The Python Language Reference

发行版本 *3.13.0*

Guido van Rossum and the Python development team

十一月 01, 2024

Python Software Foundation
Email: docs@python.org

Contents

本参考手册介绍了 Python 句法与“核心语义”。在力求简明扼要的同时，我们也尽量做到准确、完整。有关内置对象类型、内置函数、模块的语义在 library-index 中介绍。有关本语言的非正式介绍，请参阅 tutorial-index 。对于 C 或 C++ 程序员，我们还提供了两个手册：extending-index 介绍了如何编写 Python 扩展模块，c-api-index 则详细介绍了 C/C++ 的可用接口。

CHAPTER 1

概述

本手册仅描述 Python 编程语言，不宜当作教程。

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:-)

在语言参考文档里加入过多的实现细节是很危险的 --- 具体实现可能发生改变，对同一语言的其他实现可能使用不同的方式。而在另一方面，CPython 是得到广泛使用的 Python 实现 (然而其他一些实现的拥护者也在增加)，其中的特殊细节有时也值得一提，特别是当其实现方式导致额外的限制时。因此，你会发现在正文里不时会跳出来一些简短的" 实现注释"。

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。想了解更多信息请访问 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 版 Python 实现，不同于 Python.NET，这是一个生成 IL 的完整 Python 实现，并会将 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。想了解更多信息，请参看 IronPython 网站。

PyPy
: 一个完全使用 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，例如非栈式支持和实时编译器等。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）来鼓励对语言本身的试验。更多信息可在 PyPy 项目主页 获取。

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

句法和词法分析的描述采用经过改进的 Backus–Naur form (BNF) 语法标注。这将使用以下定义样式:

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

第一行表示 `name` 是 `lc_letter` 之后跟零个或多个 `lc_letter` 和下划线。而 `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，它是此标注中绑定程度最低的操作符。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。字符串字面值包含在引号内。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 *解析器*读取，输入解析器的是 *词法分析器*生成的 *形符*流。本章介绍词法分析器怎样把文件拆成形符。

Python 将读取的程序文本转为 Unicode 代码点；编码声明用于指定源文件的编码，默认为 UTF-8，详见 **PEP 3120**。源文件不能解码时，触发 `SyntaxError`。

2.1 行结构

Python 程序可以拆分为多个 *逻辑行*。

2.1.1 逻辑行

NEWLINE 形符表示结束逻辑行。语句不能超出逻辑行的边界，除非句法支持 NEWLINE（例如，复合语句中的多行子语句）。根据显式或隐式 *行拼接*规则，一个或多个 *物理行*可组成逻辑行。

2.1.2 物理行

物理行是一序列字符，由行尾序列终止。源文件和字符串可使用任意标准平台行终止序列 - Unix ASCII 字符 LF（换行）、Windows ASCII 字符序列 CR LF（回车换行）、或老式 Macintosh ASCII 字符 CR（回车）。不管在哪个平台，这些形式均可等价使用。输入结束也可以用作最终物理行的隐式终止符。

嵌入 Python 时，传入 Python API 的源码字符串应使用 C 标准惯例换行符（`\n`，代表 ASCII 字符 LF，行终止符）。

2.1.3 注释

注释以井号（`#`）开头，在物理行末尾截止。注意，井号不是字符串字面值。除非应用隐式行拼接规则，否则，注释代表逻辑行结束。句法不解析注释。

2.1.4 编码声明

Python 脚本第一或第二行的注释匹配正则表达式 `coding[=:]\s*([-\w.]+)` 时，该注释会被当作编码声明；这个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，在第二行时，则第一行必须也是注释。编码表达式的形式如下：

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

这也是 GNU Emacs 认可的形式，此外，还支持如下形式：

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

这是 Bram Moolenaar 的 VIM 认可的形式。

如果没有找到编码格式声明，则默认编码格式为 UTF-8。如果文件的隐式或显式编码格式为 UTF-8，则初始的 UTF-8 字节序标志（b'xefxbbxbf'）将被忽略而不会报告语法错误。

如果声明了编码格式，该编码格式的名称必须是 Python 可识别的 (参见 standard-encodings)。编码格式会被用于所有的词法分析，包括字符串字面值、注释和标识符等。

2.1.5 显式拼接行

两个及两个以上的物理行可用反斜杠（\）拼接为一个逻辑行，规则如下：以不在字符串或注释内的反斜杠结尾时，物理行将与下一行拼接成一个逻辑行，并删除反斜杠及其后的换行符。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
```

以反斜杠结尾的行，不能加注释；反斜杠也不能拼接注释。除字符串字面值外，反斜杠不能拼接形符（如，除字符串字面值外，不能用反斜杠把形符切分至两个物理行）。反斜杠只能在代码的字符串字面值里，在其他任何位置都是非法的。

2.1.6 隐式拼接行

圆括号、方括号、花括号内的表达式可以分成多个物理行，不必使用反斜杠。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
```

隐式行拼接可含注释；后续行的缩进并不重要；还支持空的后续行。隐式拼接行之间没有 NEWLINE 形符。三引号字符串支持隐式拼接行（见下文），但不支持注释。

2.1.7 空白行

只包含空格符、制表符、换页符、注释的逻辑行会被忽略（即不生成 NEWLINE 形符）。交互模式输入语句时，空白行的处理方式可能因读取 - 求值 - 打印循环（REPL）的具体实现方式而不同。标准交互模式解释器中，完全空白的逻辑行（即连空格或注释都没有）将结束多行复合语句。

2.1.8 缩进

逻辑行开头的空白符（空格符和制表符）用于计算该行的缩进层级，决定语句组块。

制表符（从左至右）被替换为一至八个空格，缩进空格的总数是八的倍数（与 Unix 的规则保持一致）。首个非空字符前的空格数决定了该行的缩进层次。缩进不能用反斜杠进行多行拼接；首个反斜杠之前的空白符决定了缩进的层次。

源文件混用制表符和空格符缩进时，因空格数量与制表符相关，由此产生的不一致将导致不能正常识别缩进层次，从而触发 `TabError`。

跨平台兼容性说明： 鉴于非 UNIX 平台文本编辑器本身的特性，请勿在源文件中混用制表符和空格符。另外也请注意，不同平台有可能会显式限制最大缩进层级。

行首含换页符时，缩进计算将忽略该换页符。换页符在行首空白符内其他位置的效果未定义（例如，可能导致空格计数重置为零）。

连续行的缩进层级以堆栈形式生成 INDENT 和 DEDENT 形符，说明如下。

读取文件第一行前，先向栈推入一个零值，该零值不会被移除。推入栈的层级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。如果相等，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移除，每移除一级数值生成一个 DEDENT 形符。文件末尾，栈中剩余的每个大于零的数值生成一个 DEDENT 形符。

下面的 Python 代码缩进示例虽然正确，但含混不清：

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

下例展示了多种缩进错误：

```
 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，解析器可以识别前三个错误；只有最后一个错误由词法分析器识别 --- `return r` 的缩进无法匹配从栈里移除的缩进层级。）

2.1.9 形符间的空白字符

除非在逻辑行开头或字符串内，空格符、制表符、换页符等空白符都可以分隔形符。要把两个相连形符解读为不同形符，需要用空白符分隔（例如，ab 是一个形符，a b 则是两个形符）。

2.2 其他形符

除 NEWLINE、INDENT、DEDENT 外，还有 标识符、关键字、字面值、运算符、分隔符等形符。空白符（前述的行终止符除外）不是形符，可用于分隔形符。存在二义性时，将从左至右，读取尽量长的字符串组成合法形符。

2.3 标识符和关键字

标识符（也称为 名称）的词法定义说明如下。

Python 标识符的句法基于 Unicode 标准附件 UAX-31，并加入了下文定义的细化与修改；详见 **PEP 3131** 。

Within the ASCII range (U+0001..U+007F), the valid characters for identifiers include the uppercase and lowercase letters `A` through `Z`, the underscore _ and, except for the first character, the digits `0` through `9`. Python 3.0 introduced additional characters from outside the ASCII range (see **PEP 3131**). For these characters, the classification uses the version of the Unicode Character Database as included in the `unicodedata` module.

标识符的长度没有限制，但区分大小写。

```
identifier     ::=   xid_start xid_continue*
id_start       ::=   <all characters in general categories Lu, Ll, Lt, Lm, Lo, Nl, the underscore,
```

