所直接引用的对象集是不能改变的。)

以下类型属于不可变对象:

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

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

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

**元组** 一个元组中的条目可以是任意 Python 对象。包含两个或以上条目的元组由逗号分隔的表达式构成。只有一个条目的元组 (‘单项元组’) 可通过在表达式后加一个逗号来构成 (一个表达式本身不能创建为元组，因为圆括号要用来设置表达式分组)。一个空元组可通过一对内容为空的圆括号创建。

**可变序列** 可变序列在被创建后仍可被改变。下标和切片标注可被用作赋值和*del* (删除) 语句的目标。

目前有两种内生可变序列类型:

**列表** 列表中的条目可以是任意 Python 对象。列表由用方括号括起并由逗号分隔的多个表达式构成。(注意创建长度为 0 或 1 的列表无需使用特殊规则。)

**字节数组** A bytearray object is a mutable array. They are created by the built-in `bytearray()` constructor. Aside from being mutable (and hence unhashable), byte arrays otherwise provide the same interface and functionality as immutable bytes objects.

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

**集合类型** 此类对象表示由不重复且不可变对象组成的无序且有限的集合。因此它们不能通过下标来索引。但是它们可被迭代，也可用内置函数 `len()` 返回集合中的条目数。集合常见的用处是快速成员检测，去除序列中的重复项，以及进行交、并、差和对称差等数学运算。

对于集合元素所采用的不可变规则与字典的键相同。注意数字类型遵循正常的数字比较规则: 如果两个数字相等 (例如 `1` 和 `1.0`)，则同一集合中只能包含其中一个。

目前有两种内生集合类型:

**集合** 此类对象表示可变集合。它们可通过内置的 `set()` 构造器创建，并且创建之后可以通过方法进行修改，例如 `add()`。

**冻结集合** 此类对象表示不可变集合。它们可通过内置的 `frozenset()` 构造器创建。由于 frozenset 对象不可变且*hashable*，它可以被用作另一个集合的元素或是字典的键。

**映射** 此类对象表示由任意索引集合所索引的对象的集合。通过下标 `a[k]` 可在映射 `a` 中选择索引为 `k` 的条目；这可以在表达式中使用，也可作为赋值或*del* 语句的目标。内置函数 `len()` 可返回一个映射中的条目数。

目前只有一种内生映射类型:

**字典** 此类对象表示由几乎任意值作为索引的有限个对象的集合。不可作为键的值类型只有包含列表或字典或其他可变类型，通过值而非对象编号进行比较的值，其原因在于高效的字典实现需要使用键的哈希值以保持一致性。用作键的数字类型遵循正常的数字比较规则: 如果两个数字相等 (例如 `1` 和 `1.0`) 则它们均可来用来索引同一个字典条目。

字典是可变的；它们可通过 `{...}` 标注来创建 (参见字典显示 小节)。

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

**可调用类型** 此类型可以被应用于函数调用操作 (参见调用 小节):

**用户定义函数** 用户定义函数对象可通过函数定义来创建 (参见函数定义 小节)。它被调用时应附带一个参数列表，其中包含的条目应与函数所定义的形参列表一致。

特殊属性:

| 属性 | 含义 | |
|---|---|---|
| `__doc__` `func_doc` | The function’s documentation string, or `None` if unavailable. | 可写 |
| `__name__` `func_name` | The function’s name | 可写 |
| `__module__` | 该函数所属模块的名称，没有则为 `None`。 | 可写 |
| `__defaults__` `func_defaults` | 由具有默认值的参数的默认参数值组成的元组，如无任何参数具有默认值则为 `None`。 | 可写 |
| `__code__` `func_code` | 表示编译后的函数体的代码对象。 | 可写 |
| `__globals__` `func_globals` | 对存放该函数中全局变量的字典的引用—函数所属模块的全局命名空间。 | 只读 |
| `__dict__` `func_dict` | 命名空间支持的函数属性。 | 可写 |
| `__closure__` `func_closure` | `None` or a tuple of cells that contain bindings for the function’s free variables. | 只读 |

大部分标有“Writable”的属性均会检查赋值的类型。

在 2.4 版更改: func_name is now writable.

在 2.6 版更改: The double-underscore attributes __closure__, __code__, __defaults__, and __globals__ were introduced as aliases for the corresponding func_* attributes for forwards compatibility with Python 3.

函数对象也支持获取和设置任意属性，例如这可以被用来给函数附加元数据。使用正规的属性点号标注获取和设置此类属性。注意当前实现仅支持用户定义函数属性。未来可能会增加支持内置函数属性。

有关函数定义的额外信息可以从其代码对象中提取；参见下文对内部类型的描述。

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

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

在 2.2 版更改: im_self used to refer to the class that defined the method.

在 2.6 版更改: For Python 3 forward-compatibility, im_func is also available as __func__, and im_self as __self__.

方法还支持获取 (但不能设置) 下层函数对象的任意函数属性。

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

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

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

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

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

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

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

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

**生成器函数** A function or method which uses the *yield* statement (see section *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.

**内置函数** 内置函数对象是对于 C 函数的外部封装。内置函数的例子包括 `len()` 和 `math.sin()` (`math` 是一个标准内置模块)。内置函数参数的数量和类型由 C 函数决定。特殊的只读属性: `__doc__` 是函数的文档字符串，如果没有则为 `None`; `__name__` 是函数的名称; `__self__` 设定为 `None` (参见下一条目); `__module__` 是函数所属模块的名称，如果没有则为 `None`。

**内置方法** 此类型实际上是内置函数的另一种形式，只不过还包含了一个传入 C 函数的对象作为隐式的额外参数。内置方法的一个例子是 `alist.append()`，其中 *alist* 为一个列表对象。在此示例中，特殊的只读属性 `__self__` 会被设为 *alist* 所标记的对象。

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

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

**类实例** 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)`.

**模块** Modules are imported by the *import* statement (see section *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).

属性赋值会更新模块的命名空间字典，例如 `m.x = 1` 等同于 `m.__dict__["x"] = 1`。

特殊的只读属性: `__dict__` 为以字典对象表示的模块命名空间。

由于 CPython 清理模块字典的设定，当模块离开作用域时模块字典将会被清理，即使该字典还有活动的引用。想避免此问题，可复制该字典或保持模块状态以直接使用其字典。

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.

**类** Both class types (new-style classes) and class objects (old-style/classic classes) are typically created by class definitions (see section 类定义). 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"]` (although for new-style classes in particular there are a number of hooks which allow for other means of locating attributes). When the attribute name is not found there, the attribute search continues in the base classes. For old-style classes, the search is depth-first, left-to-right in the order of occurrence in the base class list. New-style classes use the more

complex C3 method resolution order which behaves correctly even in the presence of 'diamond' inheritance structures where there are multiple inheritance paths leading back to a common ancestor. Additional details on the C3 MRO used by new-style classes can be found in the documentation accompanying the 2.3 release at https://www.python.org/download/releases/2.3/mro/.

When a class attribute reference (for class `C`, say) would yield a user-defined function object or an unbound user-defined method object whose associated class is either `C` or one of its base classes, it is transformed into an unbound user-defined method object whose `im_class` attribute is `C`. When it would yield a class method object, it is transformed into a bound user-defined method object whose `im_self` attribute is `C`. When it would yield a static method object, it is transformed into the object wrapped by the static method object. See section 实现描述器 for another way in which attributes retrieved from a class may differ from those actually contained in its `__dict__` (note that only new-style classes support descriptors).

类属性赋值会更新类的字典，但不会更新基类的字典。

类对象可被调用 (见上文) 以产生一个类实例 (见下文)。

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.

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

属性赋值和删除会更新实例的字典，但不会更新对应类的字典。如果类具有*`__setattr__()`*或*`__delattr__()`*方法，则将调用方法而不再直接更新实例的字典。

如果类实例具有某些特殊名称的方法，就可以伪装为数字、序列或映射。参见特殊方法名称 一节。

特殊属性: `__dict__` 为属性字典; `__class__` 为实例对应的类。

**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 bltin-file-objects for complete documentation of file objects.

**内部类型** 某些由解释器内部使用的类型也被暴露给用户。它们的定义可能随未来解释器版本的更新而变化，为内容完整起见在此处一并介绍。

**代码对象** 代码对象表示 编译为字节的可执行 Python 代码，或称*bytecode*。代码对象和函数对象的区别在于函数对象包含对函数全局对象 (函数所属的模块) 的显式引用，而代码对象不包含上下文；而且默认参数值会存放于函数对象而不是代码对象内 (因为它们表示在运行时算出的值)。与函数对象不同，代码对象不可变，也不包含对可变对象的引用 (不论是直接还是间接)。

Special read-only attributes: `co_name` gives the function name; `co_argcount` is the number of positional arguments (including arguments with default values); `co_nlocals` is the number of local variables used by the function (including arguments); `co_varnames` is a tuple containing the names of the local variables (starting with the argument names); `co_cellvars` is a tuple containing the names of local variables that are referenced by nested functions; `co_freevars` is a tuple containing the names of free variables; `co_code` is a string representing the sequence of bytecode instructions; `co_consts` is a tuple containing the literals

used by the bytecode; co_names is a tuple containing the names used by the bytecode; co_filename is the filename from which the code was compiled; co_firstlineno is the first line number of the function; co_lnotab is a string encoding the mapping from bytecode 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.

以下是可用于 co_flags 的标志位定义：如果函数使用 *arguments 语法来接受任意数量的位置参数，则 0x04 位被设置；如果函数使用 **keywords 语法来接受任意数量的关键字参数，则 0x08 位被设置；如果函数是一个生成器，则 0x20 位被设置。

未来特性声明 (from __future__ import division) 也使用 co_flags 中的标志位来指明代码对象的编译是否启用特定的特性: 如果函数编译时启用未来除法特性则设置 0x2000 位; 在更早的 Python 版本中则使用 0x10 和 0x1000 位。

co_flags 中的其他位被保留为内部使用。

如果代码对象表示一个函数，co_consts 中的第一项将是函数的文档字符串，如果未定义则为 None。

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

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

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

**回溯对象** 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 *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.

**切片对象** 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.

特殊的只读属性: start 为下界; stop 为上界; step 为步长值; 各值如省略则为 None。这些属性可具有任意类型。

切片对象支持一个方法:

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

2.3 新版功能.

**静态方法对象** 静态方法对象提供了一种避免上文所述将函数对象转换为方法对象的方式。静态方法对象为对任意其他对象的封装，通常用来封装用户定义方法对象。当从类或类实例获取一个静态方法对象时，实际返回的对象是封装的对象，它不会被进一步转换。静态方法对象自身不是可调用的，但它们所封装的对象通常都是可调用的。静态方法对象可通过内置的 `staticmethod()` 构造器来创建。

**类方法对象** 类方法对象和静态方法一样是对其他对象的封装，会改变从类或类实例获取该对象的方式。类方法对象在此类获取操作中的行为已在上文“用户定义方法”一节中描述。类方法对象可通过内置的 `classmethod()` 构造器来创建。

## 3.3 New-style and classic classes

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

Up to Python 2.1 the concept of `class` was unrelated to the concept of `type`, and old-style classes were the only flavor available. For an old-style class, the statement `x.__class__` provides the class of *x*, but `type(x)` is always `<type 'instance'>`. This reflects the fact that all old-style instances, independent of their class, are implemented with a single built-in type, called `instance`.

New-style classes were introduced in Python 2.2 to unify the concepts of `class` and `type`. A new-style class is simply a user-defined type, no more, no less. If *x* is an instance of a new-style class, then `type(x)` is typically the same as `x.__class__` (although this is not guaranteed –a new-style class instance is permitted to override the value returned for `x.__class__`).

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

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

While this manual aims to provide comprehensive coverage of Python’s class mechanics, it may still be lacking in some areas when it comes to its coverage of new-style classes. Please see https://www.python.org/doc/newstyle/ for sources of additional information.

Old-style classes are removed in Python 3, leaving only new-style classes.

## 3.4 特殊方法名称

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

在实现模拟任何内置类型的类时，很重要的一点是模拟的实现程度对于被模拟对象来说应当是有意义的。例如，提取单个元素的操作对于某些序列来说是适宜的，但提取切片可能就没有意义。(这种情况的一个实例是 W3C 的文档对象模型中的 `NodeList` 接口。)

### 3.4.1 基本定制

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

调用以创建一个 *cls* 类的新实例。`__new__()` 是一个静态方法 (因为是特例所以你不需要显式地声明)，它会将所请求实例所属的类作为第一个参数。其余的参数会被传递给对象构造器表达式 (对类的调用)。`__new__()` 的返回值应为新对象实例 (通常是 *cls* 的实例)。

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

如果`__new__()` 返回一个 *cls* 的实例，则新实例的`__init__()` 方法会在之后被执行，例如 `__init__(self[, ...])`，其中 *self* 为新实例，其余的参数与被传递给`__new__()` 的相同。

如果`__new__()` 未返回一个 *cls* 的实例，则新实例的`__init__()` 方法就不会被执行。

`__new__()` 的目的主要是允许不可变类型的子类 (例如 int, str 或 tuple) 定制实例创建过程。它也常会在自定义元类中被重载以便定制类创建过程。

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

Called after the instance has been created (by `__new__()`), but before it is returned to the caller. The arguments are those passed to the class constructor expression. If a base class has an `__init__()` method, the derived class's `__init__()` method, if any, must explicitly call it to ensure proper initialization of the base class part of the instance; for example: `BaseClass.__init__(self, [args...])`.

Because `__new__()` and `__init__()` work together in constructing objects (`__new__()` to create it, and `__init__()` to customise it), no non-`None` value may be returned by `__init__()`; doing so will cause a `TypeError` to be raised at runtime.

`object.`**`__del__`**`(`*self*`)`

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

---

**注解：** `del x` doesn't directly call `x.__del__()` —the former decrements the reference count for `x` by one, and the latter is only called when `x`'s reference count reaches zero. Some common situations that may prevent the reference count of an object from going to zero include: circular references between objects (e.g., a doubly-linked list or a tree data structure with parent and child pointers); a reference to the object on the stack frame of a function that caught an exception (the traceback stored in `sys.exc_traceback` keeps the stack frame alive); or

a reference to the object on the stack frame that raised an unhandled exception in interactive mode (the traceback stored in `sys.last_traceback` keeps the stack frame alive). The first situation can only be remedied by explicitly breaking the cycles; the latter two situations can be resolved by storing `None` in `sys.exc_traceback` or `sys.last_traceback`. Circular references which are garbage are detected when the option cycle detector is enabled (it's on by default), but can only be cleaned up if there are no Python-level *__del__()* methods involved. Refer to the documentation for the `gc` module for more information about how *__del__()* methods are handled by the cycle detector, particularly the description of the `garbage` value.

**警告：** 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 or in the process of being torn down (e.g. the import machinery shutting down). For this reason, *__del__()* methods should do the absolute minimum needed to maintain external invariants. Starting with version 1.5, Python guarantees that globals whose name begins with a single underscore are deleted from their module before other globals are deleted; if no other references to such globals exist, this may help in assuring that imported modules are still available at the time when the *__del__()* method is called.

See also the `-R` command-line option.