```
id_continue    ::=   <all characters in id_start, plus characters in the categories Mn, Mc, Nd, Pc
xid_start      ::=   <all characters in id_start whose NFKC normalization is in "id_start xid_cont
xid_continue   ::=   <all characters in id_continue whose NFKC normalization is in "id_continue*">
```

上述 Unicode 类别码的含义:

- *Lu* - 大写字母
- *Ll* - 小写字母
- *Lt* - 词首大写字母
- *Lm* - 修饰符字母
- *Lo* - 其他字母
- *Nl* - 字母数字
- *Mn* - 非空白标识
- *Mc* - 含空白标识
- *Nd* - 十进制数字
- *Pc* - 连接标点
- *Other_ID_Start* - 在 PropList.txt 中显式定义的用于支持向下兼容的字符列表
- *Other_ID_Continue* - 同上

在解析时，所有标识符都会被转换为规范形式 NFKC；标识符的比较都是基于 NFKC。

列出 Unicode 15.1.0 中所有可用标识符字符的非规范 HTML 文件可在 https://www.unicode.org/Public/15.1.0/ucd/DerivedCoreProperties.txt 获取

2.3.1 关键字

以下标识符为保留字，或称 关键字，不可用于普通标识符。关键字的拼写必须与这里列出的完全一致:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

2.3.2 软关键字

Added in version 3.10.

某些标识符仅在特定上下文中被保留。它们被称为 软关键字。`match`, `case`, `type` 和 `_` 等标识符在特定上下文中具有关键字的语义，但这种区分是在解析器层级完成的，而不是在分词的时候。

作为软关键字，它们能够在用于相应语法的同时仍然保持与用作标识符名称的现有代码的兼容性。

`match`, `case` 和 `_` 是在*`match`* 语句中使用。`type` 是在*`type`* 语句中使用。

在 3.12 版本发生变更: `type` 现在是一个软关键字。

2.3.3 保留的标识符类

某些标识符类（除了关键字）具有特殊含义。这些类的命名模式以下划线字符开头，并以下划线结尾:

_*
: 不会被 `from module import *` 所导入。

_

在*match* 语句内部的 case 模式中，_ 是一个软关键字，它表示通配符。

在此之外，交互式解释器会将最后一次求值的结果放到变量 _ 中。（它与 print 等内置函数一起被存储于 builtins 模块。）

在其他地方，_ 是一个常规标识符。它常常被用来命名”特殊” 条目，但对 Python 本身来说毫无特殊之处。

> **备注**
>
> _ 常用于连接国际化文本；详见 gettext 模块文档。
>
> 它还经常被用来命名无需使用的变量。

__*__

系统定义的名称，通常简称为”dunder” 。这些名称由解释器及其实现（包括标准库）定义。现有系统定义名称相关的论述详见特殊方法名称 等章节。Python 未来版本中还将定义更多此类名称。任何情况下，任何不显式遵从 __*__ 名称的文档用法，都可能导致无警告提示的错误。

__*

类的私有名称。类定义时，此类名称以一种混合形式重写，以避免基类及派生类的” 私有” 属性之间产生名称冲突。详见标识符（名称）。

2.4 字面值

字面值是内置类型常量值的表示法。

2.4.1 字符串与字节串字面值

字符串字面值的词法定义如下：

```
stringliteral     ::=   [stringprefix](shortstring | longstring)
stringprefix      ::=   "r" | "u" | "R" | "U" | "f" | "F"
                        | "fr" | "Fr" | "fR" | "FR" | "rf" | "rF" | "Rf" | "RF"
shortstring       ::=   "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring        ::=   "'''" longstringitem* "'''" | '"""' longstringitem* '"""'
shortstringitem   ::=   shortstringchar | stringescapeseq
longstringitem    ::=   longstringchar | stringescapeseq
shortstringchar   ::=   <any source character except "\" or newline or the quote>
longstringchar    ::=   <any source character except "\">
stringescapeseq   ::=   "\" <any source character>
```

```
bytesliteral     ::=   bytesprefix(shortbytes | longbytes)
bytesprefix      ::=   "b" | "B" | "br" | "Br" | "bR" | "BR" | "rb" | "rB" | "Rb" | "RB"
shortbytes       ::=   "'" shortbytesitem* "'" | '"' shortbytesitem* '"'
longbytes        ::=   "'''" longbytesitem* "'''" | '"""' longbytesitem* '"""'
shortbytesitem   ::=   shortbyteschar | bytesescapeseq
longbytesitem    ::=   longbyteschar | bytesescapeseq
shortbyteschar   ::=   <any ASCII character except "\" or newline or the quote>
longbyteschar    ::=   <any ASCII character except "\">
bytesescapeseq   ::=   "\" <any ASCII character>
```

这些产生式未指明的一个句法限制是空白符不允许在 *stringprefix* 或 *bytesprefix* 与字面值的其余部分之间出现。源字符集是由编码格式声明来定义的；如果源文件没有给出编码格式声明则默认 UTF-8；参见编码声明 一节。

直白的说明：两种类型的字面值都可用成对的单引号 (') 或双引号 (") 括起来。它们还可以用成对的连续三个单引号或双引号括起来 (这通常被称为 *三重引号字符串*)。反斜杠 (\) 字符被用来给予普通的字符特殊含义例如 `n`，当用斜杠转义时 (\n) 表示' 换行'。它还可以被用来对具有特殊含义的字符进行转义，例如换行符、反斜杠本身或者引号等。请参阅下面的转义序列 查看示例。

字节串字面值要加前缀 `'b'` 或 `'B'`；生成的是类型 `bytes` 的实例，不是类型 `str` 的实例；字节串只能包含 ASCII 字符；字节串数值大于等于 128 时，必须用转义表示。

字符串和字节串字面值都可选择加前缀字母 `'r'` 或 `'R'`；这分别被称为 *原始字符串字面值* 和 *原始字节串字面值* 并会将反斜杠视为原本的字符字面值。因此，在原始字符串字面值中，`'\U'` 和 `'\u'` 转义符号不会被特殊对待。

Added in version 3.3: 新增原始字节串 `'rb'` 前缀，是 `'br'` 的同义词。

支持 unicode 字面值（`u'value'`）遗留代码，简化 Python 2.x 和 3.x 并行代码库的维护工作。详见 **PEP 414**。

前缀为 `'f'` 或 `'F'` 的字符串称为 *格式字符串*；详见*f 字符串*。`'f'` 可与 `'r'` 连用，但不能与 `'b'` 或 `'u'` 连用，因此，可以使用原始格式字符串，但不能使用格式字节串字面值。

三引号字面值可以包含未转义的换行和引号（原样保留），除了连在一起的，用于终止字面值的，未经转义的三个引号。(" 引号" 是启用字面值的字符，可以是 `'`，也可以是 `"`。)

转义序列

如未标注 `'r'` 或 `'R'` 前缀，字符串和字节串字面值中，转义序列以类似 C 标准的规则进行解释。可用的转义序列如下：

转义序列	含意	备注
\<newline>	忽略反斜杠与换行符	(1)
\\	反斜杠（\）	
\'	单引号（'）	
\"	双引号（"）	
\a	ASCII 响铃（BEL）	
\b	ASCII 退格符（BS）	
\f	ASCII 换页符（FF）	
\n	ASCII 换行符（LF）	
\r	ASCII 回车符（CR）	
\t	ASCII 水平制表符（TAB）	
\v	ASCII 垂直制表符（VT）	
ooo	八进制数 *ooo* 字符	(2,4)
\x*hh*	十六进制数 *hh* 字符	(3,4)

字符串字面值专用的转义序列：

转义序列	含意	备注
\N{*name*}	Unicode 数据库中名为 *name* 的字符	(5)
\u*xxxx*	16 位十六进制数 *xxxx* 码位的字符	(6)
\U*xxxxxxxx*	32 位 16 进制数 *xxxxxxxx* 码位的字符	(7)

注释：

(1) 可以在行尾添加一个反斜杠来忽略换行符：

```
>>> 'This string will not include \
... backslashes or newline characters.'
'This string will not include backslashes or newline characters.'
```

同样的效果也可以使用三重引号字符串，或者圆括号和字符串字面值拼接 来达成。

(2) 与 C 标准一致，接受最多三个八进制数字。

在 3.11 版本发生变更: 取值大于 `0o377` 的八进制数转义序列会产生 `DeprecationWarning`。

在 3.12 版本发生变更: 数值大于 `0o377` 的八进制转义符会产生 `SyntaxWarning`。在未来的 Python 版本中将最终改为 `SyntaxError`。

(3) 与 C 标准不同，必须为两个十六进制数字。

(4) 字节串字面值中，十六进制数和八进制数的转义码以相应数值代表每个字节。字符串字面值中，这些转义码以相应数值代表每个 Unicode 字符。

(5) 在 3.3 版本发生变更: 加入了对别名[1] 的支持。

(6) 必须为 4 个十六进制数码。

(7) 表示任意 Unicode 字符。必须为 8 个十六进制数码。

与 C 标准不同，无法识别的转义序列在字符串里原样保留，即，输出结果保留反斜杠。(调试时，这种方式很有用：输错转义序列时，更容易在输出结果中识别错误。) 注意，在字节串字面值内，字符串字面值专用的转义序列属于无法识别的转义序列。

在 3.6 版本发生变更: 不可识别的转义序列会产生 `DeprecationWarning`。

在 3.12 版本发生变更: 不可识别的转义序列会产生 `SyntaxWarning`。在未来的 Python 版本中将最终改为 `SyntaxError`。

即使在原始字面值中，引号也可以用反斜杠转义，但反斜杠会保留在输出结果里；例如 `r"\""` 是由两个字符组成的有效字符串字面值：反斜杠和双引号；`r"\"` 则不是有效字符串字面值（原始字符串也不能以奇数个反斜杠结尾）。尤其是，原始字面值不能以单个反斜杠结尾（反斜杠会转义其后的引号）。还要注意，反斜杠加换行在字面值中被解释为两个字符，而 不是连续行。

2.4.2 字符串字面值合并

以空白符分隔的多个相邻字符串或字节串字面值，可用不同引号标注，等同于合并操作。因此，`"hello" 'world'` 等价于 `"helloworld"`。此功能不需要反斜杠，即可将长字符串分为多个物理行，还可以为不同部分的字符串添加注释，例如：

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

注意，此功能在句法层面定义，在编译时实现。在运行时，合并字符串表达式必须使用'+' 运算符。还要注意，字面值合并可以为每个部分应用不同的引号风格（甚至混用原始字符串和三引号字符串），格式字符串字面值也可以与纯字符串字面值合并。

2.4.3 f 字符串

Added in version 3.6.

格式字符串字面值或称 *f-string* 是标注了 `'f'` 或 `'F'` 前缀的字符串字面值。这种字符串可包含替换字段，即以 `{}` 标注的表达式。其他字符串字面值只是常量，格式字符串字面值则是可在运行时求值的表达式。

除非字面值标记为原始字符串，否则，与在普通字符串字面值中一样，转义序列也会被解码。解码后，用于字符串内容的语法如下：

```
f_string            ::=  (literal_char | "{{" | "}}" | replacement_field)*
replacement_field   ::=  "{" f_expression ["="] ["!" conversion] [":" format_spec] "}"
f_expression        ::=  (conditional_expression | "*" or_expr)
                         ("," conditional_expression | "," "*" or_expr)* [","]
                         | yield_expression
conversion          ::=  "s" | "r" | "a"
```

[1] https://www.unicode.org/Public/15.1.0/ucd/NameAliases.txt

```
format_spec          ::=   (literal_char | replacement_field)*
literal_char         ::=   <any code point except "{", "}" or NULL>
```

双花括号 `'{{'` 或 `'}}'` 被替换为单花括号，花括号外的字符串仍按字面值处理。单左花括号 `'{'` 标记以 Python 表达式开头的替换字段。在表达式后加等于号 `'='`，可在求值后，同时显示表达式文本及其结果（用于调试）。随后是用叹号 `'!'` 标记的转换字段。还可以在冒号 `':'` 后附加格式说明符。替换字段以右花括号 `'}'` 为结尾。

格式化字符串字面值中的表达式会与用圆括号包围的常规 Python 表达式一样处理，但有少量例外。空表达式是不被允许的，而 *lambda* 和赋值表达式 `:=` 都必须显式地用括号包围。每个表达式都将在格式化字符串字面值出现的上下文中按从左到右的顺序进行求值。替换表达式可在单引号和三引号 f-字符串中包含换行符并可包含注释。替换字段内 `#` 后面的所有内容都是注释（即使结尾花括号和引号也是）。在这种情况下，替换字段必须在另一行中结束。

```
>>> f"abc{a # This is a comment }"
... + 3}"
'abc5'
```

在 3.7 版本发生变更: Python 3.7 以前，因为实现的问题，不允许在格式字符串字面值表达式中使用 *await* 表达式与包含 *async for* 子句的推导式。

在 3.12 版本发生变更: 在 Python 3.12 之前，不允许在 f-字符串的替换字段中使用注释。

表达式里含等号 `'='` 时，输出内容包括表达式文本、`'='` 、求值结果。输出内容可以保留表达式中左花括号 `'{'` 后，及 `'='` 后的空格。没有指定格式时，`'='` 默认调用表达式的 `repr()`。指定了格式时，默认调用表达式的 `str()`，除非声明了转换字段 `'!r'`。

Added in version 3.8: 等号 `'='`。

指定了转换符时，表达式求值的结果会先转换，再格式化。转换符 `'!s'` 调用 `str()` 转换求值结果，`'!r'` 调用 `repr()`，`'!a'` 调用 `ascii()`。

然后使用 `format()` 协议对结果进行格式化。格式说明符将传给表达式或转换结果的 *__format__()* 方法。如果省略格式说明符则将传入空字符串。格式化后的结果将包括在整个字符串的最终值中。

最高层级的格式说明符可以包括嵌套的替换字段。这些嵌套字段也可以包括它们自己的转换字段和 格式说明符，但是不可再包括更深层嵌套的替换字段。这里的 格式说明符微语言与 `str.format()` 方法所使用的相同。

格式化字符串字面值可以拼接，但是一个替换字段不能拆分到多个字面值。

格式字符串字面值示例如下：

```
>>> name = "Fred"
>>> f"He said his name is {name!r}."
"He said his name is 'Fred'."
>>> f"He said his name is {repr(name)}."  # repr() is equivalent to !r
"He said his name is 'Fred'."
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}"  # nested fields
'result:      12.35'
>>> today = datetime(year=2017, month=1, day=27)
>>> f"{today:%B %d, %Y}"  # using date format specifier
'January 27, 2017'
>>> f"{today=:%B %d, %Y}" # using date format specifier and debugging
'today=January 27, 2017'
>>> number = 1024
>>> f"{number:#0x}"  # using integer format specifier
'0x400'
>>> foo = "bar"
>>> f"{ foo = }" # preserves whitespace
" foo = 'bar'"
```

(续下页)

(接上页)

```
>>> line = "The mill's closed"
>>> f"{line = }"
'line = "The mill\'s closed"'
>>> f"{line = :20}"
"line = The mill's closed   "
>>> f"{line = !r:20}"
'line = "The mill\'s closed" '
```

允许在替换字段中重用外层 f-字符串的引号类型:

```
>>> a = dict(x=2)
>>> f"abc {a["x"]} def"
'abc 2 def'
```

在 3.12 版本发生变更: 在 Python 3.12 之前不允许在替换字段中重用与外层 f-字符串相同的引号类型。

替换字段中也允许使用反斜杠并会以与在其他场景下相同的方式求值:

```
>>> a = ["a", "b", "c"]
>>> print(f"List a contains:\n{"\n".join(a)}")
List a contains:
a
b
c
```

在 3.12 版本发生变更: 在 Python 3.12 之前，f-字符串的替换字段内不允许使用反斜杠。

即便未包含表达式，格式字符串字面值也不能用作文档字符串。

```
>>> def foo():
...     f"Not a docstring"
...
>>> foo.__doc__ is None
True
```

参阅 **PEP 498**，了解格式字符串字面值的提案，以及与格式字符串机制相关的 `str.format()`。

2.4.4 数值字面值

数值字面值有三种类型：整数、浮点数、虚数。没有复数字面值（复数由实数加虚数构成）。

注意，数值字面值不含正负号；实际上，`-1` 等负数是由一元运算符'-' 和字面值 `1` 合成的。

2.4.5 整数字面值

整数字面值词法定义如下：

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

整数字面值的长度没有限制，能一直大到占满可用内存。

确定数值时，会忽略字面值中的下划线。下划线只是为了分组数字，让数字更易读。下划线可在数字之间，也可在 `0x` 等基数说明符后。

注意，除了 0 以外，十进制数字的开头不允许有零。以免与 Python 3.0 版之前使用的 C 样式八进制字面值混淆。

整数字面值示例如下：

```
7     2147483647                        0o177    0b100110111
3     79228162514264337593543950336     0o377    0xdeadbeef
      100_000_000_000                   0b_1110_0101
```

在 3.6 版本发生变更: 现已支持在字面值中，用下划线分组数字。

2.4.6 浮点数字面值

浮点数字面值的词法定义如下所述：

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

注意，解析时，整数和指数部分总以 10 为基数。例如，`077e010` 是合法的，表示的数值与 `77e10` 相同。浮点数字面值的支持范围取决于具体实现。整数字面值支持用下划线分组数字。

一些浮点数字面值的示例如下：

```
3.14    10.    .001    1e100    3.14e-10    0e0    3.14_15_93
```

在 3.6 版本发生变更: 现已支持在字面值中，用下划线分组数字。

2.4.7 虚数字面值

虚数字面值词法定义如下：

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

虚数字面值生成实部为 0.0 的复数。复数由一对浮点数表示，它们的取值范围相同。创建实部不为零的复数，则需添加浮点数，例如 `(3+4j)`。虚数字面值示例如下：

```
3.14j   10.j    10j     .001j   1e100j   3.14e-10j   3.14_15_93j
```

2.5 运算符

运算符如下所示：

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

2.6 分隔符

以下形符在语法中为分隔符：

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

句点也可以用于浮点数和虚数字面值。三个连续句点表示省略符。列表后半部分是增强赋值操作符，用作词法分隔符，但也可以执行运算。

以下 ASCII 字符具有特殊含义，对词法分析器有重要意义：