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

此方法通常被用于调试，因此确保其表示的内容包含丰富信息且无歧义是很重要的。

`object.`**`__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.

`object.`**`__lt__`**`(`*self*`, `*other*`)`
`object.`**`__le__`**`(`*self*`, `*other*`)`
`object.`**`__eq__`**`(`*self*`, `*other*`)`
`object.`**`__ne__`**`(`*self*`, `*other*`)`
`object.`**`__gt__`**`(`*self*`, `*other*`)`
`object.`**`__ge__`**`(`*self*`, `*other*`)`
2.1 新版功能.

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)`.

如果指定的参数对没有相应的实现，富比较方法可能会返回单例对象 `NotImplemented`。按照惯例，成功的比较会返回 `False` 或 `True`。不过实际上这些方法可以返回任意值，因此如果比较运算符是要用于布尔值判断（例如作为 `if` 语句的条件），Python 会对返回值调用 `bool()` 以确定结果为真还是假。

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

behave as expected. See the paragraph on *__hash__()* for some important notes on creating *hashable* objects which support custom comparison operations and are usable as dictionary keys.

There are no 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.

To automatically generate ordering operations from a single root operation, see `functools.total_ordering()`.

`object.`**`__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 *hashable* objects which support custom comparison operations and are usable as dictionary keys. (Note: the restriction that exceptions are not propagated by *__cmp__()* has been removed since Python 1.5.)

`object.`**`__rcmp__`**(*self*, *other*)

在 2.1 版更改: No longer supported.

`object.`**`__hash__`**(*self*)

Called by built-in function `hash()` and for operations on members of hashed collections including `set`, `frozenset`, and `dict`. *__hash__()* should return an integer. The only required property is that objects which compare equal have the same hash value; it is advised to mix together the hash values of the components of the object that also play a part in comparison of objects by packing them into a tuple and hashing the tuple. Example:

```
def __hash__(self):
    return hash((self.name, self.nick, self.color))
```

If a class does not define a *__cmp__()* or *__eq__()* method it should not define a *__hash__()* operation either; if it defines *__cmp__()* or *__eq__()* but not *__hash__()*, its instances will not be usable in hashed collections. If a class defines mutable objects and implements a *__cmp__()* or *__eq__()* method, it should not implement *__hash__()*, since hashable collection implementations require that an object's hash value is immutable (if the object's hash value changes, it will be in the wrong hash bucket).

User-defined classes have *__cmp__()* and *__hash__()* methods by default; with them, all objects compare unequal (except with themselves) and `x.__hash__()` returns a result derived from `id(x)`.

Classes which inherit a *__hash__()* method from a parent class but change the meaning of *__cmp__()* or *__eq__()* such that the hash value returned is no longer appropriate (e.g. by switching to a value-based concept of equality instead of the default identity based equality) can explicitly flag themselves as being unhashable by setting `__hash__ = None` in the class definition. Doing so means that not only will instances of the class raise an appropriate `TypeError` when a program attempts to retrieve their hash value, but they will also be correctly identified as unhashable when checking `isinstance(obj, collections.Hashable)` (unlike classes which define their own *__hash__()* to explicitly raise `TypeError`).

在 2.5 版更改: *__hash__()* may now also return a long integer object; the 32-bit integer is then derived from the hash of that object.

在 2.6 版更改: *__hash__* may now be set to `None` to explicitly flag instances of a class as unhashable.

`object.`**`__nonzero__`**(*self*)

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

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

### 3.4.2 自定义属性访问

可以定义下列方法来自定义对类实例属性访问（`x.name` 的使用、赋值或删除）的具体含义.

object.**__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 *__getattr__()* would have no way to access other attributes of the instance. Note that at least for instance variables, you can fake total control by not inserting any values in the instance attribute dictionary (but instead inserting them in another object). See the *__getattribute__()* method below for a way to actually get total control in new-style classes.

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`. For new-style classes, rather than accessing the instance dictionary, it should call the base class method with the same name, for example, `object.__setattr__(self, name, value)`.

object.**__delattr__**(*self*, *name*)
类似于*__setattr__()* 但其作用为删除而非赋值。此方法应该仅在 `del obj.name` 对于该对象有意义时才被实现。

#### More attribute access for new-style classes

The following methods only apply to new-style classes.

object.**__getattribute__**(*self*, *name*)
此方法会无条件地被调用以实现对类实例属性的访问。如果类还定义了*__getattr__()*，则后者不会被调用，除非*__getattribute__()* 显式地调用它或是引发了 `AttributeError`。此方法应当返回（找到的）属性值或是引发一个 `AttributeError` 异常。为了避免此方法中的无限递归，其实现应该总是调用具有相同名称的基类方法来访问它所需要的任何属性，例如 `object.__getattribute__(self, name)`。

---

**注解：** This method may still be bypassed when looking up special methods as the result of implicit invocation via language syntax or built-in functions. See *Special method lookup for new-style classes*.

---

### 实现描述器

以下方法仅当一个包含该方法的类（称为 *描述器类*）的实例出现于一个 *所有者类* 中的时候才会起作用（该描述器必须在所有者类或其某个上级类的字典中）。在以下示例中，“属性”指的是名称为所有者类 `__dict__` 中的特征属性的键名的属性。

`object.`**`__get__`**`(`*self*, *instance*, *owner*`)`
调用此方法以获取所有者类的属性（类属性访问）或该类的实例的属性（实例属性访问）。*所有者* 是指所有者类，而 *实例* 是指被用来访问属性的实例，如果是 *所有者* 被用来访问属性时则为 `None`。此方法应当返回（计算出的）属性值或是引发一个 `AttributeError` 异常。

`object.`**`__set__`**`(`*self*, *instance*, *value*`)`
调用此方法以设置 *instance* 指定的所有者类的实例的属性为新值 *value*。

`object.`**`__delete__`**`(`*self*, *instance*`)`
调用此方法以删除 *instance* 指定的所有者类的实例的属性。

### 发起调用描述器

总的说来，描述器就是具有“绑定行为”的对象属性，其属性访问已被描述器协议中的方法所重载，包括 *__get__()*, *__set__()* 和 *__delete__()*。如果一个对象定义了以上方法中的任意一个，它就被称为描述器。

属性访问的默认行为是从一个对象的字典中获取、设置或删除属性。例如，`a.x` 的查找顺序会从 `a.__dict__['x']` 开始，然后是 `type(a).__dict__['x']`，接下来依次查找 `type(a)` 的上级基类，不包括元类。

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

描述器发起调用的开始点是一个绑定 `a.x`。参数的组合方式依 `a` 而定:

**直接调用** 最简单但最不常见的调用方式是用户代码直接发起调用一个描述器方法: `x.__get__(a)`。

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

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

**超绑定** 如果 `a` 是 `super` 的一个实例，则绑定 `super(B, obj).m()` 会在 `obj.__class__.__mro__` 中搜索 `B` 的直接上级基类 `A` 然后通过以下调用发起调用描述器: `A.__dict__['m'].__get__(obj, obj.__class__)`。

对于实例绑定，发起描述器调用的优先级取决于定义了哪些描述器方法。一个描述器可以定义 *__get__()*、*__set__()* 和 *__delete__()* 的任意组合。如果它没有定义 *__get__()*，则访问属性会返回描述器对象自身，除非对象的实例字典中有相应属性值。如果描述器定义了 *__set__()* 和/或 *__delete__()*，则它是一个数据描述器；如果以上两个都未定义，则它是一个非数据描述器。通常，数据描述器会同时定义 *__get__()* 和 *__set__()*，而非数据描述器只有 *__get__()* 方法。定义了 *__set__()* 和 *__get__()* 的数据描述器总是会重载实例字典中的定义。与之相对的，非数据描述器可被实例所重载。

Python 方法 (包括 `staticmethod()` 和 `classmethod()`) 都是作为非描述器来实现的。因此实例可以重定义并重载方法。这允许单个实例获得与相同类的其他实例不一样的行为。

`property()` 函数是作为数据描述器来实现的。因此实例不能重载特性属性的行为。

### __slots__

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

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

**__slots__**

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

2.2 新版功能.

使用 *__slots__* 的注意事项

- When inheriting from a class without *__slots__*, the *__dict__* attribute of that class will always be accessible, so a *__slots__* definition in the subclass is meaningless.
- 没有 *__dict__* 变量，实例就不能给未在 *__slots__* 定义中列出的新变量赋值。尝试给一个未列出的变量名赋值将引发 `AttributeError`。新变量需要动态赋值，就要将 `'__dict__'` 加入到 *__slots__* 声明的字符串序列中。

  在 2.3 版更改: Previously, adding `'__dict__'` to the *__slots__* declaration would not enable the assignment of new attributes not specifically listed in the sequence of instance variable names.
- 如果未给每个实例设置 *__weakref__* 变量，定义了 *__slots__* 的类就不支持对其实际的弱引用。如果需要弱引用支持，就要将 `'__weakref__'` 加入到 *__slots__* 声明的字符串序列中。

  在 2.3 版更改: Previously, adding `'__weakref__'` to the *__slots__* declaration would not enable support for weak references.
- *__slots__* 是通过为每个变量名创建描述器 (实现描述器) 在类层级上实现的。因此，类属性不能被用来为通过 *__slots__* 定义的实例变量设置默认值；否则，类属性就会覆盖描述器赋值。
- The action of a *__slots__* declaration is limited to the class where it is defined. As a result, subclasses will have a *__dict__* unless they also define *__slots__* (which must only contain names of any *additional* slots).
- 如果一个类定义的位置在某个基类中也有定义，则由基类位置定义的实例变量将不可访问（除非通过直接从基类获取其描述器的方式）。这会使得程序的含义变成未定义。未来可能会添加一个防止此情况的检查。
- Nonempty *__slots__* does not work for classes derived from “variable-length” built-in types such as `long`, `str` and `tuple`.
- 任何非字符串可迭代对象都可以被赋值给 *__slots__*。映射也可以被使用；不过，未来可能会分别赋给每个键具有特殊含义的值。
- *__class__* 赋值仅在两个类具有相同的 *__slots__* 时才会起作用。

  在 2.6 版更改: Previously, *__class__* assignment raised an error if either new or old class had *__slots__*.

### 3.4.3 自定义类创建

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

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

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

These steps will have to be performed in the metaclass's *__new__()* method –`type.__new__()` can then be called from this method to create a class with different properties. This example adds a new element to the class dictionary before creating the class:

```
class metacls(type):
    def __new__(mcs, name, bases, dict):
        dict['foo'] = 'metacls was here'
        return type.__new__(mcs, name, bases, dict)
```

You can of course also override other class methods (or add new methods); for example defining a custom *__call__()* method in the metaclass allows custom behavior when the class is called, e.g. not always creating a new instance.

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

  2.2 新版功能.

The appropriate metaclass is determined by the following precedence rules:

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

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

### 3.4.4 自定义实例及子类检查

2.6 新版功能.

以下方法被用来重载 `isinstance()` 和 `issubclass()` 内置函数的默认行为。

特别地，元类 `abc.ABCMeta` 实现了这些方法以便允许将抽象基类（ABC）作为“虚拟基类”添加到任何类或类型（包括内置类型），包括其他 ABC 之中。

`class.`**`__instancecheck__`**`(`*self*, *instance*`)`
: 如果 *instance* 应被视为 *class* 的一个（直接或间接）实例则返回真值。如果定义了此方法，则会被调用以实现 `isinstance(instance, class)`。

`class.`**`__subclasscheck__`**`(`*self*, *subclass*`)`
: Return true 如果 *subclass* 应被视为 *class* 的一个（直接或间接）子类则返回真值。如果定义了此方法，则会被调用以实现 `issubclass(subclass, class)`。

请注意这些方法的查找是基于类的类型（元类）。它们不能作为类方法在实际的类中被定义。这与基于实例被调用的特殊方法的查找是一致的，只有在此情况下实例本身被当作是类。

**参见:**

**PEP 3119 - 引入抽象基类** 新增功能描述，通过 *__instancecheck__()* 和 *__subclasscheck__()* 来定制 `isinstance()` 和 `issubclass()` 行为，加入此功能的动机是出于向该语言添加抽象基类的内容（参见 `abc` 模块）。

### 3.4.5 模拟可调用对象

`object.`**`__call__`**(*self*[, *args...*])
此方法会在实例作为一个函数被“调用”时被调用；如果定义了此方法，则 `x(arg1, arg2, ...)` 就相当于 `x.__call__(arg1, arg2, ...)` 的快捷方式。

### 3.4.6 模拟容器类型

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

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

**CPython implementation detail:** In CPython, the length is required to be at most `sys.maxsize`. If the length is larger than `sys.maxsize` some features (such as `len()`) may raise `OverflowError`. To prevent raising `OverflowError` by truth value testing, an object must define a *__nonzero__()* method.

`object.`**`__getitem__`**(*self*, *key*)
调用此方法以实现 `self[key]` 的求值。对于序列类型，接受的键应为整数和切片对象。请注意负数索引（如果类想要模拟序列类型）的特殊解读是取决于 *__getitem__()* 方法。如果 *key* 的类型不正确则会引发 `TypeError` 异常；如果为序列索引集范围以外的值（在进行任何负数索引的特殊解读之后）则应引发 `IndexError` 异常。对于映射类型，如果 *key* 找不到（不在容器中）则应引发 `KeyError` 异常。

**注解:** *for* 循环在有不合法索引时会期待捕获 IndexError 以便正确地检测到序列的结束。

object.**__setitem__**(*self*, *key*, *value*)
调用此方法以实现向 self[key] 赋值。注意事项与*__getitem__()* 相同。为对象实现此方法应该仅限于需要映射允许基于键修改值或添加键，或是序列允许元素被替换时。不正确的 *key* 值所引发的异常应与*__getitem__()* 方法的情况相同。

object.**__delitem__**(*self*, *key*)
调用此方法以实现 self[key] 的删除。注意事项与*__getitem__()* 相同。为对象实现此方法应该权限于需要映射允许移除键，或是序列允许移除元素时。不正确的 *key* 值所引发的异常应与*__getitem__()* 方法的情况相同。

object.**__missing__**(*self*, *key*)
此方法由 dict.*__getitem__()* 在找不到字典中的键时调用以实现 dict 子类的 self[key]。

object.**__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().

迭代器对象也需要实现此方法；它们需要返回对象自身。有关迭代器对象的详情请参看 typeiter 一节。

object.**__reversed__**(*self*)
此方法（如果存在）会被 reversed() 内置函数调用以实现逆向迭代。它应当返回一个新的以逆序逐个迭代容器内所有对象的迭代器对象。

如果未提供*__reversed__()* 方法，则 reversed() 内置函数将回退到使用序列协议 (*__len__()* 和*__getitem__()*)。支持序列协议的对象应当仅在能够提供比 reversed() 所提供的实现更高效的实现时才提供*__reversed__()* 方法。

2.6 新版功能.