```
'       "       #       \
```

以下 ASCII 字符不用于 Python。在字符串字面值或注释外使用时，将直接报错：

```
$       ?       `
```

备注

CHAPTER 3

数据模型

3.1 对象、值与类型

对象是 Python 中对数据的抽象。Python 程序中的所有数据都是由对象或对象间关系来表示的。(从某种意义上说，按照冯 · 诺依曼的“存储程序计算机”模型，代码本身也是由对象来表示的。)

每个对象都有相应的标识号、类型和值。一个对象被创建后它的 *标识号* 就绝不会改变；你可以将其理解为该对象在内存中的地址。*is* 运算符比较两个对象的标识号是否相同；`id()` 函数返回一个代表其标识号的整数。

CPython 实现细节： 在 CPython 中，`id(x)` 就是存放 `x` 的内存的地址。

对象的类型决定该对象所支持的操作 (例如” 对象是否有长度属性？”) 并且定义了该类型的对象可能的取值。`type()` 函数能返回一个对象的类型 (类型本身也是对象)。与编号一样，一个对象的 *类型* 也是不可改变的。[1]

有些对象的 *值* 可以改变。值可以改变的对象被称为 *可变对象*；值不可以改变的对象就被称为 *不可变对象*。(一个不可变容器对象如果包含对可变对象的引用，当后者的值改变时，前者的值也会改变；但是该容器仍属于不可变对象，因为它所包含的对象集是不会改变的。因此，不可变并不严格等同于值不能改变，实际含义要更微妙。) 一个对象的可变性是由其类型决定的；例如，数字、字符串和元组是不可变的，而字典和列表是可变的。

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 实现细节： CPython 目前使用带有 (可选) 延迟检测循环链接垃圾的引用计数方案，会在对象不可访问时立即回收其中的大部分，但不保证回收包含循环引用的垃圾。请查看 `gc` 模块的文档了解如何控制循环垃圾的收集相关信息。其他实现会有不同的行为方式，CPython 现有方式也可能改变。不要依赖不可访问对象的立即终结机制 (所以你应当总是显式地关闭文件)。

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。

有些对象包含对“外部”资源如打开的文件或窗口的引用。当对象被作为垃圾回收时这些资源也应该会被释放，但由于垃圾回收并不确保发生，这些对象还提供了明确地释放外部资源的操作，通常为一个 `close()` 方法。强烈推荐在程序中显式关闭此类对象。*try*...*finally* 语句和 *with* 语句提供了进行此种操作的更便捷方式。

[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。

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。甚至对象标识号的重要性也在某种程度上受到影响：对于不可变类型，计算新值的操作实际上可能返回一个指向具有相同类型和值的任何现存对象的引用，而对于可变对象来说这是不允许的。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。这是因为 `int` 是不可变对象，因此对 `1` 的引用可以被重用。此行为依赖于所使用的具体实现，因此不应该依赖它，而在使用对象标识测试时需要注意。不过，在 `c = []; d = []` 之后，*c* 和 *d* 保证会指向两个不同的、独特的、新创建的空列表。(注意 `e = f = []` 会将 同一个对象同时赋值给 *e* 和 *f*。)

3.2 标准类型层级结构

以下是 Python 内置类型的列表。扩展模块 (具体实现会以 C, Java 或其他语言编写) 可以定义更多的类型。未来版本的 Python 可能会加入更多的类型 (例如有理数、高效存储的整型数组等等)，不过新增类型往往都是通过标准库来提供的。

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。

3.2.1 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。

3.2.2 NotImplemented

此类型只有一种取值。是一个具有该值的单独对象。此对象通过内置名称 `NotImplemented` 访问。数值方法和丰富比较方法如未实现指定运算符表示的运算则应返回该值。(解释器会根据具体运算符继续尝试反向运算或其他回退操作。) 它不应被解读为布尔值。

详情参见 implementing-the-arithmetic-operations。

在 3.9 版本发生变更: 对 `NotImplemented` 求布尔值的操作已被弃用。虽然它目前会被求解为真值，但将同时发出 `DeprecationWarning`。它将在未来的 Python 版本中引发 `TypeError`。

3.2.3 Ellipsis

此类型只有一种取值。是一个具有此值的单独对象。此对象通过字面值 `...` 或内置名称 `Ellipsis` 访问。它的逻辑值为真。

3.2.4 `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。

数字类的字符串表示形式，由*__repr__()* 和*__str__()* 算出，具有以下特征属性:

- 它们是有效的数字字面值，当被传给它们的类构造器时，将会产生具有原数字值的对象。
- 表示形式会在可能的情况下采用 10 进制。
- 开头的零，除小数点前可能存在的单个零之外，将不会被显示。
- 末尾的零，除小数点后可能存在的单个零之外，将不会被显示。
- 正负号仅在当数字为负值时会被显示。

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)。

> **备注**
>
> 整型数表示规则的目的是在涉及负整型数的变换和掩码运算时提供最为合理的解释。

整型数可细分为两种类型:

整型 (int)
: 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。

布尔型 (bool)
: 此类对象表示逻辑值 False 和 True。代表 `False` 和 `True` 值的两个对象是唯二的布尔对象。布尔类型是整型的子类型，两个布尔值在各种场合的行为分别类似于数值 0 和 1，例外情况只有在转换为字符串时分别返回字符串 `"False"` 或 `"True"`。

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 (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。

3.2.5 序列

这些代表以非负数为索引的有限有序集合。内置函数 `len()` 将返回序列的项数。当序列的长度为 *n* 时，索引集合将包含数字 0, 1, ..., *n*-1。`a[i]` 是选择序列 *a* 中的第 *i* 项。某些序列，包括内置的序列，可通过加上序列长度来解读负下标值。例如，`a[-2]` 等价于 `a[n-2]`，即长度为 `n` 的 a 序列的倒数第二项。

序列还支持切片: `a[i:j]` 是选择索引为 *k* 使得 *i* <= *k* < *j* 的所有条目。当用作表达式时，切片就是一个相同类型的新序列。以上有关负索引的注释也适用于切片位置的负值。

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
: 字符串是由代表 Unicode 码位的值组成的序列。取值范围在 `U+0000 - U+10FFFF` 之内的所有码位都可在字符串中使用。Python 没有 `char` 类型；而是将字符串中的每个码位表示为一个长度为 `1` 的字符串对象。内置函数 `ord()` 可将一个码位由字符串形式转换为取值范围在 `0 - 10FFFF` 之内的整数；`chr()` 可将一个取值范围在 `0 - 10FFFF` 之内的整数转换为长度为 `1` 的对应字符串对象。`str.encode()` 可以使用给定的文本编码格式将 `str` 转换为 `bytes`，而 `bytes.decode()` 则可以被用来实现相反的解码操作。

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。

字节串
字节串对象是不是变的数组。其中的条目是 8 比特位的字节，以取值范围 0 <= x < 256 内的整数表示。字节串字面值 (如 `b'abc'`) 和内置的 `bytes()` 构造器可被用来创建字节串对象。并且，字节串对象还可通过 `decode()` 方法被解码为字符串。

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。

> **备注**
>
> `collections` 和 `array` 模块提供了可变序列类型的更多例子。

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
字节数组对象属于可变数组。可以通过内置的 `bytearray()` 构造器来创建。除了是可变的 (因而也是不可哈希的)，在其他方面字节数组提供的接口和功能都与不可变的 `bytes` 对象一致。

3.2.6 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。

3.2.7 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。

字典会保留插入顺序，这意味着键将以它们被添加的顺序在字典中依次产生。替换某个现有的键不会改变其顺序，但是移除某个键再重新插入则会将其添加到末尾而不会保留其原有位置。

字典是可变对象；它们可通过 {} 标注方式来创建（参见字典显示 一节）。

扩展模块 dbm.ndbm 和 dbm.gnu 提供了额外的映射类型示例，collections 模块也是如此。

在 3.7 版本发生变更: 在 Python 3.6 版之前字典不会保留插入顺序。在 CPython 3.6 中插入顺序会被保留，但这在当时被当作是一个实现细节而非确定的语言特性。

3.2.8 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。

特殊的只读属性

属性	含意