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.

object.**__contains__**(*self*, *item*)
调用此方法以实现成员检测运算符。如果 *item* 是 *self* 的成员则应返回真，否则返回假。对于映射类型，此检测应基于映射的键而不是值或者键值对。

对于未定义*__contains__()* 的对象，成员检测将首先尝试通过*__iter__()* 进行迭代，然后再使用*__getitem__()* 的旧式序列迭代协议，参看语言参考中的相应部分。

### 3.4.7 Additional methods for emulation of sequence types

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

object.**__getslice__**(*self*, *i*, *j*)
2.0 版后已移除: Support slice objects as parameters to the *__getitem__()* method. (However, built-in types in CPython currently still implement *__getslice__()*. Therefore, you have to override it in derived classes when implementing slicing.)

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.maxsize, 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.

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

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

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

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

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

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

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

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

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

### 3.4.8 模拟数字类型

定义以下方法即可模拟数字类型。特定种类的数字不支持的运算（例如非整数不能进行位运算）所对应的方法应当保持未定义状态。

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

如果这些方法中的某一个不支持与所提供参数进行运算，它应该返回 NotImplemented。

object.**__div__**(*self*, *other*)
object.**__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.

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

These methods are called to implement the binary arithmetic operations (+, -, *, /, %, divmod(), pow(), **, <<, >>, &, ^, |) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation and the operands are of different types.[2] For instance, to evaluate the expression `x - y`, where *y* is an instance of a class that has an *__rsub__()* method, `y.__rsub__(x)` is called if `x.__sub__(y)` returns *NotImplemented*.

请注意三元版的 pow() 并不会尝试调用*__rpow__()* (因为强制转换规则会太过复杂)。

[2] 对于相同类型的操作数，如果非反射方法 (例如*__add__()*) 失败则会认为相应运算不被支持，这就是反射方法未被调用的原因。

**注解:** 如果右操作数类型为左操作数类型的一个子类，且该子类提供了指定运算的反射方法，则此方法会先于左操作数的非反射方法被调用。此行为可允许子类重载其祖先类的运算符。

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

These methods are called to implement the augmented arithmetic assignments (`+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=`, `|=`). 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 assignment falls back to the normal methods. For instance, to execute the statement `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`.

object.**__neg__**(*self*)
object.**__pos__**(*self*)
object.**__abs__**(*self*)
object.**__invert__**(*self*)

调用此方法以实现一元算术运算 (`-`, `+`, `abs()` 和 `~`)。

object.**__complex__**(*self*)
object.**__int__**(*self*)
object.**__long__**(*self*)
object.**__float__**(*self*)

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

object.**__oct__**(*self*)
object.**__hex__**(*self*)

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

object.**__index__**(*self*)

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

2.5 新版功能.

object.**__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). A return value of `NotImplemented` is equivalent to returning `None`.

### 3.4.9 Coercion rules

This section used to document the rules for coercion. As the language has evolved, the coercion rules have become hard to document precisely; documenting what one version of one particular implementation does is undesirable. Instead, here are some informal guidelines regarding coercion. In Python 3, coercion will not be supported.

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

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

  在 2.7 版更改: The complex type no longer makes implicit calls to the *__coerce__()* method for mixed-type binary arithmetic operations.

### 3.4.10 with 语句上下文管理器

2.5 新版功能.

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

上下文管理器的典型用法包括保存和恢复各种全局状态，锁定和解锁资源，关闭打开的文件等等。

要了解上下文管理器的更多信息，请参阅 typecontextmanager。

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

`object.__exit__`(*self*, *exc_type*, *exc_value*, *traceback*)
退出关联到此对象的运行时上下文。各个参数描述了导致上下文退出的异常。如果上下文是无异常地退出的，三个参数都将为 `None`。

如果提供了异常，并且希望方法屏蔽此异常（即避免其被传播），则应当返回真值。否则的话，异常将在退出此方法时按正常流程处理。

请注意`__exit__()` 方法不应该重新引发被传入的异常，这是调用者的责任。

**参见:**

**PEP 343 - "with" 语句** Python `with` 语句的规范描述、背景和示例。

### 3.4.11 Special method lookup for old-style classes

For old-style classes, special methods are always looked up in exactly the same way as any other method or attribute. This is the case regardless of whether the method is being looked up explicitly as in `x.__getitem__(i)` or implicitly as in `x[i]`.

This behaviour means that special methods may exhibit different behaviour for different instances of a single old-style class if the appropriate special attributes are set differently:

```
>>> class C:
...     pass
...
>>> c1 = C()
>>> c2 = C()
>>> c1.__len__ = lambda: 5
>>> c2.__len__ = lambda: 9
>>> len(c1)
5
>>> len(c2)
9
```

### 3.4.12 Special method lookup for new-style classes

For new-style classes, implicit invocations of special methods are only guaranteed to work correctly if defined on an object's type, not in the object's instance dictionary. That behaviour is the reason why the following code raises an exception (unlike the equivalent example with old-style classes):

```
>>> class C(object):
...     pass
...
>>> c = C()
>>> c.__len__ = lambda: 5
>>> len(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'C' has no len()
```

此行为背后的原理在于包括类型对象在内的所有对象都会实现的几个特殊方法，例如*__hash__()*和*__repr__()*。如果这些方法的隐式查找使用了传统的查找过程，它们会在对类型对象本身发起调用时失败:

```
>>> 1 .__hash__() == hash(1)
True
>>> int.__hash__() == hash(int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor '__hash__' of 'int' object needs an argument
```

以这种方式不正确地尝试发起调用一个类的未绑定方法有时被称为‘元类混淆’，可以通过在查找特殊方法时绕过实例的方式来避免:

```
>>> type(1).__hash__(1) == hash(1)
True
>>> type(int).__hash__(int) == hash(int)
True
```

除了为了正确性而绕过任何实例属性之外，隐式特殊方法查找通常也会绕过*__getattribute__()*方法，甚至包括对象的元类:

```
>>> class Meta(type):
...    def __getattribute__(*args):
...       print "Metaclass getattribute invoked"
...       return type.__getattribute__(*args)
...
>>> class C(object):
...     __metaclass__ = Meta
...     def __len__(self):
...         return 10
...     def __getattribute__(*args):
...         print "Class getattribute invoked"
...         return object.__getattribute__(*args)
...
>>> c = C()
>>> c.__len__()                 # Explicit lookup via instance
Class getattribute invoked
10
>>> type(c).__len__(c)          # Explicit lookup via type
Metaclass getattribute invoked
```

(下页继续)

(续上页)

```
10
>>> len(c)                      # Implicit lookup
10
```

以这种方式绕过__getattribute__()机制为解析器内部的速度优化提供了显著的空间，其代价则是牺牲了处理特殊方法时的一些灵活性（特殊方法必须设置在类对象本身上以便始终一致地由解释器发起调用）。

### 备注

CHAPTER 4

# 执行模型

## 4.1 命名与绑定

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

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

代码块在 *执行帧* 中被执行。一个帧会包含某些管理信息（用于调试）并决定代码块执行完成后应前往何处以及如何继续执行。

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 –this includes generator expressions since they are implemented using a function scope. This means that the following will fail:

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

当一个名称在代码块中被使用时，会由包含它的最近作用域来解析。对一个代码块可见的所有这种作用域的集合称为该代码块的 *环境*。

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

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

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

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

每条赋值或导入语句均发生于类或函数内部定义的代码块中，或是发生于模块层级（即最高层级的代码块）。

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

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

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

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

模块的作用域会在模块第一次被导入时自动创建。一个脚本的主模块总是被命名为 `__main__`。

*global* 语句与同一代码块中名称绑定具有相同的作用域。如果一个自由变量的最近包含作用域中有一条 global 语句，则该自由变量也会被当作是全局变量。

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

### 4.1.1 与动态特性的交互

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

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

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

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

The `eval()`, `execfile()`, and `input()` functions and the *exec* statement do not have access to the full environment for resolving names. Names may be resolved in the local and global namespaces of the caller. Free variables are not

resolved in the nearest enclosing namespace, but in the global namespace.[1] The *exec* statement and the `eval()` and `execfile()` functions have optional arguments to override the global and local namespace. If only one namespace is specified, it is used for both.

## 4.2 异常

异常是中断代码块的正常控制流程以便处理错误或其他异常条件的一种方式。异常会在错误被检测到的位置引发，它可以被当前包围代码块或是任何直接或间接发起调用发生错误的代码块的其他代码块所处理。

Python 解析器会在检测到运行时错误（例如零作为被除数）的时候引发异常。Python 程序也可以通过*raise*语句显式地引发异常。异常处理是通过*try* ··· *except* 语句来指定的。该语句的*finally* 子句可被用来指定清理代码，它并不处理异常，而是无论之前的代码是否发生异常都会被执行。

Python 的错误处理采用的是“终止”模型：异常处理器可以找出发生了什么问题，并在外层继续执行，但它不能修复错误的根源并重试失败的操作（除非通过从顶层重新进入出错的代码片段）。

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`.

异常是通过类实例来标识的。*except* 子句会依据实例的类来选择：它必须引用实例的类或是其所属的基类。实例可通过处理器被接收，并可携带有关异常条件的附加信息。

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

---

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

---

另请参看*The try statement* 小节中对*try* 语句的描述以及*The raise statement* 小节中对*raise* 语句的描述。

### 备注

[1] 出现这样的限制是由于通过这些操作执行的代码在模块被编译的时候并不可用。

CHAPTER 5

# 表达式

本章将解释 Python 中组成表达式的各种元素的的含义。

**语法注释:** 在本章和后续章节中，会使用扩展 BNF 标注来描述语法而不是词法分析。当（某种替代的）语法规则具有如下形式

```
name   ::=   othername
```

并且没有给出语义，则这种形式的 `name` 在语法上与 `othername` 相同。

## 5.1 算术转换

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 *Coercion rules*. If both arguments are standard numeric types, the following coercions are applied:

- 如果任一参数为复数，另一参数会被转换为复数；
- 否则，如果任一参数为浮点数，另一参数会被转换为浮点数；
- 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 原子

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

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

### 5.2.1 标识符（名称）

作为原子出现的标识符叫做名称。请参看标识符和关键字 一节了解其词法定义，以及命名与绑定 获取有关命名与绑定的文档。

当名称被绑定到一个对象时，对该原子求值将返回相应对象。当名称未被绑定时，尝试对其求值将引发 NameError 异常。

**私有名称转换:** 当以文本形式出现在类定义中的一个标识符以两个或更多下划线开头并且不以两个或更多下划线结尾，它会被视为该类的 *私有名称*。私有名称会在为其生成代码之前被转换为一种更长的形式。转换时会插入类名，移除打头的下划线再在名称前增加一个下划线。例如，出现在一个名为 Ham 的类中的标识符 __spam 会被转换为 _Ham__spam。这种转换独立于标识符所使用的相关句法。如果转换后的名称太长（超过 255 个字符），可能发生由具体实现定义的截断。如果类名仅由下划线组成，则不会进行转换。

### 5.2.2 字面值

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 字面值 for details.

所有字面值都对应与不可变数据类型，因此对象标识的重要性不如其实际值。多次对具有相同值的字面值求值（不论是发生在程序文本的相同位置还是不同位置）可能得到相同对象或是具有相同值的不同对象。

### 5.2.3 带圆括号的形式

带圆括号的形式是包含在圆括号中的可选表达式列表。

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

带圆括号的表达式列表将返回该表达式列表所产生的任何东西：如果该列表包含至少一个逗号，它会产生一个元组；否则，它会产生该表达式列表所对应的单一表达式。

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).

请注意元组并不是由圆括号构建，实际起作用的是逗号操作符。例外情况是空元组，这时圆括号 才是必须的 —允许在表达式中使用不带圆括号的“空”会导致歧义，并会造成常见输入错误无法被捕获。

### 5.2.4 列表显示

列表显示是一个用方括号括起来的可能为空的表达式系列:

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

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

### 5.2.5 Displays for sets and dictionaries

For constructing a set or a dictionary Python provides special syntax called “displays”, each of them in two flavors:

- 第一种是显式地列出容器内容
- 第二种是通过一组循环和筛选指令计算出来，称为 推导式。

推导式的常用句法元素为:

```
comprehension  ::=   expression comp_for
comp_for       ::=   "for" target_list "in" or_test [comp_iter]
comp_iter      ::=   comp_for | comp_if
comp_if        ::=   "if" expression_nocond [comp_iter]
```

The comprehension 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 container 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 an element each time the innermost block is reached.

Note that the comprehension is executed in a separate scope, so names assigned to in the target list don’t “leak” in the enclosing scope.

[1] In Python 2.3 and later releases, a list comprehension “leaks” the control variables of each `for` it contains into the containing scope. However, this behavior is deprecated, and relying on it will not work in Python 3.

### 5.2.6 生成器表达式

生成器表达式是用圆括号括起来的紧凑形式生成器标注。

```
generator_expression  ::=  "(" expression comp_for ")"
```

生成器表达式会产生一个新的生成器对象。其句法与推导式相同，区别在于它是用圆括号而不是用方括号或花括号括起来的。

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

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

### 5.2.7 字典显示

字典显示是一个用花括号括起来的可能为空的键/数据对系列:

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

字典显示会产生一个新的字典对象。

如果给出一个由逗号分隔的键/数据对序列，它们会从左至右被求值以定义字典的条目：每个键对象会被用作在字典中存放相应数据的键。这意味着你可以在键/数据对序列中多次指定相同的键，最终字典的值将由最后一次给出的键决定。

字典推导式与列表和集合推导式有所不同，它需要以冒号分隔的两个表达式，后面带上标准的 “for” 和 “if” 子句。当推导式被执行时，作为结果的键和值元素会按它们的产生顺序被加入新的字典。

对键取值类型的限制已列在之前的标准类型层级结构 一节中。(总的说来，键的类型应该为*hashable*，这就把所有可变对象都排除在外。) 重复键之间的冲突不会被检测；指定键所保存的最后一个数据 (即在显示中排最右边的文本) 为最终有效数据。

### 5.2.8 集合显示

集合显示是用花括号标明的，与字典显示的区别在于没有冒号分隔的键和值:

```
set_display  ::=  "{" (expression_list | comprehension) "}"
```

集合显示会产生一个新的可变集合对象，其内容通过一系列表达式或一个推导式来指定。当提供由逗号分隔的一系列表达式时，其元素会从左至右被求值并加入到集合对象。当提供一个推导式时，集合会根据推导式所产生的结果元素进行构建。

空集合不能用 `{}` 来构建；该字面值所构建的是一个空字典。

### 5.2.9 String conversions

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

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

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

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

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

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

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

### 5.2.10 yield 表达式

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

2.5 新版功能.

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

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

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

### 生成器-迭代器的方法

这个子小节描述了生成器迭代器的方法。它们可被用于控制生成器函数的执行。

请注意在生成器已经在执行时调用以下任何方法都会引发 ValueError 异常。

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

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

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

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

这里是一个简单的例子，演示了生成器和生成器函数的行为:

```
>>> def echo(value=None):
...     print "Execution starts when 'next()' is called for the first time."
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except Exception, e:
...                 value = e
...     finally:
...         print "Don't forget to clean up when 'close()' is called."
...
>>> generator = echo(1)
>>> print generator.next()
Execution starts when 'next()' is called for the first time.
1
>>> print generator.next()
None
>>> print generator.send(2)
2
>>> generator.throw(TypeError, "spam")
TypeError('spam',)
>>> generator.close()
Don't forget to clean up when 'close()' is called.
```

**参见:**

**PEP 342 - 通过增强型生成器实现协程** 增强生成器 API 和语法的提议，使其可以被用作简单的协程。

## 5.3 原型

原型代表编程语言中最紧密绑定的操作。它们的句法如下:

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

### 5.3.1 属性引用

属性引用是后面带有一个句点加一个名称的原型:

```
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 抽取

抽取就是在序列（字符串、元组或列表）或映射（字典）对象中选择一项:

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

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

如果原型为映射，表达式列表必须求值为一个以该映射的键为值的对象，抽取操作会在映射中选出该键所对应的值。(表达式列表为一个元组，除非其中只有一项。)

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).

字符串的项是字符。字符不是单独的数据类型而是仅有一个字符的字符串。

### 5.3.3 切片

切片就是在序列对象（字符串、元组或列表）中选择某个范围内的项。切片可被用作表达式以及赋值或*del*语句的目标。切片的句法如下:

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

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

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

The semantics for an extended slicing are as follows. The primary must evaluate to a mapping object, and it is indexed with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of an ellipsis slice item is the built-in `Ellipsis` object. The conversion of a proper slice is a slice object (see section 标准类型层级结构) 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 调用

所谓调用就是附带可能为空的一系列参数 来执行一个可调用对象 (例如*function*):

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

```
keyword_arguments      ::=   keyword_item (","  keyword_item)*
keyword_item           ::=   identifier "="  expression
```

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

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

如果存在关键字参数，它们会先通过以下操作被转换为位置参数。首先，为正式参数创建一个未填充空位的列表. 如果有 N 个位置参数，则将它们放入前 N 个空位。然后，对于每个关键字参数，使用标识符来确定其对应的空位（如果标识符与第一个正式参数名相同则使用第一个个空位，依此类推）。如果空位已被填充，则会引发 `TypeError` 异常。否则，将参数值放入空位进行填充（即使表达式为 `None` 也会填充空位）。当所有参数处理完毕时，尚未填充的空位将用来自函数定义的相应默认值来填充。（函数一旦定义其参数默认值就会被计算；因此，当列表或字典这类可变对象被用作默认值时，将会被所有未指定相应空位参数值的调用所共享；这种情况通常应当避免。）如果任何一个未填充空位没有指定默认值，则会引发 `TypeError` 异常。否则的话，已填充空位的列表会被作为调用的参数列表。

某些实现可能提供位置参数没有名称的内置函数，即使它们在文档说明的场合下有“命名”，因此不能以关键字形式提供参数。在 CPython 中，以 C 编写并使用 `PyArg_ParseTuple()` 来解析其参数的函数实现就属于这种情况。

如果存在比正式参数空位多的位置参数，将会引发 `TypeError` 异常，除非有一个正式参数使用了 `*identifier` 句法；在此情况下，该正式参数将接受一个包含了多余位置参数的元组（如果没有多余位置参数则为一个空元组）。

如果任何关键字参数没有与之对应的正式参数名称，将会引发 `TypeError` 异常，除非有一个正式参数使用了 `**identifier` 句法，该正式参数将接受一个包含了多余关键字参数的字典（使用关键字作为键而参数值作为与键对应的值），如果没有多余关键字参数则为一个（新的）空字典。

If the syntax `*expression` appears in the function call, `expression` must evaluate to an iterable. Elements from this iterable are treated as if they were additional positional arguments; if there are positional arguments *x1*, ..., *xN*, and `expression` evaluates to a sequence *y1*, ..., *yM*, this is equivalent to a call with M+N positional arguments *x1*, ..., *xN*, *y1*, ..., *yM*.

A consequence of this is that although the `*expression` syntax may appear *after* some 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 <module>
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
1 2
```

在同一个调用中同时使用关键字参数和 `*expression` 句法并不常见，因此实际上这样的混淆不会发生。

If the syntax `**expression` appears in the function call, `expression` must evaluate to a mapping, 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.

除非引发了异常，调用总是会有返回值，返回值也可能为 None。返回值的计算方式取决于可调用对象的类型。

如果类型为—

**用户自定义函数:** 函数的代码块会被执行，并向其传入参数列表。代码块所做的第一件事是将正式形参绑定到对应参数；相关描述参见*函数定义* 一节。当代码块执行*return* 语句时，由其指定函数调用的返回值。

**内置函数或方法:** 具体结果依赖于解释器；有关内置函数和方法的描述参见 built-in-funcs。

**类对象:** 返回该类的一个新实例。

**类实例方法:** 调用相应的用户自定义函数，向其传入的参数列表会比调用的参数列表多一项：该实例将成为第一个参数。

**类实例:** 该类必须定义有*__call__()* 方法；作用效果将等价于调用该方法。

## 5.4 幂运算符

幂运算符的绑定比在其左侧的一元运算符更紧密；但绑定紧密程度不及在其右侧的一元运算符。句法如下:

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

因此，在一个未加圆括号的幂运算符和单目运算符序列中，运算符将从右向左求值（这不会限制操作数的求值顺序）：-1**2 结果将为 -1。

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

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

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

## 5.5 一元算术和位运算

所有算术和位运算具有相同的优先级:

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

一元运算符 - (负) 会产生其数值参数的负值。

一元运算符 + (正) 会产生与其数值参数相同的值。

The unary ~ (invert) operator yields the bitwise inversion of its plain or long integer argument. The bitwise inversion of

x is defined as -(x+1). It only applies to integral numbers.

在所有三种情况下，如果参数的类型不正确，将引发 TypeError 异常。

## 5.6 二元算术运算符

二元算术运算符遵循传统的优先级。请注意某些此类运算符也作用于特定的非数字类型。除幂运算符以外只有两个优先级别，一个作用于乘法型运算符，另一个作用于加法型运算符:

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

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

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

运算符 % (模) 将输出第一个参数除以第二个参数的余数。两个数字参数将先被转换为相同类型。右参数为零将引发 ZeroDivisionError 异常。参数可以为浮点数，例如 3.14%0.7 等于 0.34 (因为 3.14 等于 4*0.7 + 0.34)。模运算符的结果的正负总是与第二个操作数一致（或是为零）；结果的绝对值一定小于第二个操作数的绝对值[2]。

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

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

2.3 版后已移除: The floor division operator, the modulo operator, and the divmod() function are no longer defined for complex numbers. Instead, convert to a floating point number using the abs() function if appropriate.

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

运算符 - (减) 将输出其参数的差。两个数字参数将先被转换为相同类型。

[2] 虽然 abs(x%y) < abs(y) 在数学中必为真，但对于浮点数而言，由于舍入的存在，其在数值上未必为真。例如，假设在某个平台上的 Python 浮点数为一个 IEEE 754 双精度数值，为了使 -1e-100 % 1e100 具有与 1e100 相同的正负性，计算结果将是 -1e-100 + 1e100，这在数值上正好等于 1e100。函数 math.fmod() 返回的结果则会具有与第一个参数相同的正负性，因此在这种情况下将返回 -1e-100。何种方式更适宜取决于具体的应用。

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

## 5.7 移位运算

移位运算的优先级低于算术运算:

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

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

A right shift by *n* bits is defined as division by `pow(2, n)`. A left shift by *n* bits is defined as multiplication with `pow(2, n)`. Negative shift counts raise a `ValueError` exception.

---

**注解:** In the current implementation, the right-hand operand is required to be at most `sys.maxsize`. If the right-hand operand is larger than `sys.maxsize` an `OverflowError` exception is raised.

---

## 5.8 二元位运算

三种位运算具有各不相同的优先级:

```
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 比较运算

与 C 不同，Python 中所有比较运算的优先级相同，低于任何算术、移位或位运算。另一个与 C 不同之处在于 `a < b < c` 这样的表达式会按传统算术法则来解读:

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

比较运算将输出布尔值: `True` 或 `False`。

比较运算可以任意串连，例如 `x < y <= z` 等价于 `x < y and y <= z`，除了 `y` 只被求值一次（但在两种写法下当 `x < y` 值为假时 `z` 都不会被求值）。

正式的说法是这样：如果 *a*, *b*, *c*, ..., *y*, *z* 为表达式而 *op1*, *op2*, ..., *opN* 为比较运算符，则 `a op1 b op2 c`

`... y opN z` 就等价于 `a op1 b and b op2 c and ... y opN z`，后者的不同之处只是每个表达式最多只被求值一次。

请注意 `a op1 b op2 c` 不意味着在 *a* 和 *c* 之间进行任何比较，因此，如 `x < y > z` 这样的写法是完全合法的（虽然也许不太好看）。

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

### 5.9.1 值比较

运算符 <, >, ==, >=, <= 和 != 将比较两个对象的值。两个对象不要求为相同类型。

对象、值与类型 一章已说明对象都有相应的值（还有类型和标识号）。对象值在 Python 中是一个相当抽象的概念：例如，对象值并没有一个规范的访问方法。而且，对象值并不要求具有特定的构建方式，例如由其全部数据属性组成等。比较运算符实现了一个特定的对象值概念。人们可以认为这是通过实现对象比较间接地定义了对象值。

Types can customize their comparison behavior by implementing a *__cmp__()* method or *rich comparison methods* like *__lt__()*, described in 基本定制.

默认的一致性比较 (== 和 !=) 是基于对象的标识号。因此，具有相同标识号的实例一致性比较结果为相等，具有不同标识号的实例一致性比较结果为不等。规定这种默认行为的动机是希望所有对象都应该是自反射的 (即 `x is y` 就意味着 `x == y`)。

The default order comparison (<, >, <=, and >=) gives a consistent but arbitrary order.

(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.)

按照默认的一致性比较行为，具有不同标识号的实例总是不相等，这可能不适合某些对象值需要有合理定义并有基于值的一致性的类型。这样的类型需要定制自己的比较行为，实际上，许多内置类型都是这样做的。

以下列表描述了最主要内置类型的比较行为。

- 内置数值类型 (typesnumeric) 以及标准库类型 `fractions.Fraction` 和 `decimal.Decimal` 可进行类型内部和跨类型的比较，例外限制是复数不支持次序比较。在类型相关的限制以内，它们会按数学（算法）规则正确进行比较且不会有精度损失。
- Strings (instances of `str` or `unicode`) compare lexicographically using the numeric equivalents (the result of the built-in function `ord()`) of their characters.[4] When comparing an 8-bit string and a Unicode string, the 8-bit string is converted to Unicode. If the conversion fails, the strings are considered unequal.
- Instances of `tuple` or `list` can be compared only within each of their types. Equality comparison across these types results in unequality, and ordering comparison across these types gives an arbitrary order.

  These sequences compare lexicographically using comparison of corresponding elements, whereby reflexivity of the elements is enforced.

  In enforcing reflexivity of elements, the comparison of collections assumes that for a collection element `x`, `x == x` is always true. Based on that assumption, element identity is compared first, and element comparison is performed only for distinct elements. This approach yields the same result as a strict element comparison would,

[4] Unicode 标准明确区分 码位 (例如 U+0041) 和 抽象字符 (例如 “大写拉丁字母 A”)。虽然 Unicode 中的大多数抽象字符都只用一个码位来代表，但也存在一些抽象字符可使用由多个码位组成的序列来表示。例如，抽象字符 “带有下加符的大写拉丁字母 C” 可以用 U+00C7 码位上的单个 预设字符来表示，也可以用一个 U+0043 码位上的 基础字符 (大写拉丁字母 C) 加上一个 U+0327 码位上的 组合字符 (组合下加符) 组成的序列来表示。

The comparison operators on unicode strings compare at the level of Unicode code points. This may be counter-intuitive to humans. For example, `u"\u00C7" == u"\u0043\u0327"` is `False`, even though both strings represent the same abstract character “LATIN CAPITAL LETTER C WITH CEDILLA”.

要按抽象字符级别（即对人类来说更直观的方式）对字符串进行比较，应使用 `unicodedata.normalize()`。

if the compared elements are reflexive. For non-reflexive elements, the result is different than for strict element comparison.

内置多项集间的字典序比较规则如下:

- 两个多项集若要相等，它们必须为相同类型、相同长度，并且每对相应的元素都必须相等（例如，`[1,2] == (1,2)` 为假值，因为类型不同）。
- Collections are ordered the same as their first unequal elements (for example, `cmp([1,2,x], [1,2,y])` returns the same as `cmp(x,y)`). If a corresponding element does not exist, the shorter collection is ordered first (for example, `[1,2] < [1,2,3]` is true).

- 两个映射 (`dict` 的实例) 若要相等，必须当且仅当它们具有相同的 *(键, 值)* 对。键和值的一致性比较强制规定自反射性。

  Outcomes other than equality are resolved consistently, but are not otherwise defined.[5]

- Most other objects of built-in 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.

在可能的情况下，用户定义类在定制其比较行为时应当遵循一些一致性规则:

- 相等比较应该是自反射的。换句话说，相同的对象比较时应该相等:

  `x is y` 意味着 `x == y`

- 比较应该是对称的。换句话说，下列表达式应该有相同的结果:

  `x == y` 和 `y == x`

  `x != y` 和 `y != x`

  `x < y` 和 `y > x`

  `x <= y` 和 `y >= x`

- 比较应该是可传递的。下列（简要的）例子显示了这一点:

  `x > y and y > z` 意味着 `x > z`

  `x < y and y <= z` 意味着 `x < z`

- 反向比较应该导致布尔值取反。换句话说，下列表达式应该有相同的结果:

  `x == y` 和 `not x != y`

  `x < y` 和 `not x >= y` (对于完全排序)

  `x > y` 和 `not x <= y` (对于完全排序)

  最后两个表达式适用于完全排序的多项集（即序列而非集合或映射）。另请参阅 `total_ordering()` 装饰器。

- `hash()` 的结果应该与是否相等一致。相等的对象应该或者具有相同的哈希值，或者标记为不可哈希。

Python does not enforce these consistency rules.

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

### 5.9.2 成员检测运算

The operators *in* and *not in* test for membership. `x in s` evaluates to `True` if *x* is a member of *s*, and `False` otherwise. `x not in s` returns the negation of `x in s`. All built-in sequences and set types support this as well as dictionary, for which *in* tests whether the dictionary has a given key. For container types such as list, tuple, set, frozenset, dict, or collections.deque, the expression `x in y` is equivalent to `any(x is e or x == e for e in y)`.

对于字符串和字节串类型来说，当且仅当 *x* 是 *y* 的子串时 `x in y` 为 `True`。一个等价的检测是 `y.find(x) != -1`。空字符串总是被视为任何其他字符串的子串，因此 `"" in "abc"` 将返回 `True`。

对于定义了*__contains__()* 方法的用户自定义类来说，如果 `y.__contains__(x)` 返回真值则 `x in y` 返回 `True`，否则返回 `False`。

For user-defined classes which do not define *__contains__()* but do define *__iter__()*, `x in y` is `True` if some value `z` with `x == z` is produced while iterating over `y`. If an exception is raised during the iteration, it is as if *in* raised that exception.

Lastly, the old-style iteration protocol is tried: if a class defines *__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*.

### 5.9.3 标识号比较

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.[6]

## 5.10 布尔运算

```
or_test   ::=   and_test | or_test "or" and_test
and_test  ::=   not_test | and_test "and" not_test
not_test  ::=   comparison | "not" not_test
```

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

运算符*not* 将在其参数为假值时产生 `True`，否则产生 `False`。

表达式 `x and y` 首先对 *x* 求值；如果 *x* 为假则返回该值；否则对 *y* 求值并返回其结果值。

表达式 `x or y` 首先对 *x* 求值；如果 *x* 为真则返回该值；否则对 *y* 求值并返回其结果值。

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

[6] 由于存在自动垃圾收集、空闲列表以及描述器的动态特性，你可能会注意到在特定情况下使用*is* 运算符会出现看似不正常的行为，例如涉及到实例方法或常量之间的比较时就是如此。更多信息请查看有关它们的文档。

## 5.11 Conditional Expressions

2.5 新版功能.