function.**__globals__**	对存放该函数中全局变量 的 字典的引用——函数定义所在模块的全局命名空间。
function.**__closure__**	None 或单元的 tuple，其中包含了名称在函数的代码对象 的*co_freevars* 中对指定名称的绑定。 单元对象具有 cell_contents 属性。这可被用来获取以及设置单元的值。

特殊的可写属性

这些属性大多会检查赋值的类型：

属性	含意
function.__doc__	函数的文档字符串，或者如果不可用则为 None。
function.__name__	函数的名称。另请参阅: __name__ 属性。
function.__qualname__	函数的*qualified name*。另请参阅: __qualname__ 属性。 Added in version 3.3.
function.__module__	该函数所属模块的名称，没有则为 None。
function.__defaults__	由具有默认值的形参的默认*parameter* 值组成的 tuple，或者如果无任何形参具有默认值则为 None。
function.__code__	代表已编译的函数体的代码对象。
function.__dict__	命名空间支持任意函数属性。另请参阅: *__dict__* 属性。
function.__annotations__	包含形参 标注的 字典。该字典的键是形参名称，如存在返回标注则将包含 'return' 键。另请参阅: annotations-howto。
function.__kwdefaults__	包含仅限关键字形参 默认值的 字典。
function.__type_params__	包含泛型函数 类型形参 的 tuple。 Added in version 3.12.

函数对象也支持获取和设置任意属性，举例来说，这可被用于将元数据关联到函数。通常使用带点号的属性标注来获取和设置这样的属性。

CPython 实现细节： CPython 目前的实现仅支持用户自定义函数上的函数属性。未来可能会支持内置函数 上的函数属性。

有关函数定义的额外信息可以从其代码对象 中提取（可通过*__code__* 属性来访问）。

实例方法

实例方法用于结合类、类实例和任何可调用对象 (通常为用户定义函数)。

特殊的只读属性:

method.**__self__**	指向方法所绑定 的类实例对象。
method.**__func__**	指向原本的函数对象
method.**__doc__**	方法的文档（等同于*method.__func__.__doc__*）。如果原始函数具有文档字符串则为一个字符串，否则为 None。
method.**__name__**	方法名称（与*method.__func__.__name__* 相同）
method.**__module__**	方法定义所在模块的名称，如不可用则为 None。

方法还支持读取（但不能设置）下层函数对象 的任意函数属性。

用户自定义方法对象可在获取一个类的属性（可能是通过该类的实例）时被创建，如果该属性是一个用户自定义函数对象 或 classmethod 对象的话。

当通过从类的实例获取一个用户自定义函数对象 的方式创建一个实例方法对象时，该方法对象的*__self__* 属性即为该实例，而该方法对象将被称作已 绑定。该新建方法的*__func__* 属性将是原来的函数对象。

当通过从类或实例获取一个 classmethod 对象的方式创建一个实例方法对象时，该对象的*__self__* 属性即为该类本身，而其*__func__* 属性将是类方法对应的下层函数对象。

当一个实例方法被调用时，会调用对应的下层函数 (*__func__*)，并将类实例 (*__self__*) 插入参数列表的开头。例如，当 C 是一个包含 f() 函数定义的类，而 x 是 C 的一个实例，则调用 x.f(1) 就等价于调用 C.f(x, 1)。

当一个实例方法对象是派生自一个 classmethod 对象时，保存在*__self__* 中的“类实例”实际上会是该类本身，因此无论是调用 x.f(1) 还是 C.f(1) 都等同于调用 f(C,1)，其中 f 为对应的下层函数。

需要重点关注的是作为类实例的属性的用户自定义函数不会被转换为绑定方法；这 只会在函数是类的属性时才会发生。

生成器函数

一个使用*yield* 语句（见*yield 语句* 章节）的函数或方法被称为 生成器函数。这样的函数在被调用时，总是返回一个可以执行该函数体的*iterator* 对象：调用该迭代器的 iterator.__next__() 方法将导致这个函数一直运行到它使用 yield 语句提供一个值。当这个函数执行*return* 语句或到达函数体末尾时，将引发 StopIteration 异常并且该迭代器将到达所返回的值集合的末尾。

协程函数

使用*async def* 来定义的函数或方法就被称为 协程函数。这样的函数在被调用时会返回一个*coroutine* 对象。它可能包含*await* 表达式以及*async with* 和*async for* 语句。详情可参见协程对象 一节。

异步生成器函数

使用*async def* 来定义并使用了*yield* 语句的函数或方法被称为 异步生成器函数。这样的函数在被调用时，将返回一个*asynchronous iterator* 对象，该对象可在*async for* 语句中被用来执行函数体。

调用异步迭代器的*aiterator.__anext__* 方法将返回一个*awaitable*，此对象会在被等待时执行直到使用*yield* 产生一个值。当函数执行到空的*return* 语句或函数末尾时，将会引发 StopAsyncIteration 异常并且异步迭代器也将到达要产生的值集合的末尾。

内置函数

内置函数是针对特定 C 函数的包装器。内置函数的例子包括 `len()` 和 `math.sin()` 等 (`math` 是一个标准内置模块)。参数的数量和类型是由 C 函数确定的。特殊的只读属性:

- `__doc__` 是函数的文档字符串，或者如果不可用则为 `None`。参见*function.__doc__*。
- `__name__` 是函数的名称。参见*function.__name__*。
- `__self__` 被设为 `None` (但请参见下一项)。
- `__module__` 是函数定义所在模块的名称，或者如果不可用则为 `None`。参见*function.__module__*。

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转入 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* 是一个列表对象。在此示例中，特殊的只读属性 `__self__` 会被设为 *alist* 所标记的对象。(该属性的语义与其他实例方法 的相同。)

类

类是可调用对象。这些对象通常是用作创建自身实例的 “工厂”，但类也可以有重载*__new__()* 的变体类型。调用的参数会传递给 `__new__()`，并且在通常情况下，也会传递给*__init__()* 来初始化新的实例。

类实例

任意类的实例可以通过在其所属类中定义*__call__()* 方法变成可调用对象。

3.2.9 模块

模块是 Python 代码的基本组织单元，由导入系统 创建，它或是通过*import* 语句，或是通过调用 `importlib.import_module()` 和内置的 `__import__()` 等函数发起调用。模块对象具有通过 字典对象实现的命名空间 (就是被定义在模块中的函数的*__globals__* 属性所引用的字典)。属性引用将被转换为在该字典中的查找操作，例如 `m.x` 就等价于 `m.__dict__["x"]`。模块对象不包含用于初始化模块的代码对象 (因为初始化完成后已不再需要它)。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`。

模块对象上与导入相关的属性

Module objects have the following attributes that relate to the *import system*. When a module is created using the machinery associated with the import system, these attributes are filled in based on the module's *spec*, before the *loader* executes and loads the module.

To create a module dynamically rather than using the import system, it's recommended to use `importlib.util.module_from_spec()`, which will set the various import-controlled attributes to appropriate values. It's also possible to use the `types.ModuleType` constructor to create modules directly, but this technique is more error-prone, as most attributes must be manually set on the module object after it has been created when using this approach.

小心

With the exception of *__name__*, it is **strongly** recommended that you rely on *__spec__* and its attributes instead of any of the other individual attributes listed in this subsection. Note that updating an attribute on `__spec__` will not update the corresponding attribute on the module itself:

```
>>> import typing
>>> typing.__name__, typing.__spec__.name
('typing', 'typing')
>>> typing.__spec__.name = 'spelling'
>>> typing.__name__, typing.__spec__.name
('typing', 'spelling')
>>> typing.__name__ = 'keyboard_smashing'
>>> typing.__name__, typing.__spec__.name
('keyboard_smashing', 'spelling')
```

module.**__name__**

用于在导入系统中唯一地标识模块的名称。对于直接执行的模块，这将被设为 "__main__"。

该属性必须被设为模块的完整限定名称。它应当与 module.__spec__.name 的值相匹配。

module.**__spec__**

模块与导入系统相关联的状态的记录。

Set to the module spec that was used when importing the module. See 模块规格说明 for more details.

Added in version 3.4.

module.**__package__**

The *package* a module belongs to.

If the module is top-level (that is, not a part of any specific package) then the attribute should be set to '' (the empty string). Otherwise, it should be set to the name of the module's package (which can be equal to *module.__name__* if the module itself is a package). See **PEP 366** for further details.

This attribute is used instead of *__name__* to calculate explicit relative imports for main modules. It defaults to None for modules created dynamically using the types.ModuleType constructor; use importlib.util.module_from_spec() instead to ensure the attribute is set to a str.

It is **strongly** recommended that you use module.__spec__.parent instead of module.__package__. *__package__* is now only used as a fallback if __spec__.parent is not set, and this fallback path is deprecated.

在 3.4 版本发生变更: This attribute now defaults to None for modules created dynamically using the types.ModuleType constructor. Previously the attribute was optional.

在 3.6 版本发生变更: The value of __package__ is expected to be the same as __spec__.parent. *__package__* is now only used as a fallback during import resolution if __spec__.parent is not defined.

在 3.10 版本发生变更: ImportWarning is raised if an import resolution falls back to __package__ instead of __spec__.parent.

在 3.12 版本发生变更: Raise DeprecationWarning instead of ImportWarning when falling back to __package__ during import resolution.

Deprecated since version 3.13, will be removed in version 3.15: __package__ will cease to be set or taken into consideration by the import system or standard library.

module.**__loader__**

The *loader* object that the import machinery used to load the module.

This attribute is mostly useful for introspection, but can be used for additional loader-specific functionality, for example getting data associated with a loader.

__loader__ defaults to None for modules created dynamically using the types.ModuleType constructor; use importlib.util.module_from_spec() instead to ensure the attribute is set to a *loader* object.

It is **strongly** recommended that you use module.__spec__.loader instead of module.__loader__.

在 3.4 版本发生变更: This attribute now defaults to None for modules created dynamically using the types.ModuleType constructor. Previously the attribute was optional.

Deprecated since version 3.12, will be removed in version 3.14: Setting __loader__ on a module while failing to set __spec__.loader is deprecated. In Python 3.14, __loader__ will cease to be set or taken into consideration by the import system or the standard library.

module.**__path__**

A (possibly empty) *sequence* of strings enumerating the locations where the package's submodules will be found. Non-package modules should not have a __path__ attribute. See 模块的 *__path__* 属性 for more details.

It is **strongly** recommended that you use `module.__spec__.submodule_search_locations` instead of `module.__path__`.

`module.__file__`

`module.__cached__`

`__file__` and `__cached__` are both optional attributes that may or may not be set. Both attributes should be a `str` when they are available.

`__file__` indicates the pathname of the file from which the module was loaded (if loaded from a file), or the pathname of the shared library file for extension modules loaded dynamically from a shared library. It might be missing for certain types of modules, such as C modules that are statically linked into the interpreter, and the *import system* may opt to leave it unset if it has no semantic meaning (for example, a module loaded from a database).

If `__file__` is set then the `__cached__` attribute might also be set, which is the path to any compiled version of the code (for example, a byte-compiled file). The file does not need to exist to set this attribute; the path can simply point to where the compiled file *would* exist (see **PEP 3147**).

Note that `__cached__` may be set even if `__file__` is not set. However, that scenario is quite atypical. Ultimately, the *loader* is what makes use of the module spec provided by the *finder* (from which `__file__` and `__cached__` are derived). So if a loader can load from a cached module but otherwise does not load from a file, that atypical scenario may be appropriate.

It is **strongly** recommended that you use `module.__spec__.cached` instead of `module.__cached__`.

Deprecated since version 3.13, will be removed in version 3.15: Setting `__cached__` on a module while failing to set `__spec__.cached` is deprecated. In Python 3.15, `__cached__` will cease to be set or taken into consideration by the import system or standard library.

Other writable attributes on module objects

As well as the import-related attributes listed above, module objects also have the following writable attributes:

`module.__doc__`

The module's documentation string, or `None` if unavailable. See also: `__doc__ attributes`.

`module.__annotations__`

包含在模块体执行期间收集的变量标注 的字典。有关使用*__annotations__* 的最佳实践，请参阅 annotations-howto。

模块字典

模块对象还具有以下特殊的只读属性：Module objects also have the following special read-only attribute:

`module.__dict__`

The module's namespace as a dictionary object. Uniquely among the attributes listed here, `__dict__` cannot be accessed as a global variable from within a module; it can only be accessed as an attribute on module objects.

CPython 实现细节： 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。

3.2.10 自定义类

自定义类这种类型一般是通过类定义来创建 (参见类定义 一节)。每个类都有一个通过字典对象实现的命名空间。类属性引用会被转化为在此字典中查找，例如，`C.x` 会被转化为 `C.__dict__["x"]` (不过也存在一些钩子对象允许其他定位属性的方式)。当未在其中找到某个属性名称时，会继续在基类中查找。这种基类搜索使用 C3 方法解析顺序，即使存在'钻石形'继承结构既有多条继承路径连到一个共同祖先也能保持正确的行为。有关 Python 使用的 C3 MRO 的详情可在 python_2.3_mro 查看。

当一个类属性引用 (假设类名为 `C`) 会产生一个类方法对象时，它将转化为一个*__self__* 属性为 `C` 的实例方法对象。当它会产生一个 `staticmethod` 对象时，它将转换为该静态方法对象所包装的对象。有关有类的*__dict__* 实际包含内容以外获取属性的其他方式请参阅实现描述器 一节。

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)。

特殊属性

属性	含意
type.**__name__**	类的名称。另请参阅: __name__ 属性。
type.**__qualname__**	类的*qualified name*。另请参阅: __qualname__ 属性。
type.**__module__**	类定义所在模块的名称。
type.**__dict__**	一个提供类的命名空间的只读视图的 映射代理。另请参阅: *__dict__ 属性*。
type.**__bases__**	一个包含类的基类的 tuple，对于定义为 class X(A, B, C) 的类，X.__bases__ 将等于 (A, B, C)。
type.**__doc__**	类的文档字符串，如果未定义则为 None。不会被子类继承。if undefined. Not inherited by subclasses.
type.**__annotations__**	A dictionary containing *variable annotations* collected during class body execution. For best practices on working with __annotations__, please see annotations-howto. **小心** Accessing the __annotations__ attribute of a class object directly may yield incorrect results in the presence of metaclasses. In addition, the attribute may not exist for some classes. Use inspect.get_annotations() to retrieve class annotations safely.
type.**__type_params__**	A tuple containing the *type parameters* of a *generic class*. Added in version 3.12.
type.**__static_attributes__**	A tuple containing names of attributes of this class which are assigned through self.X from any function in its body. Added in version 3.13.
type.**__firstlineno__**	The line number of the first line of the class definition, including decorators. Setting the __module__ attribute removes the __firstlineno__ item from the type's dictionary. Added in version 3.13.
type.**__mro__**	The tuple of classes that are considered when looking for base classes during method resolution.

Special methods

In addition to the special attributes described above, all Python classes also have the following two methods available:

type.**mro**()
: This method can be overridden by a metaclass to customize the method resolution order for its instances. It is called at class instantiation, and its result is stored in *__mro__*.

type.**__subclasses__**()
: Each class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive. The list is in definition order. Example:

```
>>> class A: pass
>>> class B(A): pass
>>> A.__subclasses__()
[<class 'B'>]
```

3.2.11 类实例

类实例可通过调用类对象来创建（见上文）。每个类实例都有通过一个字典对象实现的独立命名空间，属性引用会首先在此字典中进行查找。当未在其中发现某个属性，而实例对应的类中有该属性时，会继续在类属性中查找。如果找到的类属性是一个用户自定义函数对象，它会被转化为实例方法对象，其*__self__*属性即该实例。静态方法和类方法对象也会被转化；参见上文的“类”小节。要了解其他通过类实例来获取相应类属性的方式请参阅实现描述器 小节，这样得到的属性可能与实际存放在类的*__dict__*中的对象不同。如果未找到类属性，而对象所属的类具有*__getattr__()*方法，则会调用该方法来满足查找要求。

属性赋值和删除会更新实例的字典，但绝不会更新类的字典。如果类具有*__setattr__()*或*__delattr__()*方法，则将调用该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

object.**__class__**
: 类实例所属的类。

object.**__dict__**
: 一个用于存储对象的（可写）属性的字典或其他映射对象。并非所有实例都具有 `__dict__` 属性；请参阅*__slots__*章节了解详情。

3.2.12 I/O 对象 (或称文件对象)

file object 表示一个打开的文件。有多种快捷方式可用来创建文件对象: `open()` 内置函数，以及 `os.popen()`, `os.fdopen()` 和 socket 对象的 `makefile()` 方法 (还可能使用某些扩展模块所提供的其他函数或方法)。

`sys.stdin`, `sys.stdout` 和 `sys.stderr` 会初始化为对应于解释器标准输入、输出和错误流的文件对象；它们都会以文本模式打开，因此都遵循 `io.TextIOBase` 抽象类所定义的接口。

3.2.13 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)。

特殊的只读属性

codeobject.**co_name**	函数名
codeobject.**co_qualname**	完整限定函数名 Added in version 3.11.
codeobject.**co_argcount**	函数的位置形参 的总数（包括仅限位置形参和具有默认值的形参）
codeobject.**co_posonlyargcount**	函数的仅限位置形参 的总数（包括具有默认值的参数）
codeobject.**co_kwonlyargcount**	函数的仅限关键字形参 的数量（包括具有默认值的参数）
codeobject.**co_nlocals**	函数使用的局部变量 的数量（包括形参）
codeobject.**co_varnames**	一个 tuple，其中包含函数中局部变量的名称（从形参名称开始）
codeobject.**co_cellvars**	包含被函数内至少一个*nested scope* 所引用的局部变量 的名称的 tuple。
codeobject.**co_freevars**	A tuple containing the names of *free (closure) variables* that a *nested scope* references in an outer scope. See also *function.__closure__*. Note: references to global and builtin names are *not* included.
codeobject.**co_code**	一个表示函数中的*bytecode* 指令序列的字符串
codeobject.**co_consts**	一个包含函数中的*bytecode* 所使用的字面值的 tuple
codeobject.**co_names**	一个包含函数中的*bytecode* 所使用的名称的 tuple
codeobject.**co_filename**	被编译代码所在文件的名称
codeobject.**co_firstlineno**	函数第一行所对应的行号
codeobject.**co_lnotab**	一个编码了从*bytecode* 偏移量到行号的映射的字符串。要获取更多细节，请查看解释器的源代码。 自 3.12 版本弃用: 代码对象的这个属性已被弃用，并可能在 Python 3.14 中移除。
codeobject.**co_stacksize**	需要的代码对象栈大小
codeobject.**co_flags**	用于对一系列解释器旗标进行编码的 整数。

以下是针对*co_flags* 定义的旗标位：如果函数使用 *arguments 语法来接受任意数量的位置参数则设置

0x04 位；如果函数使用 **keywords 语法来接受任意数量的关键字参数则设置 0x08 位；如果函数是一个生成器则设置 0x20 位。请参阅 inspect-module-co-flags 可能出现的每个旗标的语义详情。

未来特性声明 (from __future__ import division) 也使用*co_flags* 中的位来提示代码对象的编译是否启用了特定的特性：如果函数编译时启用了未来除法特性则将设置 0x2000 位；在更早的 Python 版本中则会使用 0x10 和 0x1000 位。.

co_flags 中的其他位被保留供内部使用。

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

代码对象的方法

codeobject.**co_positions**()

返回一个包含代码对象中每条*bytecode* 指令的源代码位置的可迭代对象。

此迭代器返回包含 (start_line, end_line, start_column, end_column) 的 tuple。其中第 *i* 个元组冲锋衣官方编译为第 *i* 个代码单元的源代码的位置。列信息是给定源代码行从 0 开始索引的 utf-8 字节偏移量。

此位置信息可能会丢失。可能发生这种情况下非详尽列表如下:

- 附带 -X no_debug_ranges 运行解释器。
- 在使用 -X no_debug_ranges 时加载一个已编译的 pyc 文件。
- 与人工指令相对应的位置元组。
- 由于具体实现专属的限制而无法表示的行号和列号。

当发生此情况时，元组的部分或全部元素可以为 None。

Added in version 3.11.

> **备注**
>
> 此特性需要在代码对象中存储列位置，这可能会导致编译的 which may result in a small increase of disk usage of compiled Python 文件占用的磁盘空间或解释器占用的内存略有增加。要避免存储额外信息和/或取消打印额外的回溯信息，可以使用 -X no_debug_ranges 命令行旗标或 PYTHONNODEBUGRANGES 环境变量。

codeobject.**co_lines**()

返回一个产生有关*bytecode* 的连续范围的信息的迭代器。其产生的每一项都是一个 (start, end, lineno) tuple:

- start (一个 int) 代表相对于*bytecode* 范围开始位置的偏移量 (不包括该位置)。
- end (int 值) 代表相对于*bytecode* 范围末尾位置的偏移量（不包括该位置)。
- lineno 是一个代表*bytecode* 范围内的行号的 int，或者如果给定范围内的字节码没有行号则为 None。

产生的条目将具有下列特征属性:

- 产出的第一个范围将以 0 作为 start。
- (start, end) 范围将是非递减和连续的。也就是说，对于任何一对 tuple，第二个的 start 将等于第一个的 end。
- 任何范围都不会是反向的：对于所有三元组均有 end >= start。
- 产生的最后一个 tuple 的 end 将等于*bytecode* 的大小。

零宽度范围，即 start == end 也是允许的。零宽度范围的使用场景是源代码中存在，但被*bytecode* 编译器所去除的那些行。

Added in version 3.10.

> **参见**
>
> **PEP 626 - 在调试和其他工具中使用精确的行号。**
> 引入 `co_lines()` 方法的 PEP。

`codeobject.`**`replace`**`(`*`**kwargs`*`)`

返回代码对象的一个副本，使用指定的新字段值。

代码对象也被泛型函数 `copy.replace()` 所支持。

Added in version 3.8.

帧对象

帧对象表示执行帧。它们可能出现在回溯对象 中，还会被传递给已注册的跟踪函数。

特殊的只读属性

`frame.`**`f_back`**	指向前一个栈帧（对于调用方而言），或者如果这是最底部的栈帧则为 `None`
`frame.`**`f_code`**	该帧中正在执行的代码对象。访问该属性将引发一个 审计事件 `object.__getattr__`，附带参数 `obj` 和 `"f_code"`。
`frame.`**`f_locals`**	被该帧用来查找局部变量 的映射。如果该帧指向一个*optimized scope*，这可能返回一个直通写入代理对象。 在 3.13 版本发生变更: 返回一个已优化作用域的代理。
`frame.`**`f_globals`**	被帧用于查找全局变量 的字典
`frame.`**`f_builtins`**	被帧用于查找内置（内建）名称 的字典
`frame.`**`f_lasti`**	帧对象的“准确指令”（这是代码对象 的*bytecode*字符串的索引）

特殊的可写属性

frame.**f_trace**	如果不为 None，则是在代码执行期间调用各类事件的函数 (由调试器使用)。通常每个新的源代码行会触发一个事件 (参见*f_trace_lines*)。
frame.**f_trace_lines**	将该属性设为 False 以禁用为每个源代码行触发跟踪事件。
frame.**f_trace_opcodes**	将该属性设为 True 以允许请求每个操作码事件。请注意如果跟踪函数引发的异常逃逸到被跟踪的函数中这可能会导致未定义的解释器行为。
frame.**f_lineno**	该帧的当前行号 -- 在这里写入从一个跟踪函数内部跳转到的给定行（仅用于最底层的帧）。调试器可以通过写入该属性实现一个 Jump 命令（即设置下一条语句）。

帧对象方法

帧对象支持一个方法:

frame.**clear**()

此方法将清除该帧持有的全部对*局部变量* 的引用。并且，如果该帧归属于一个*generator*，此生成器将被终结。这有助于打破涉及帧对象的循环引用（例如当捕获一个 异常并保存其*回溯* 供以后使用）。

如果该帧当前正在执行或已挂起则会引发 RuntimeError。

Added in version 3.4.

在 3.13 版本发生变更: 尝试清除已挂起的帧将引发 RuntimeError (执行帧的情况将总是如此)。

回溯对象

回溯对象代表一个 异常的栈跟踪信息。当异常发生时会隐式地创建一个回溯对象，也可以通过调用 types.TracebackType 显式地创建。

在 3.7 版本发生变更: 现在回溯对象可以通过 Python 代码显式地实例化。

对于隐式地创建的回溯对象，当查找异常处理器使得执行栈展开时，会在每个展开层级的当前回溯之前插入一个回溯对象。当进入一个异常处理器时，程序将可以使用栈跟踪。(参见*try 语句* 一节。) 它可作为 sys.exc_info() 所返回的元组的第三项，以及所捕获异常的 __traceback__ 属性被获取。

当程序不包含适用的处理器时，栈跟踪会（以良好的格式）写入到标准错误流；如果解释器处于交互模式，它也将作为 sys.last_traceback 供用户使用。

对于显式地创建的回溯对象，则由回溯对象的创建者来决定应该如何连接*tb_next* 属性以构成完整的线跟踪。

特殊的只读属性:

traceback.**tb_frame**	指向当前层级的执行帧对象。 访问该属性将引发一个审计事件 object.__getattr__，附带参数 obj 和 "tb_frame"。
traceback.**tb_lineno**	给出异常发生所在的行号
traceback.**tb_lasti**	表示“精确指令”。

回溯中的行号和最后一条指令可能与其帧对象的行号不同，如果异常发生在try语句中且没有匹配的except子句或是有finally子句的话。

traceback.**tb_next**

特殊的可写属性 tb_next 是栈跟踪中的下一层级（通往发生异常的帧），如果没有下一层级则为 None。

在 3.7 版本发生变更: 该属性现在是可写的。

切片对象

切片对象被用来表示__getitem__()方法所使用的切片。该对象也可使用内置的 slice() 函数来创建。

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*)

此方法接受一个整型参数 *length* 并计算在切片对象被应用到 *length* 指定长度的条目序列时切片的相关信息应如何描述。其返回值为三个整型数组成的元组；这些数分别为切片的 *start* 和 *stop* 索引号以及 *step* 步长值。索引号缺失或越界则按照与正规切片相一致的方式处理。

静态方法对象

静态方法对象提供了一种胜过上文所述将函数对象转换为方法对象的方式。静态方法对象是对任意其他对象的包装器，通常用来包装用户自定义的方法对象。当从类或类实例获取一个静态方法对象时，实际返回的是经过包装的对象，它不会被进一步转换。静态方法对象也是可调用对象。静态方法对象可通过内置的 staticmethod() 构造器来创建。

类方法对象

类方法对象与静态方法类似，是对其他对象的包装器，会改变从类或类实例获取该对象的方式。类方法对象在这种获取操作中的行为已在上文中描述，见"实例方法"一节。类方法对象是通过内置 classmethod() 构造器创建的。

3.3 特殊方法名称

一个类可以通过定义具有特殊名称的方法来实现由特殊语法来发起调用的特定操作（例如算术运算或抽取与切片）。这是 Python 实现运算符重载的方式，允许每个类自行定义基于该语言运算符的特定行为。举例来说，如果一个类定义了名为__getitem__()的方法，并且 x 是该类的一个实例，则 x[i] 基本就等价于 type(x).__getitem__(x, i)。除非有说明例外情况，在没有定义适当方法的时候尝试执行某种操作将引发一个异常 (通常为 AttributeError 或 TypeError)。