```
conditional_expression  ::=  or_test ["if" or_test "else" expression]
expression              ::=  conditional_expression | lambda_expr
```

条件表达式（有时称为“三元运算符”）在所有 Python 运算中具有最低的优先级。

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

请参阅 **PEP 308** 了解有关条件表达式的详情。

## 5.12 lambda 表达式

```
lambda_expr      ::=  "lambda" [parameter_list]: expression
old_lambda_expr  ::=  "lambda" [parameter_list]: old_expression
```

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

```
def <lambda>(parameters):
    return expression
```

See section 函数定义 for the syntax of parameter lists. Note that functions created with lambda expressions cannot contain statements.

## 5.13 表达式列表

```
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.

末尾的逗号仅在创建单独元组 (或称 *单例*) 时需要；在所有其他情况下都是可选项。没有末尾逗号的单独表达式不会创建一个元组，而是产生该表达式的值。(要创建一个空元组，应使用一对内容为空的圆括号: `()`。)

## 5.14 求值顺序

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

在以下几行中，表达式将按其后缀的算术优先顺序被求值。:

```
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
expr1(expr2, expr3, *expr4, **expr5)
expr3, expr4 = expr1, expr2
```

## 5.15 运算符优先级

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

| 运算符 | 描述 |
|---|---|
| *lambda* | lambda 表达式 |
| *if –else* | 条件表达式 |
| *or* | 布尔逻辑或 OR |
| *and* | 布尔逻辑与 AND |
| *not* x | 布尔逻辑非 NOT |
| *in*, *not in*, *is*, *is not*, <, <=, >, >=, <>, !=, == | 比较运算，包括成员检测和标识号检测 |
| \| | 按位或 OR |
| ^ | 按位异或 XOR |
| & | 按位与 AND |
| <<, >> | 移位 |
| +, – | 加和减 |
| *, /, //, % | Multiplication, division, remainder[7] |
| +x, –x, ~x | 正，负，按位非 NOT |
| ** | 乘方[8] |
| x[index], x[index:index], x(arguments...), x.attribute | 抽取，切片，调用，属性引用 |
| (expressions...), [expressions...], {key: value...}, `expressions...` | Binding or tuple display, list display, dictionary display, string conversion |

[7] % 运算符也被用于字符串格式化；在此场合下会使用同样的优先级。
[8] 幂运算符 ** 绑定的紧密程度低于在其右侧的算术或按位一元运算符，也就是说 2**-1 为 0.5。

### 备注

CHAPTER 6

# 简单语句

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

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

## 6.1 表达式语句

表达式语句用于计算和写入值（大多是在交互模式下），或者（通常情况）调用一个过程 (过程就是不返回有意义结果的函数；在 Python 中，过程的返回值为 `None`)。表达式语句的其他使用方式也是允许且有特定用处的。表达式语句的句法为:

```
expression_stmt  ::=  expression_list
```

表达式语句会对指定的表达式列表（也可能为单一表达式）进行求值。

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 *The print statement*) on a line by itself. (Expression statements yielding `None` are not written, so that procedure calls do not cause any output.)

## 6.2 赋值语句

赋值语句用于将名称（重）绑定到特定值，以及修改属性或可变对象的成员项:

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

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

赋值语句会对指定的表达式列表进行求值（注意这可能为单一表达式或是由逗号分隔的列表，后者将产生一个元组）并将单一结果对象从左至右逐个赋值给目标列表。

赋值是根据目标（列表）的格式递归地定义的。当目标为一个可变对象（属性引用、抽取或切片）的组成部分时，该可变对象必须最终执行赋值并决定其有效性，如果赋值操作不可接受也可能引发异常。各种类型可用的规则和引发的异常通过对象类型的定义给出（参见标准类型层级结构 一节）。

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 an iterable 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.

对象赋值给单个目标的操作按以下方式递归地定义。

- 如果目标为标识符（名称）：

  - 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.

  如果该名称已经被绑定则将被重新绑定。这可能导致之前被绑定到该名称的对象的引用计数变为零，造成该对象进入释放过程并调用其析构器（如果存在）。

- If the target is a target list enclosed in parentheses or in square brackets: The object must be an iterable 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.
- 如果该对象为属性引用：引用中的原型表达式会被求值。它应该产生一个具有可赋值属性的对象；否则将引发 `TypeError`。该对象会被要求将可赋值对象赋值给指定的属性；如果它无法执行赋值，则会引发异常 (通常应为 `AttributeError` 但并不强制要求)。

注意：如果该对象为类实例并且属性引用在赋值运算符的两侧都出现，则右侧表达式 `a.x` 可以访问实例属性或（如果实例属性不存在）类属性。左侧目标 `a.x` 将总是设定为实例属性，并在必要时创建该实例属性。因此，`a.x` 的两次出现不一定指向相同的属性：如果右侧表达式指向一个类属性，则左侧表达式会创建一个新的实例属性作为赋值的目标:

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

此描述不一定作用于描述器属性，例如通过 `property()` 创建的特征属性。

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

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

  如果原型为一个映射对象（例如字典），抽取必须具有与该映射的键类型相兼容的类型，然后映射中会创建一个将抽取映射到被赋值对象的键/值对。这可以是替换一个现有键/值对并保持相同键值，也可以是插入一个新键/值对（如果具有相同值的键不存在）。

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

在当前实现中，目标的句法被当作与表达式的句法相同，无效的句法会在代码生成阶段被拒绝，导致不太详细的错误信息。

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

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

### 6.2.1 增强赋值语句

增强赋值语句就是在单个语句中将二元运算和赋值语句合为一体:

```
augmented_assignment_stmt  ::=  augtarget augop (expression_list | yield_expression)
augtarget                  ::=  identifier | attributeref | subscription | slicing
augop                      ::=  "+=" | "-=" | "*=" | "/=" | "//=" | "%=" | "**="
                                | ">>=" | "<<=" | "&=" | "^=" | "|="
```

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

增强赋值语句将对目标和表达式列表求值（与普通赋值语句不同的是，前者不能为可迭代对象拆包），对两个操作数相应类型的赋值执行指定的二元运算，并将结果赋值给原始目标。目标仅会被求值一次。

增强赋值语句例如 `x += 1` 可以改写为 `x = x + 1` 获得类似但并非完全等价的效果。在增强赋值的版本中，`x` 仅会被求值一次。而且，在可能的情况下，实际的运算是 *原地* 执行的，也就是说并不是创建一个新对象并将其赋值给目标，而是直接修改原对象。

除了在单个语句中赋值给元组和多个目标的例外情况，增强赋值语句的赋值操作处理方式与普通赋值相同。类似地，除了可能存在 *原地* 操作行为的例外情况，增强赋值语句执行的二元运算也与普通二元运算相同。

对于属性引用类目标，针对常规赋值的关于类和实例属性的警告 也同样适用。

## 6.3 The `assert` statement

assert 语句是在程序中插入调试性断言的简便方式:

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

简单形式 `assert expression` 等价于

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

扩展形式 `assert expression1, expression2` 等价于

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

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

赋值给 `__debug__` 是非法的。该内置变量的值会在解释器启动时确定。

## 6.4 The `pass` statement

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

*pass* 是一个空操作—当它被执行时，什么都不发生。它适合当语法上需要一条语句但并不需要执行任何代码时用来临时占位，例如:

```
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
```

删除是递归定义的，与赋值的定义方式非常类似。此处不再详细说明，只给出一些提示。

目标列表的删除将从左至右递归地删除每一个目标。

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

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

属性引用、抽取和切片的删除会被传递给相应的原型对象；删除一个切片基本等价于赋值为一个右侧类型的空切片（但即便这一点也是由切片对象决定的）。

## 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 a whitespace character except `' '`, 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.)

---

**注解：** 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* 在语法上只会出现于函数定义所嵌套的代码，不会出现于类定义所嵌套的代码。

如果提供了表达式列表，它将被求值，否则以 None 替代。

*return* 会离开当前函数调用，并以表达式列表 (或 None) 作为返回值。

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

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

## 6.8 The `yield` statement

```
yield_stmt   ::=   yield_expression
```

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

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

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

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

有关*yield* 语义的完整细节请参看*yield 表达式* 一节。

---

**注解：** In Python 2.2, the *yield* statement was only allowed when the generators feature has been enabled. This __future__ import statement was used to enable the feature:

```
from __future__ import generators
```

---

**参见：**

**PEP 255 - Simple Generators** The proposal for adding generators and the *yield* statement to Python.

**PEP 342 - Coroutines via Enhanced Generators** The proposal that, among other generator enhancements, proposed allowing *yield* to appear inside a *try* ··· *finally* block.

## 6.9 The `raise` statement

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

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

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

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

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

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

有关异常的更多信息可在异常 一节查看，有关处理异常的信息可在*The try statement* 一节查看。

## 6.10 The `break` statement

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

*break* 在语法上只会出现于*for* 或*while* 循环所嵌套的代码，但不会出现于该循环内部的函数或类定义所嵌套的代码。

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

如果一个*for* 循环被*break* 所终结，该循环的控制目标会保持其当前值。

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* 在语法上只会出现于*for* 或*while* 循环所嵌套的代码，但不会出现于该循环内部的函数或类定义或者*finally* 子句所嵌套的代码。它会继续执行最近的外层循环的下一个轮次。

When *continue* passes control out of a *try* statement with a *finally* clause, that *finally* clause is executed before really starting the next loop cycle.

## 6.12 The `import` statement

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

Import statements are executed in two steps: (1) find a module, and initialize it if necessary; (2) define a name or names in the local namespace (of the scope where the *import* statement occurs). The statement comes in two forms differing on whether it uses the *from* keyword. 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.

To understand how step (1) occurs, one must first understand how Python handles hierarchical naming of modules. To help organize modules and provide a hierarchy in naming, Python has a concept of packages. A package can contain other packages and modules while modules cannot contain other modules or packages. From a file system perspective, packages are directories and modules are files.

Once the name of the module is known (unless otherwise specified, the term "module" will refer to both packages and modules), searching for the module or package can begin. The first place checked is `sys.modules`, the cache of all modules that have been imported previously. If the module is found there then it is used in step (2) of import.

If the module is not found in the cache, then `sys.meta_path` is searched (the specification for `sys.meta_path` can be found in **PEP 302**). The object is a list of *finder* objects which are queried in order as to whether they know how to load the module by calling their `find_module()` method with the name of the module. If the module happens to be contained within a package (as denoted by the existence of a dot in the name), then a second argument to `find_module()` is given as the value of the `__path__` attribute from the parent package (everything up to the last dot in the name of the module being imported). If a finder can find the module it returns a *loader* (discussed later) or returns `None`.

If none of the finders on `sys.meta_path` are able to find the module then some implicitly defined finders are queried. Implementations of Python vary in what implicit meta path finders are defined. The one they all do define, though, is one that handles `sys.path_hooks`, `sys.path_importer_cache`, and `sys.path`.

The implicit finder searches for the requested module in the "paths" specified in one of two places ("paths" do not have to be file system paths). If the module being imported is supposed to be contained within a package then the second argument passed to `find_module()`, `__path__` on the parent package, is used as the source of paths. If the module is not contained in a package then `sys.path` is used as the source of paths.

Once the source of paths is chosen it is iterated over to find a finder that can handle that path. The dict at `sys.path_importer_cache` caches finders for paths and is checked for a finder. If the path does not have a finder cached then `sys.path_hooks` is searched by calling each object in the list with a single argument of the path, returning a finder or raises `ImportError`. If a finder is returned then it is cached in `sys.path_importer_cache` and then used for that path entry. If no finder can be found but the path exists then a value of `None` is stored in `sys.path_importer_cache` to signify that an implicit, file-based finder that handles modules stored as individual files should be used for that path. If the path does not exist then a finder which always returns `None` is placed in the cache for the path.

If no finder can find the module then `ImportError` is raised. Otherwise some finder returned a loader whose `load_module()` method is called with the name of the module to load (see **PEP 302** for the original definition of loaders). A loader has several responsibilities to perform on a module it loads. First, if the module already exists in `sys.modules` (a possibility if the loader is called outside of the import machinery) then it is to use that module for

initialization and not a new module. But if the module does not exist in `sys.modules` then it is to be added to that dict before initialization begins. If an error occurs during loading of the module and it was added to `sys.modules` it is to be removed from the dict. If an error occurs but the module was already in `sys.modules` it is left in the dict.

The loader must set several attributes on the module. `__name__` is to be set to the name of the module. `__file__` is to be the "path" to the file unless the module is built-in (and thus listed in `sys.builtin_module_names`) in which case the attribute is not set. If what is being imported is a package then `__path__` is to be set to a list of paths to be searched when looking for modules and packages contained within the package being imported. `__package__` is optional but should be set to the name of package that contains the module or package (the empty string is used for module not contained in a package). `__loader__` is also optional but should be set to the loader object that is loading the module.

If an error occurs during loading then the loader raises `ImportError` if some other exception is not already being propagated. Otherwise the loader returns the module that was loaded and initialized.

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

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

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

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

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

When specifying what module to import you do not have to specify the absolute name of the module. When a module or package is contained within another package it is possible to make a relative import within the same top package without having to mention the package name. By using leading dots in the specified module or package after *from* you can specify how high to traverse up the current package hierarchy without specifying exact names. One leading dot means the current package where the module making the import exists. Two dots means up one package level. Three dots is up two levels, etc. So if you execute `from . import mod` from a module in the `pkg` package then you will end up importing `pkg.mod`. If you execute `from ..subpkg2 import mod` from within `pkg.subpkg1` you will import `pkg.subpkg2.mod`. The specification for relative imports is contained within **PEP 328**.

`importlib.import_module()` is provided to support applications that determine which modules need to be loaded dynamically.

### 6.12.1 future 语句

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

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

future 语句必须在靠近模块开头的位置出现。可以出现在 future 语句之前行只有:

- 模块的文档字符串（如果存在），
- 注释，
- 空行，以及
- 其他 future 语句。

The features recognized by Python 2.6 are `unicode_literals`, `print_function`, `absolute_import`, `division`, `generators`, `nested_scopes` and `with_statement`. `generators`, `with_statement`, `nested_scopes` are redundant in Python version 2.6 and above because they are always enabled.

future 语句在编译时会被识别并做特殊对待：对核心构造语义的改变常常是通过生成不同的代码来实现。新的特性甚至可能会引入新的不兼容语法（例如新的保留字），在这种情况下编译器可能需要以不同的方式来解析模块。这样的决定不能推迟到运行时方才作出。

对于任何给定的发布版本，编译器要知道哪些特性名称已被定义，如果某个 future 语句包含未知的特性则会引发编译时错误。

直接运行时的语义与任何 import 语句相同：存在一个后文将详细说明的标准模块 `__future__`，它会在执行 future 语句时以通常的方式被导入。

相应的运行时语义取决于 future 语句所启用的指定特性。

请注意以下语句没有任何特别之处:

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

这并非 future 语句；它只是一条没有特殊语义或语法限制的普通 import 语句。

Code compiled by an *exec* statement or calls to the built-in 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 for details.

在交互式解释器提示符中键入的 future 语句将在解释器会话此后的交互中有效。如果一个解释器的启动使用了 `-i` 选项启动，并传入了一个脚本名称来执行，且该脚本包含 future 语句，它将在交互式会话开始执行脚本之后保持有效。

**参见:**

**PEP 236 - 回到 __future__** 有关 __future__ 机制的最初提议。

## 6.13 The global statement

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

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

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

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

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

## 6.14 The exec statement

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

This statement supports dynamic execution of Python code. The first expression should evaluate to either a Unicode string, a *Latin-1* encoded string, an open file object, a code object, or a tuple. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs).[1] If it is an open file, the file is parsed until EOF and executed. If it is a code object, it is simply executed. For the interpretation of a tuple, see below. In all cases, the code that's executed is expected to be valid as file input (see section 文件输入). Be aware that the *return* and *yield* statements may not be used outside of function definitions even within the context of code passed to the *exec* statement.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only the first expression after `in` is specified, it should be a dictionary, which will be used for both the global and the local variables. If two expressions are given, they are used for the global and local variables, respectively. If provided, *locals* can be any mapping object. Remember that at module level, globals and locals are the same dictionary. If two separate objects are given as *globals* and *locals*, the code will be executed as if it were embedded in a class definition.

The first expression may also be a tuple of length 2 or 3. In this case, the optional parts must be omitted. The form `exec(expr, globals)` is equivalent to `exec expr in globals`, while the form `exec(expr, globals, locals)` is equivalent to `exec expr in globals, locals`. The tuple form of `exec` provides compatibility with Python 3, where `exec` is a function rather than a statement.

在 2.4 版更改: Formerly, *locals* was required to be a dictionary.

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

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

[1] Note that the parser only accepts the Unix-style end of line convention. If you are reading the code from a file, make sure to use *universal newlines* mode to convert Windows or Mac-style newlines.

CHAPTER 7

# 复合语句

复合语句是包含其它语句（语句组）的语句；它们会以某种方式影响或控制所包含其它语句的执行。通常，复合语句会跨越多行，虽然在某些简单形式下整个复合语句也可能包含于一行之内。

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
```

总结:

```
compound_stmt   ::=   if_stmt
                    | while_stmt
                    | for_stmt
                    | try_stmt
                    | with_stmt
                    | funcdef
                    | classdef
                    | decorated
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).

为了保证清晰，以下各节中语法规则采用将每个子句都放在单独行中的格式。

## 7.1 The `if` statement

*if* 语句用于有条件的执行:

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

它通过对表达式逐个求值直至找到一个真值（请参阅布尔运算 了解真值与假值的定义）在子句体中选择唯一匹配的一个；然后执行该子句体（而且*if* 语句的其他部分不会被执行或求值）。如果所有表达式均为假值，则如果*else* 子句体如果存在就会被执行。

## 7.2 The `while` statement

*while* 语句用于在表达式保持为真的情况下重复地执行:

```
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

*for* 语句用于对序列（例如字符串、元组或列表）或其他可迭代对象中的元素进行迭代:

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

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

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

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

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

---

**注解:** There is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, e.g. 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

*try* 语句可为一组语句指定异常处理器和/或清理代码:

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

在 2.5 版更改: In previous versions of Python, *try*…*except*…*finally* did not work. *try*…*except* had to be nested in *try*…*finally*.

The *except* clause(s) specify one or more exception handlers. When no exception occurs in the *try* clause, no exception handler is executed. When an exception occurs in the *try* suite, a search for an exception handler is started. This search inspects the except clauses in turn until one is found that matches the exception. An expression-less except clause, if present, must be last; it matches any exception. For an except clause with an expression, that expression is evaluated, and the clause matches the exception if the resulting object is "compatible" with the exception. An object is compatible with an exception if it is the class or a base class of the exception object, or a tuple containing an item compatible with the exception.

如果没有 except 子句与异常相匹配，则会在周边代码和发起调用栈上继续搜索异常处理器。[1]

如果在对 except 子句头中的表达式求值时引发了异常，则原来对处理器的搜索会被取消，并在周边代码和调用栈上启动对新异常的搜索（它会被视作是整个*try* 语句所引发的异常）。

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

[1] 异常会被传播给发起调用栈，除非存在一个*finally* 子句正好引发了另一个异常。新引发的异常将导致旧异常的丢失。

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 标准类型层级结构) 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 the control flow leaves the *try* suite, no exception was raised, and no *return*, *continue*, or *break* statement was executed. Exceptions in the *else* clause are not handled by the preceding *except* clauses.

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

```
>>> def f():
...     try:
...         1/0
...     finally:
...         return 42
...
>>> f()
42
```

在*finally* 子句执行期间，程序不能获取异常信息。

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).

The return value of a function is determined by the last *return* statement executed. Since the *finally* clause always executes, a *return* statement executed in the *finally* clause will always be the last one executed:

```
>>> def foo():
...     try:
...         return 'try'
...     finally:
...         return 'finally'
...
>>> foo()
'finally'
```

有关异常的更多信息可以在异常 一节找到，有关使用*raise* 语句生成异常的信息可以在*The raise statement* 一节找到。

## 7.5 The `with` statement

2.5 新版功能.

*with* 语句用于包装带有使用上下文管理器 (参见*with* 语句上下文管理器 一节) 定义的方法的代码块的执行。这允许对普通的*try*…*except*…*finally* 使用模式进行封装以方便地重用。

```
with_stmt  ::=  "with" with_item (","  with_item)* ":"  suite
```

```
with_item  ::=  expression ["as" target]
```

带有一个“项目”的*with*语句的执行过程如下:

1. 对上下文表达式 (在 *with_item* 中给出的表达式) 求值以获得一个上下文管理器。
2. 载入上下文管理器的*__exit__()* 以便后续使用。
3. 发起调用上下文管理器的*__enter__()* 方法。
4. 如果*with*语句中包含一个目标，来自*__enter__()* 的返回值将被赋值给它。

   **注解:** *with*语句会保证如果*__enter__()* 方法返回时未发生错误，则*__exit__()* 将总是被调用。因此，如果在对目标列表赋值期间发生错误，则会将其视为在语句体内部发生的错误。参见下面的第 6 步。

5. 执行语句体。
6. The context manager’s *__exit__()* method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to *__exit__()*. Otherwise, three `None` arguments are supplied.

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

   如果语句体由于异常以外的任何原因退出，则来自*__exit__()* 的返回值会被忽略，并会在该类退出正常的发生位置继续执行。

如果有多个项目，则会视作存在多个*with*语句嵌套来处理多个上下文管理器:

```
with A() as a, B() as b:
    suite
```

等价于

```
with A() as a:
    with B() as b:
        suite
```

**注解:** In Python 2.5, the *with* statement is only allowed when the `with_statement` feature has been enabled. It is always enabled in Python 2.6.

在 2.7 版更改: 支持多个上下文表达式。

**参见:**

**PEP 343 - “with”语句** Python *with*语句的规范描述、背景和示例。

## 7.6 函数定义

函数定义就是对用户自定义函数的定义（参见标准类型层级结构 一节）：

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

函数定义是一条可执行语句。它执行时会在当前局部命名空间中将函数名称绑定到一个函数对象（函数可执行代码的包装器）。这个函数对象包含对当前全局命名空间的引用，作为函数被调用时所使用的全局命名空间。

函数定义并不会执行函数体；只有当函数被调用时才会执行此操作。[2]

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

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

is equivalent to:

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

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

**Default parameter values are evaluated when the function definition is executed.** This means that the expression is evaluated once, when the function is defined, and that the 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 = []
```

(下页继续)

[2] 作为函数体的第一条语句出现的字符串字面值会被转换为函数的 __doc__ 属性，也就是该函数的*docstring*。

(续上页)

```
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section 调用. 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 expressions, described in section *lambda* 表达式. Note that the lambda expression 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 expression. The "*def*" form is actually more powerful since it allows the execution of multiple statements.

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

## 7.7 类定义

类定义就是对类对象的定义 (参见标准类型层级结构 一节):

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

A class definition is an executable statement. It first evaluates the inheritance list, if present. Each item in the inheritance list should evaluate to a class object or class type which allows subclassing. The class's suite is then executed in a new execution frame (see section 命名与绑定), 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.[3] 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 create instance variables, they can be set in a method with self.name = value. 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 can be used as defaults for instance variables, but using mutable values there can lead to unexpected results. For *new-style class*es, descriptors can be used to create instance variables with different implementation details.

Class definitions, like function definitions, may be wrapped by one or more *decorator* expressions. The evaluation rules for the decorator expressions are the same as for functions. The result must be a class object, which is then bound to the class name.

[3] 作为类体的第一条语句出现的字符串字面值会被转换为命名空间的 __doc__ 条目，也就是该类的*docstring*。

### 备注

CHAPTER 8

# 最高层级组件

Python 解释器可以从多种源获得输入：作为标准输入或程序参数传入的脚本，以交互方式键入的语句，导入的模块源文件等等。这一章将给出在这些情况下所用的语法。

## 8.1 完整的 Python 程序

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.

适用于一个完整 Python 程序的语法即下节所描述的文件输入。

解释器也可以通过交互模式被发起调用；在此情况下，它并不读取和执行一个完整程序，而是每次读取和执行一条语句（可能为复合语句）。此时的初始环境与一个完整程序的相同；每条语句会在 `__main__` 的命名空间中被执行。

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 文件输入

所有从非交互式文件读取的输入都具有相同的形式:

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

此语法用于下列几种情况:

- 解析一个完整 Python 程序时（从文件或字符串）；
- 解析一个模块时；
- when parsing a string passed to the *exec* statement;

## 8.3 交互式输入

交互模式下的输入使用以下语法进行解析:

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

请注意在交互模式下一条（最高层级）复合语句必须带有一个空行；这对于帮助解析器确定输入的结束是必须的。

## 8.4 表达式输入

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.

CHAPTER 9

# 完整的语法规范

这是完整的 Python 语法，它被送入解析器生成器，以生成解析 Python 源文件的解析器：

```
# Grammar for Python

# Note:  Changing the grammar specified in this file will most likely
#        require corresponding changes in the parser module
#        (../Modules/parsermodule.c).  If you can't make the changes to
#        that module yourself, please co-ordinate the required changes
#        with someone who can; ask around on python-dev for help.  Fred
#        Drake <fdrake@acm.org> will probably be listening there.

# NOTE WELL: You should also follow all the steps listed in PEP 306,
# "How to Change Python's Grammar"

# Start symbols for the grammar:
#       single_input is a single interactive statement;
#       file_input is a module or sequence of commands read from an input file;
#       eval_input is the input for the eval() and input() functions.
# NB: compound_stmt in single_input is followed by extra NEWLINE!
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER

decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
decorators: decorator+
decorated: decorators (classdef | funcdef)
funcdef: 'def' NAME parameters ':' suite
parameters: '(' [varargslist] ')'
varargslist: ((fpdef ['=' test] ',')*
              ('*' NAME [',' '**' NAME] | '**' NAME) |
              fpdef ['=' test] (',' fpdef ['=' test])* [','])
fpdef: NAME | '(' fplist ')'
fplist: fpdef (',' fpdef)* [',']
```

(下页继续)

(续上页)

```
stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | print_stmt  | del_stmt | pass_stmt | flow_stmt |
             import_stmt | global_stmt | exec_stmt | assert_stmt)
expr_stmt: testlist (augassign (yield_expr|testlist) |
                     ('=' (yield_expr|testlist))*)
augassign: ('+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' |
            '<<=' | '>>=' | '**=' | '//=')
# For normal assignments, additional restrictions enforced by the interpreter
print_stmt: 'print' ( [ test (',' test)* [','] ] |
                      '>>' test [ (',' test)+ [','] ] )
del_stmt: 'del' exprlist
pass_stmt: 'pass'
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist]
yield_stmt: yield_expr
raise_stmt: 'raise' [test [',' test [',' test]]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
import_from: ('from' ('.'* dotted_name | '.'+)
              'import' ('*' | '(' import_as_names ')' | import_as_names))
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)* [',']
dotted_as_names: dotted_as_name (',' dotted_as_name)*
dotted_name: NAME ('.' NAME)*
global_stmt: 'global' NAME (',' NAME)*
exec_stmt: 'exec' expr ['in' test [',' test]]
assert_stmt: 'assert' test [',' test]

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef |␣
↪classdef | decorated
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
try_stmt: ('try' ':' suite
           ((except_clause ':' suite)+
            ['else' ':' suite]
            ['finally' ':' suite] |
           'finally' ':' suite))
with_stmt: 'with' with_item (',' with_item)*  ':' suite
with_item: test ['as' expr]
# NB compile.c makes sure that the default except clause is last
except_clause: 'except' [test [('as' | ',') test]]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT

# Backward compatibility cruft to support:
# [ x for x in lambda: True, lambda: False if x() ]
# even while also allowing:
# lambda x: 5 if x else 2
# (But not a mix of the two)
testlist_safe: old_test [(',' old_test)+ [',']]
old_test: or_test | old_lambdef
old_lambdef: 'lambda' [varargslist] ':' old_test
```

(下页继续)

(续上页)

```
test: or_test ['if' or_test 'else' test] | lambdef
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
power: atom trailer* ['**' factor]
atom: ('(' [yield_expr|testlist_comp] ')' |
       '[' [listmaker] ']' |
       '{' [dictorsetmaker] '}' |
       '`' testlist1 '`' |
       NAME | NUMBER | STRING+)
listmaker: test ( list_for | (',' test)* [','] )
testlist_comp: test ( comp_for | (',' test)* [','] )
lambdef: 'lambda' [varargslist] ':' test
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
subscriptlist: subscript (',' subscript)* [',']
subscript: '.' '.' '.' | test | [test] ':' [test] [sliceop]
sliceop: ':' [test]
exprlist: expr (',' expr)* [',']
testlist: test (',' test)* [',']
dictorsetmaker: ( (test ':' test (comp_for | (',' test ':' test)* [','])) |
                  (test (comp_for | (',' test)* [','])) )

classdef: 'class' NAME ['(' [testlist] ')'] ':' suite

arglist: (argument ',')* (argument [',']
                         |'*' test (',' argument)* [',' '**' test]
                         |'**' test)
# The reason that keywords are test nodes instead of NAME is that using NAME
# results in an ambiguity. ast.c makes sure it's a NAME.
argument: test [comp_for] | test '=' test

list_iter: list_for | list_if
list_for: 'for' exprlist 'in' testlist_safe [list_iter]
list_if: 'if' old_test [list_iter]

comp_iter: comp_for | comp_if
comp_for: 'for' exprlist 'in' or_test [comp_iter]
comp_if: 'if' old_test [comp_iter]

testlist1: test (',' test)*

# not used in grammar, but may appear in "node" passed from Parser to Compiler
encoding_decl: NAME

yield_expr: 'yield' [testlist]
```

APPENDIX A

# 术语对照表

**>>>** 交互式终端中默认的 Python 提示符。往往会显示于能以交互方式在解释器里执行的样例代码之前。

**...** The default Python prompt of the interactive shell when entering code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

**2to3** 一个将 Python 2.x 代码转换为 Python 3.x 代码的工具，能够处理大部分通过解析源码并遍历解析树可检测到的不兼容问题。

2to3 包含在标准库中，模块名为 `lib2to3`；并提供一个独立入口点 `Tools/scripts/2to3`。参见 2to3-reference。

**abstract base class –抽象基类** Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with *magic methods*). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections` module), numbers (in the `numbers` module), and streams (in the `io` module). You can create your own ABCs with the `abc` module.

**argument –参数** A value passed to a *function* (or *method*) when calling the function. There are two types of arguments:

- *关键字参数*: 在函数调用中前面带有标识符（例如 `name=`）或者作为包含在前面带有 `**` 的字典里的值传入。举例来说，`3` 和 `5` 在以下对 `complex()` 的调用中均属于关键字参数:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *位置参数*: 不属于关键字参数的参数。位置参数可出现于参数列表的开头以及/或者作为前面带有 `*` 的*iterable* 里的元素被传入。举例来说，`3` 和 `5` 在以下调用中均属于位置参数:

```
complex(3, 5)
complex(*(3, 5))
```

参数会被赋值给函数体中对应的局部变量。有关赋值规则参见调用 一节。根据语法，任何表达式都可用来表示一个参数；最终算出的值会被赋给对应的局部变量。

See also the *parameter* glossary entry and the FAQ question on the difference between arguments and parameters.

**attribute –属性** 关联到一个对象的值，可以使用点号表达式通过其名称来引用。例如，如果一个对象 *o* 具有一个属性 *a*，就可以用 *o.a* 来引用它。

**BDFL** Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python’s creator.

**bytes-like object –字节类对象** An object that supports the buffer protocol, like `str`, `bytearray` or `memoryview`. Bytes-like objects can be used for various operations that expect binary data, such as compression, saving to a binary file or sending over a socket. Some operations need the binary data to be mutable, in which case not all bytes-like objects can apply.

**bytecode –字节码** Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This “intermediate language” is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

字节码指令列表可以在 dis 模块的文档中查看。

**class –类** 用来创建用户定义对象的模板。类定义通常包含对该类的实例进行操作的方法定义。

**classic class** Any class which does not inherit from `object`. See *new-style class*. Classic classes have been removed in Python 3.

**coercion –强制类型转换** The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` built-in function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

**complex number –复数** 对普通实数系统的扩展，其中所有数字都被表示为一个实部和一个虚部的和。虚数是虚数单位（`-1` 的平方根）的实倍数，通常在数学中写为 `i`，在工程学中写为 `j`。Python 内置了对复数的支持，采用工程学标记方式；虚部带有一个 `j` 后缀，例如 `3+1j`。如果需要 `math` 模块内对象的对应复数版本，请使用 `cmath`，复数的使用是一个比较高级的数学特性。如果你感觉没有必要，忽略它们也几乎不会有任何问题。

**context manager –上下文管理器** 在 *`with`* 语句中使用，通过定义 *`__enter__()`* 和 *`__exit__()`* 方法来控制环境状态的对象。参见 **PEP 343**。

**CPython** Python 编程语言的规范实现，在 python.org 上发布。”CPython” 一词用于在必要时将此实现与其他实现例如 Jython 或 IronPython 相区别。

**decorator –装饰器** 返回值为另一个函数的函数，通常使用 `@wrapper` 语法形式来进行函数变换。装饰器的常见例子包括 `classmethod()` 和 `staticmethod()`。

装饰器语法只是一种语法糖，以下两个函数定义在语义上完全等价:

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

同的样概念也适用于类，但通常较少这样使用。有关装饰器的详情可参见函数定义 和类定义 的文档。

**descriptor –描述器** Any *new-style* object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

有关描述符的方法的详情可参看实现描述器。

**dictionary –字典** An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

**dictionary view –字典视图** The objects returned from `dict.viewkeys()`, `dict.viewvalues()`, and `dict.viewitems()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See dict-views.

**docstring –文档字符串** 作为类、函数或模块之内的第一个表达式出现的字符串字面值。它在代码执行时会被忽略，但会被解释器识别并放入所在类、函数或模块的 `__doc__` 属性中。由于它可用于代码内省，因此是对象存放文档的规范位置。

**duck-typing –鸭子类型** 指一种编程风格，它并不依靠查找对象类型来确定其是否具有正确的接口，而是直接调用或使用其方法或属性（"看起来像鸭子，叫起来也像鸭子，那么肯定就是鸭子。"）由于强调接口而非特定类型，设计良好的代码可通过允许多态替代来提升灵活性。鸭子类型避免使用 `type()` 或 `isinstance()` 检测。(但要注意鸭子类型可以使用抽象基类 作为补充。) 而往往会采用 `hasattr()` 检测或是*EAFP* 编程。

**EAFP** "求原谅比求许可更容易" 的英文缩写。这种 Python 常用代码编写风格会假定所需的键或属性存在，并在假定错误时捕获异常。这种简洁快速风格的特点就是大量运用 `try` 和 `except` 语句。于其相对的则是所谓*LBYL* 风格，常见于 C 等许多其他语言。

**expression –表达式** A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statement*s which cannot be used as expressions, such as `print` or `if`. Assignments are also statements, not expressions.

**extension module –扩展模块** 以 C 或 C++ 编写的模块，使用 Python 的 C API 来与语言核心以及用户代码进行交互。

**file object –文件对象** 对外提供面向文件 API 以使用下层资源的对象（带有 `read()` 或 `write()` 这样的方法）。根据其创建方式的不同，文件对象可以处理对真实磁盘文件，对其他类型存储，或是对通讯设备的访问（例如标准输入/输出、内存缓冲区、套接字、管道等等）。文件对象也被称为 *文件类对象* 或 *流*。

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

**file-like object –文件类对象** *file object* 的同义词。

**finder –查找器** An object that tries to find the *loader* for a module. It must implement a method named `find_module()`. See **PEP 302** for details.

**floor division –向下取整除法** 向下舍入到最接近的整数的数学除法。向下取整除法的运算符是 `//`。例如，表达式 `11 // 4` 的计算结果是 `2`，而与之相反的是浮点数的真正除法返回 `2.75`。注意 `(-11) // 4` 会返回 `-3` 因为这是 `-2.75` 向下舍入得到的结果。见 **PEP 238**。

**function –函数** 可以向调用者返回某个值的一组语句。还可以向其传入零个或多个参数 并在函数体执行中被使用。另见*parameter*, *method* 和函数定义 等节。

**__future__** A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression `11/4` currently evaluates to `2`. If the module in which it

is executed had enabled *true division* by executing:

```
from __future__ import division
```

the expression `11/4` would evaluate to `2.75`. By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

**garbage collection –垃圾回收** The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles.

**generator –生成器** A function which returns an iterator. It looks like a normal function except that it contains *yield* statements for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function. Each *yield* temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

**generator expression –生成器表达式** An expression that returns an iterator. It looks like a normal expression followed by a *for* expression defining a loop variable, range, and an optional *if* expression. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

**GIL** 参见*global interpreter lock*。

**global interpreter lock –全局解释器锁** *CPython* 解释器所采用的一种机制，它确保同一时刻只有一个线程在执行 Python *bytecode*。此机制通过设置对象模型（包括 `dict` 等重要内置类型）针对并发访问的隐式安全简化了 CPython 实现。给整个解释器加锁使得解释器多线程运行更方便，其代价则是牺牲了在多处理器上的并行性。

不过，某些标准库或第三方库的扩展模块被设计为在执行计算密集型任务如压缩或哈希时释放 GIL。此外，在执行 I/O 操作时也总是会释放 GIL。

创建一个（以更精细粒度来锁定共享数据的）“自由线程”解释器的努力从未获得成功，因为这会牺牲在普通单处理器情况下的性能。据信克服这种性能问题的措施将导致实现变得更复杂，从而更难以维护。

**hashable –可哈希** An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a *__hash__()* method), and can be compared to other objects (it needs an *__eq__()* or *__cmp__()* method). Hashable objects which compare equal must have the same hash value.

可哈希性使得对象能够作为字典键或集合成员使用，因为这些数据结构要在内部使用哈希值。

All of Python’s immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal (except with themselves), and their hash value is derived from their `id()`.

**IDLE** Python 的 IDE，“集成开发与学习环境”的英文缩写。是 Python 标准发行版附带的基本编程器和解释器环境。

**immutable –不可变** 具有固定值的对象。不可变对象包括数字、字符串和元组。这样的对象不能被改变。如果必须存储一个不同的值，则必须创建新的对象。它们在需要常量哈希值的地方起着重要作用，例如作为字典中的键。

**integer division** Mathematical division discarding any remainder. For example, the expression `11/4` currently evaluates to `2` in contrast to the `2.75` returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is

another numeric type (such as a `float`), the result will be coerced (see *coercion*) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also *__future__*.

**importing –导入** 令一个模块中的 Python 代码能为另一个模块中的 Python 代码所使用的过程。

**importer –导入器** 查找并加载模块的对象；此对象既属于*finder* 又属于*loader*。

**interactive –交互** Python 带有一个交互式解释器，即你可以在解释器提示符后输入语句和表达式，立即执行并查看其结果。只需不带参数地启动 `python` 命令（也可以在你的计算机开始菜单中选择相应菜单项）。在测试新想法或检验模块和包的时候用这种方式会非常方便（请记得使用 `help(x)`）。

**interpreted –解释型** Python 一是种解释型语言，与之相对的是编译型语言，虽然两者的区别由于字节码编译器的存在而会有所模糊。这意味着源文件可以直接运行而不必显式地创建可执行文件再运行。解释型语言通常具有比编译型语言更短的开发/调试周期，但是其程序往往运行得更慢。参见*interactive*。

**iterable –可迭代对象** An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `file` and objects of any classes you define with an *__iter__()* or *__getitem__()* method. Iterables can be used in a *for* loop and in many other places where a sequence is needed (`zip()`, `map()`, ···). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

**iterator –迭代器** An object representing a stream of data. Repeated calls to the iterator's *next()* method return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its *next()* method just raise `StopIteration` again. Iterators are required to have an *__iter__()* method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a *for* loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

更多信息可查看 typeiter。

**key function –键函数** 键函数或称整理函数，是能够返回用于排序或排位的值的可调用对象。例如，`locale.strxfrm()` 可用于生成一个符合特定区域排序约定的排序键。

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a key function. For example. the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, an ad-hoc key function can be built from a *lambda* expression such as `lambda r: (r[0], r[2])`. Also, the `operator` module provides three key function constructors: `attrgetter()`, `itemgetter()`, and `methodcaller()`. See the Sorting HOW TO for examples of how to create and use key functions.

**keyword argument –关键字参数** 参见*argument*。

**lambda** 由一个单独*expression* 构成的匿名内联函数，表达式会在调用时被求值。创建 lambda 函数的句法为 `lambda [parameters]: expression`

**LBYL** "先查看后跳跃"的英文缩写。这种代码编写风格会在进行调用或查找之前显式地检查前提条件。此风格与*EAFP* 方式恰成对比，其特点是大量使用*if* 语句。

在多线程环境中，LBYL 方式会导致"查看"和"跳跃"之间发生条件竞争风险。例如，以下代码 `if key in mapping: return mapping[key]` 可能由于在检查操作之后其他线程从 *mapping* 中移除了 *key* 而出错。这种问题可通过加锁或使用 EAFP 方式来解决。

**list –列表** Python 内置的一种*sequence*。虽然名为列表，但更类似于其他语言中的数组而非链接列表，因为访问元素的时间复杂度为 O(1)。

**list comprehension –列表推导式** A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The *`if`* clause is optional. If omitted, all elements in `range(256)` are processed.

**loader –加载器** An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See **PEP 302** for details.

**magic method –魔术方法** *special method* 的非正式同义词。

**mapping –映射** A container object that supports arbitrary key lookups and implements the methods specified in the `Mapping` or `MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

**metaclass –元类** 一种用于创建类的类。类定义包含类名、类字典和基类列表。元类负责接受上述三个参数并创建相应的类。大部分面向对象的编程语言都会提供一个默认实现。Python 的特别之处在于可以创建自定义元类。大部分用户永远不需要这个工具，但当需要出现时，元类可提供强大而优雅的解决方案。它们已被用于记录属性访问日志、添加线程安全性、跟踪对象创建、实现单例，以及其他许多任务。

更多详情参见*自定义类创建*。

**method 方法** 在类内部定义的函数。如果作为该类的实例的一个属性来调用，方法将会获取实例对象作为其第一个*argument* (通常命名为 `self`)。参见*function* 和*nested scope*。

**method resolution order –方法解析顺序** 方法解析顺序就是在查找成员时搜索全部基类所用的先后顺序。请查看 Python 2.3 方法解析顺序 了解自 2.3 版起 Python 解析器所用相关算法的详情。

**module 模块** 此对象是 Python 代码的一种组织单位。各模块具有独立的命名空间，可包含任意 Python 对象。模块可通过*importing* 操作被加载到 Python 中。

另见*package*。

**MRO** 参见*method resolution order*。

**mutable –可变** 可变对象可以在其 `id()` 保持固定的情况下改变其取值。另请参见*immutable*。

**named tuple –具名元组** Any tuple-like class whose indexable elements are also accessible using named attributes (for example, `time.localtime()` returns a tuple-like object where the *year* is accessible either with an index such as `t[0]` or with a named attribute like `t.tm_year`).

A named tuple can be a built-in type such as `time.struct_time`, or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function `collections.namedtuple()`. The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

**namespace –命名空间** The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

**nested scope –嵌套作用域** The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

**new-style class** –**新式类** Any class which inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, and `__getattribute__()`.

More information can be found in *New-style and classic classes*.

**object** –**对象** 任何具有状态（属性或值）以及预定义行为（方法）的数据。object 也是任何*new-style class* 的最顶层基类名。

**package** –**包** 一种可包含子模块或递归地包含子包的 Python *module*。从技术上说，包是带有 `__path__` 属性的 Python 模块。

**parameter** –**形参** A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are four types of parameters:

- *positional-or-keyword*：位置或关键字，指定一个可以作为*位置参数* 传入也可以作为*关键字参数* 传入的实参。这是默认的形参类型，例如下面的 *foo* 和 *bar*:

```
def func(foo, bar=None): ...
```

- *positional-only*：仅限位置，指定一个只能按位置传入的参数。Python 中没有定义仅限位置形参的语法。但是一些内置函数有仅限位置形参（比如 `abs()`）。
- *var-positional*：可变位置，指定可以提供由一个任意数量的位置参数构成的序列（附加在其他形参已接受的位置参数之后）。这种形参可通过在形参名称前加缀 `*` 来定义，例如下面的 *args*:

```
def func(*args, **kwargs): ...
```

- *var-keyword*：可变关键字，指定可以提供任意数量的关键字参数（附加在其他形参已接受的关键字参数之后）。这种形参可通过在形参名称前加缀 `**` 来定义，例如上面的 *kwargs*。

形参可以同时指定可选和必选参数，也可以为某些可选参数指定默认值。

See also the *argument* glossary entry, the FAQ question on the difference between arguments and parameters, and the *函数定义* section.

**PEP** "Python 增强提议" 的英文缩写。一个 PEP 就是一份设计文档，用来向 Python 社区提供信息，或描述一个 Python 的新增特性及其进度或环境。PEP 应当提供精确的技术规格和所提议特性的原理说明。

PEP 应被作为提出主要新特性建议、收集社区对特定问题反馈以及为必须加入 Python 的设计决策编写文档的首选机制。PEP 的作者有责任在社区内部建立共识，并应将不同意见也记入文档。

参见 **PEP 1**。

**positional argument** –**位置参数** 参见*argument*。

**Python 3000** Python 3.x 发布路线的昵称（这个名字在版本 3 的发布还遥遥无期的时候就已出现了）。有时也被缩写为 "Py3k"。

**Pythonic** 指一个思路或一段代码紧密遵循了 Python 语言最常用的风格和理念，而不是使用其他语言中通用的概念来实现代码。例如，Python 的常用风格是使用 *for* 语句循环来遍历一个可迭代对象中的所有元素。许多其他语言没有这样的结构，因此不熟悉 Python 的人有时会选择使用一个数字计数器:

```
for i in range(len(food)):
    print food[i]
```

而相应的更简洁更 Pythonic 的方法是这样的:

```
for piece in food:
    print piece
```

**reference count –引用计数** 对特定对象的引用的数量。当一个对象的引用计数降为零时，所分配资源将被释放。引用计数对 Python 代码来说通常是不可见的，但它是*CPython* 实现的一个关键元素。sys 模块定义了一个 getrefcount() 函数，程序员可调用它来返回特定对象的引用计数。

**__slots__** A declaration inside a *new-style class* that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

**sequence –序列** An *iterable* which supports efficient element access using integer indices via the *__getitem__()* special method and defines a len() method that returns the length of the sequence. Some built-in sequence types are list, str, tuple, and unicode. Note that dict also supports *__getitem__()* and *__len__()*, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

**slice –切片** An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, [] with colons between numbers when several are given, such as in variable_name[1:3:5]. The bracket (subscript) notation uses slice objects internally (or in older versions, *__getslice__()* and *__setslice__()*).

**special method –特殊方法** 一种由 Python 隐式调用的方法，用来对某个类型执行特定操作例如相加等等。这种方法的名称的首尾都为双下划线。特殊方法的文档参见特殊方法名称。

**statement –语句** 语句是程序段（一个代码“块”）的组成单位。一条语句可以是一个*expression* 或某个带有关键字的结构，例如*if*、*while* 或*for*。

**struct sequence** A tuple with named elements. Struct sequences expose an interface similiar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like _make() or _asdict(). Examples of struct sequences include sys.float_info and the return value of os.stat().

**triple-quoted string –三引号字符串** 首尾各带三个连续双引号（"）或者单引号（'）的字符串。它们在功能上与首尾各用一个引号标注的字符串没有什么不同，但是有多种用处。它们允许你在字符串内包含未经转义的单引号和双引号，并且可以跨越多行而无需使用连接符，在编写文档字符串时特别好用。

**type –类型** 类型决定一个 Python 对象属于什么种类；每个对象都具有一种类型。要知道对象的类型，可以访问它的 __class__ 属性，或是通过 type(obj) 来获取。

**universal newlines –通用换行** A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention '\n', the Windows convention '\r\n', and the old Macintosh convention '\r'. See **PEP 278** and **PEP 3116**, as well as str.splitlines() for an additional use.

**virtual environment –虚拟环境** 一种采用协作式隔离的运行时环境，允许 Python 用户和应用程序在安装和升级 Python 分发包时不会干扰到同一系统上运行的其他 Python 应用程序的行为。

**virtual machine –虚拟机** 一台完全通过软件定义的计算机。Python 虚拟机可执行字节码编译器所生成的*bytecode*。

**Zen of Python –Python 之禅** 列出 Python 设计的原则与哲学，有助于理解与使用这种语言。查看其具体内容可在交互模式提示符中输入“import this”。

APPENDIX B

# 文档说明

这些文档生成自 reStructuredText 原文档，由 Sphinx （一个专门为 Python 文档写的文档生成器）创建。

本文档和它所用工具链的开发完全是由志愿者完成的，这和 Python 本身一样。如果您想参与进来，请阅读 reporting-bugs 了解如何参与。我们随时欢迎新的志愿者！

特别鸣谢：

- Fred L. Drake, Jr.，创造了用于早期 Python 文档的工具链，以及撰写了非常多的文档；
- Docutils 软件包 项目，创建了 reStructuredText 文本格式和 Docutils 软件套件；
- Fredrik Lundh，Sphinx 从他的 Alternative Python Reference 项目中获得了很多好的想法。

## B.1 Python 文档的贡献者

有很多对 Python 语言，Python 标准库和 Python 文档有贡献的人，随 Python 源代码发布的 Misc/ACKS 文件列出了部分贡献者。

有了 Python 社区的输入和贡献，Python 才有了如此出色的文档 - 谢谢你们！

APPENDIX C

# 历史和许可证

## C.1 该软件的历史

Python 由荷兰数学和计算机科学研究学会（CWI，见 https://www.cwi.nl/ ）的 Guido van Rossum 于 1990 年代初设计，作为一门叫做 ABC 的语言的替代品。尽管 Python 包含了许多来自其他人的贡献，Guido 仍是其主要作者。

1995 年，Guido 在弗吉尼亚州的国家创新研究公司（CNRI，见 https://www.cnri.reston.va.us/ ）继续他在 Python 上的工作，并在那里发布了该软件的多个版本。

2000 年五月，Guido 和 Python 核心开发团队转到 BeOpen.com 并组建了 BeOpen PythonLabs 团队。同年十月，PythonLabs 团队转到 Digital Creations (现为 Zope Corporation；见 https://www.zope.org/)。2001 年，Python 软件基金会 (PSF，见 https://www.python.org/psf/) 成立，这是一个专为拥有 Python 相关知识产权而创建的非营利组织。Zope Corporation 现在是 PSF 的赞助成员。

所有的 Python 版本都是开源的（有关开源的定义参阅 https://opensource.org/ ）。历史上，绝大多数 Python 版本是 GPL 兼容的；下表总结了各个版本情况。

| 发布版本 | 源自 | 年份 | 所有者 | GPL 兼容？ |
|---|---|---|---|---|
| 0.9.0 至 1.2 | n/a | 1991-1995 | CWI | 是 |
| 1.3 至 1.5.2 | 1.2 | 1995-1999 | CNRI | 是 |
| 1.6 | 1.5.2 | 2000 | CNRI | 否 |
| 2.0 | 1.6 | 2000 | BeOpen.com | 否 |
| 1.6.1 | 1.6 | 2001 | CNRI | 否 |
| 2.1 | 2.0+1.6.1 | 2001 | PSF | 否 |
| 2.0.1 | 2.0+1.6.1 | 2001 | PSF | 是 |
| 2.1.1 | 2.1+2.0.1 | 2001 | PSF | 是 |
| 2.1.2 | 2.1.1 | 2002 | PSF | 是 |
| 2.1.3 | 2.1.2 | 2002 | PSF | 是 |
| 2.2 及更高 | 2.1.1 | 2001 至今 | PSF | 是 |

**注解：** GPL 兼容并不意味着 Python 在 GPL 下发布。与 GPL 不同，所有 Python 许可证都允许您分发修改后

的版本，而无需开源所做的更改。GPL 兼容的许可证使得 Python 可以与其它在 GPL 下发布的软件结合使用；但其它的许可证则不行。

感谢众多在 Guido 指导下工作的外部志愿者，使得这些发布成为可能。

## C.2 获取或以其他方式使用 Python 的条款和条件

### C.2.1 用于 PYTHON 2.7.18 的 PSF 许可协议

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   2.7.18 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 2.7.18 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2020 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 2.7.18 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 2.7.18 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
   2.7.18.

4. PSF is making Python 2.7.18 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 2.7.18 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.18
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.18, 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 2.7.18, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

### C.2.2 用于 PYTHON 2.0 的 BEOPEN.COM 许可协议

BEOPEN PYTHON 开源许可协议第 1 版

```
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
```

(下页继续)

(续上页)

```
   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.
```

### C.2.3 用于 PYTHON 1.6.1 的 CNRI 许可协议

```
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
```

(下页继续)

(续上页)

```
   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.
```

### C.2.4 用于 PYTHON 0.9.0 至 1.2 的 CWI 许可协议

```
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.
```

## C.3 被收录软件的许可证与鸣谢

本节是 Python 发行版中收录的第三方软件的许可和致谢清单，该清单是不完整且不断增长的。

### C.3.1 Mersenne Twister

_random 模块包含基于 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 下载的代码。以下是原始代码的完整注释（声明）：

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

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

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

(下页继续)

(续上页)

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

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

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

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

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


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

### C.3.2 套接字

socket 模块使用 getaddrinfo() 和 getnameinfo() 函数，这些函数源代码在 WIDE 项目（http://www.wide.ad.jp/）的单独源文件中。

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
```

(下页继续)

(续上页)

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

### C.3.3 Floating point exception control

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

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

### C.3.4 MD5 message digest algorithm

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

```
Copyright (C) 1999, 2002 Aladdin Enterprises.  All rights reserved.

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
      http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite
(section A.5) but excluding the rest of Appendix A.  It does not include
any code or documentation that is identified in the RFC as being
copyrighted.

The original and principal author of md5.h is L. Peter Deutsch
<ghost@aladdin.com>.  Other authors are noted in the change history
that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed
      references to Ghostscript; clarified derivation from RFC 1321;
      now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
      added conditionalization for C++ compilation from Martin
      Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
```

### C.3.5 异步套接字服务

asynchat and asyncore 模块包含以下声明:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

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

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

### C.3.6 Cookie 管理

The Cookie module contains the following notice:

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

               All Rights Reserved

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

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

### C.3.7 执行追踪

trace 模块包含以下声明:

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

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

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

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

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


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

### C.3.8 UUencode 与 UUdecode 函数

uu 模块包含以下声明:

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

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
```

(下页继续)

(续上页)

```
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

### C.3.9 XML 远程过程调用

The `xmlrpclib` module contains the following notice:

```
    The XML-RPC client interface is

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

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

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

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

### C.3.10 test_epoll

The `test_epoll` contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(下页继续)

(续上页)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

### C.3.11 Select kqueue

The `select` and contains the following notice for the kqueue interface:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

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

### C.3.12 strtod and dtoa

`Python/dtoa.c` 文件提供了 C 语言的 dtoa 和 strtod 函数，用于将 C 语言的双精度型和字符串进行转换，该文件由 David M. Gay 的同名文件派生而来，当前可从 http://www.netlib.org/fp/ 下载。2009 年 3 月 16 日检索到的原始文件包含以下版权和许可声明:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
```

(下页继续)

(续上页)

```
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ************************************************************/
```

### C.3.13 OpenSSL

如果操作系统可用，则 hashlib, posix, ssl, crypt 模块使用 OpenSSL 库来提高性能。此外，适用于 Python 的 Windows 和 Mac OS X 安装程序可能包括 OpenSSL 库的拷贝，所以在此处也列出了 OpenSSL 许可证的拷贝:

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   *
   * 1. Redistributions of source code must retain the above copyright
   *    notice, this list of conditions and the following disclaimer.
   *
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in
   *    the documentation and/or other materials provided with the
   *    distribution.
   *
   * 3. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
```

(下页继续)

(续上页)

```
   *    acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
   *
   * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
   * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
   * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
   * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
   * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
   * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
   * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
   * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
   * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
   * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
   * OF THE POSSIBILITY OF SUCH DAMAGE.
   * ====================================================================
   *
   * This product includes cryptographic software written by Eric Young
   * (eay@cryptsoft.com).  This product includes software written by Tim
   * Hudson (tjh@cryptsoft.com).
   *
   */

Original SSLeay License
-----------------------

  /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
   * All rights reserved.
   *
   * This package is an SSL implementation written
   * by Eric Young (eay@cryptsoft.com).
   * The implementation was written so as to conform with Netscapes SSL.
   *
   * This library is free for commercial and non-commercial use as long as
   * the following conditions are aheared to.  The following conditions
   * apply to all code found in this distribution, be it the RC4, RSA,
   * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
   * included with this distribution is covered by the same copyright terms
   * except that the holder is Tim Hudson (tjh@cryptsoft.com).
   *
   * Copyright remains Eric Young's, and as such any Copyright notices in
   * the code are not to be removed.
   * If this package is used in a product, Eric Young should be given attribution
   * as the author of the parts of the library used.
   * This can be in the form of a textual message at program startup or
   * in documentation (online or textual) provided with the package.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   * 1. Redistributions of source code must retain the copyright
   *    notice, this list of conditions and the following disclaimer.
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in the
   *    documentation and/or other materials provided with the distribution.
```

(下页继续)

(续上页)

```
   * 3. All advertising materials mentioning features or use of this software
   *    must display the following acknowledgement:
   *    "This product includes cryptographic software written by
   *     Eric Young (eay@cryptsoft.com)"
   *    The word 'cryptographic' can be left out if the rouines from the library
   *    being used are not cryptographic related :-).
   * 4. If you include any Windows specific code (or a derivative thereof) from
   *    the apps directory (application code) you must include an acknowledgement:
   *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
   *
   * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
   * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
   * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
   * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
   * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
   * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
   * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
   * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
   * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
   * SUCH DAMAGE.
   *
   * The licence and distribution terms for any publically available version or
   * derivative of this code cannot be changed.  i.e. this code cannot simply be
   * copied and put under another distribution licence
   * [including the GNU Public Licence.]
   */
```

### C.3.14 expat

除非使用 `--with-system-expat` 配置了构建，否则 `pyexpat` 扩展都是用包含 expat 源的拷贝构建的:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

### C.3.15 libffi

除非使用 `--with-system-libffi` 配置了构建，否则 `_ctypes` 扩展都是包含 libffi 源的拷贝构建的:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

### C.3.16 zlib

如果系统上找到的 zlib 版本太旧而无法用于构建，则使用包含 zlib 源代码的拷贝来构建 `zlib` 扩展:

```
Copyright (C) 1995-2010 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

# APPENDIX D

## Copyright

Python 与这份文档：



有关完整的许可证和许可信息，参见历史和许可证。

# 索引

## 非字母

## A



## B

## C

## D



## E

## F



## G

## M



## N

## O



## P

## Q



## R



## S

## T



## U



## V

## W



## X

## Y



## Z