将一个特殊方法设为 None 表示对应的操作不可用。例如，如果一个类将__iter__()设为 None，则该类就是不可迭代的，因此对其实例调用 iter() 将引发一个 TypeError (而不会回退至__getitem__())。[2]

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。)

3.3.1 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)。

[2] __hash__(), __iter__(), __reversed__(), __contains__(), __class_getitem__() 和 __fspath__() 方法对此有特殊处理。其他方法仍然会引发 TypeError，但可能会依赖 None 是不可调用对象的行为来做到这一点。

典型的实现会附带适当的参数使用 `super().__new__(cls[, ...])` 通过发起调用超类的 *__new__()* 方法来创建一个新的类实例然后在返回它之前根据需要修改新创建的实例。

如果 *__new__()* 在构造对象期间被发起调用并且它返回了一个 *cls* 的实例，则新实例的 *__init__()* 方法将以 `__init__(self[, ...])` 的形式被发起调用，其中 *self* 为新实例而其余的参数与被传给对象构造器的参数相同。

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*[, ...])

在实例 (通过 *__new__()*) 被创建之后，返回调用者之前调用。其参数与传递给类构造器表达式的参数相同。一个基类如果有 *__init__()* 方法，则其所派生的类如果也有 *__init__()* 方法，就必须显式地调用它以确保实例基类部分的正确初始化；例如: `super().__init__([args...])`.

因为对象是由 *__new__()* 和 *__init__()* 协作构造完成的 (由 *__new__()* 创建，并由 *__init__()* 定制)，所以 *__init__()* 返回的值只能是 `None`，否则会在运行时引发 `TypeError`。

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

在实例将被销毁时调用。这还被称为终结器或析构器（不适当）。如果一个基类具有 *__del__()* 方法，则其所派生的类如果也有 *__del__()* 方法，就必须显式地调用它以确保实例基类部分的正确清除。

__del__() 方法可以 (但不推荐!) 通过创建一个该实例的新引用来推迟其销毁。这被称为对象 重生。*__del__()* 是否会在重生的对象将被销毁时再次被调用是由具体实现决定的；当前的*CPython* 实现只会调用一次。

当解释器退出时并不保证会为仍然存在的对象调用 *__del__()* 方法。`weakref.finalize` 提供了一种直观的方式来注册当对象被作为垃圾回收时要调用的清理函数。

备注

`del x` 并不直接调用 `x.__del__()` --- 前者会将 `x` 的引用计数减一，而后者仅会在 `x` 的引用计数变为零时被调用。

CPython 实现细节： 一个引用循环可以阻止对象的引用计数归零。在这种情况下，循环将稍后被检测到并被循环垃圾回收器 删除。导致引用循环的一个常见原因是当一个异常在局部变量中被捕获。帧的局部变量将会引用该异常，这将引用它自己的回溯信息，它会又引用在回溯中捕获的所有帧的局部变量。

参见

`gc` 模块的文档。

警告

由于调用 *__del__()* 方法时周边状况已不确定，在其执行期间发生的异常将被忽略，改为打印一个警告到 `sys.stderr`。特别地：

- *__del__()* 可在任意代码被执行时启用，包括来自任意线程的代码。如果 *__del__()* 需要接受锁或启用其他阻塞资源，可能会发生死锁，例如该资源已被为执行 *__del__()* 而中断的代码所获取。
- *__del__()* 可以在解释器关闭阶段被执行。因此，它需要访问的全局变量（包含其他模块）可能已被删除或设为 `None`。Python 会保证先删除模块中名称以单个下划线打头的全局变量再删除其他全局变量；如果已不存在其他对此类全局变量的引用，这有助于确保导入的模块在 *__del__()* 方法被调用时仍然可用。

object.**__repr__**(*self*)

由 repr() 内置函数调用以输出一个对象的“官方”字符串表示。如果可能，这应类似一个有效的 Python 表达式，能被用来重建具有相同取值的对象（只要有适当的环境）。如果这不可能，则应返回形式如 <...some useful description...> 的字符串。返回值必须是一个字符串对象。如果一个类定义了*__repr__()* 但未定义*__str__()*，则在需要该类的实例的“非正式”字符串表示时也会使用*__repr__()*。

This is typically used for debugging, so it is important that the representation is information-rich and unambiguous. A default implementation is provided by the object class itself.

object.**__str__**(*self*)

Called by str(object), the default *__format__()* implementation, and the built-in function print(), to compute the ”informal” or nicely printable string representation of an object. The return value must be a str object.

此方法与*object.__repr__()* 的不同点在于*__str__()* 并不预期返回一个有效的 Python 表达式：可以使用更方便或更准确的描述信息。

内置类型 object 所定义的默认实现会调用*object.__repr__()*。

object.**__bytes__**(*self*)

Called by bytes to compute a byte-string representation of an object. This should return a bytes object. The object class itself does not provide this method.

object.**__format__**(*self*, *format_spec*)

通过 format() 内置函数、扩展、格式化字符串字面值 的求值以及 str.format() 方法调用以生成一个对象的“格式化”字符串表示。*format_spec* 参数为包含所需格式选项描述的字符串。*format_spec* 参数的解读是由实现*__format__()* 的类型决定的，不过大多数类或是将格式化委托给某个内置类型，或是使用相似的格式化选项语法。

请参看 formatspec 了解标准格式化语法的描述。

返回值必须为一个字符串对象。

The default implementation by the object class should be given an empty *format_spec* string. It delegates to *__str__()*.

在 3.4 版本发生变更: object 本身的 __format__ 方法如果被传入任何非空字符，将会引发一个 TypeError。

在 3.7 版本发生变更: object.__format__(x, '') 现在等同于 str(x) 而不再是 format(str(x), '')。

object.**__lt__**(*self*, *other*)

object.**__le__**(*self*, *other*)

object.**__eq__**(*self*, *other*)

object.**__ne__**(*self*, *other*)

object.**__gt__**(*self*, *other*)

object.**__ge__**(*self*, *other*)

以上这些被称为“富比较”方法。运算符号与方法名称的对应关系如下：x<y 调用 x.__lt__(y)、x<=y 调用 x.__le__(y)、x==y 调用 x.__eq__(y)、x!=y 调用 x.__ne__(y)、x>y 调用 x.__gt__(y)、x>=y 调用 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。

在默认情况下，object 通过使用 is 来实现*__eq__()* ，并在比较结果为假值时返回 NotImplemented：True if x is y else NotImplemented 。对于*__ne__()* ，默认会委托给*__eq__()* 并对结果取反，除非结果为 NotImplemented 。比较运算符之间没有其他隐含关系或默认实现；例如，(x<y or x==y) 为真并不意味着 x<=y 。要根据单根运算自动生成排序操作，请参看 functools.total_ordering() 。

By default, the `object` class provides implementations consistent with 值比较: equality compares according to object identity, and order comparisons raise `TypeError`. Each default method may generate these results directly, but may also return `NotImplemented`.

请查看*__hash__()* 的相关段落，了解创建可支持自定义比较运算并可用作字典键的*hashable* 对象时要注意的一些事项。

这些方法都没有对调参数版本（在左边参数不支持该操作但右边参数支持时使用）；而是*__lt__()* 和*__gt__()* 互为对方的反向，*__le__()* 和*__ge__()* 互为对方的反射，而*__eq__()* 和*__ne__()* 则是它们自己的反射。如果两个操作数的类型不同，且右操作数的类型是左操作数类型的直接或间接子类，则优先选择右操作数的反射方法，在其他情况下优先选择左操作数的方法。虚拟子类化不会被考虑。

当没有合适的方法返回任何 `NotImplemented` 以外的值时，`==` 和 `!=` 运算符将分别回退至 `is` 和 `is not`。

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

通过内置函数 `hash()` 调用以对哈希集的成员进行操作，属于哈希集的类型包括 `set`、`frozenset` 以及 `dict`。`__hash__()` 应该返回一个整数。对象比较结果相同所需的唯一特征属性是其具有相同的哈希值；建议的做法是把参与比较的对象全部组件的哈希值混在一起，即将它们打包为一个元组并对该元组做哈希运算。例如:

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

备注

`hash()` 会从一个对象自定义的*__hash__()* 方法返回值中截断为 `Py_ssize_t` 的大小。通常对 64 位构建为 8 字节，对 32 位构建为 4 字节。如果一个对象的*__hash__()* 必须在不同位大小的构建上进行互操作，请确保检查全部所支持构建的宽度。做到这一点的简单方法是使用 `python -c "import sys; print(sys.hash_info.width)"`。

如果一个类没有定义*__eq__()* 方法，那么它也不应该定义*__hash__()* 操作；如果它定义了*__eq__()* 但没有定义*__hash__()*，则其实例将不可被用作可哈希多项集的条目。如果一个类定义了可变对象并实现了*__eq__()* 方法，则它不应该实现*__hash__()*，因为*hashable* 多项集的实现要求键的哈希值是不可变的（如果对象的哈希值发生改变，它将位于错误的哈希桶中）。

User-defined classes have *__eq__()* and *__hash__()* methods by default (inherited from the `object` class); with them, all objects compare unequal (except with themselves) and `x.__hash__()` returns an appropriate value such that `x == y` implies both that `x is y` and `hash(x) == hash(y)`.

一个类如果重载了*__eq__()* 且没有定义*__hash__()* 则会将其*__hash__()* 隐式地设为 `None`。当一个类的*__hash__()* 方法为 `None` 时，该类的实例将在一个程序尝试获取其哈希值时正确地引发 `TypeError`，并会在检测 `isinstance(obj, collections.abc.Hashable)` 时被正确地识别为不可哈希对象。

如果一个重载了*__eq__()* 的类需要保留来自父类的*__hash__()* 实现，则必须通过设置 `__hash__ = <ParentClass>.__hash__` 来显式地告知解释器。

如果一个没有重载*__eq__()* 的类需要去掉哈希支持，则应该在类定义中包含 `__hash__ = None`。一个自定义了*__hash__()* 以显式地引发 `TypeError` 的类会被 `isinstance(obj, collections.abc.Hashable)` 调用错误地识别为可哈希对象。

备注

在默认情况下，str 和 bytes 对象的*__hash__()* 值会使用一个不可预知的随机值“加盐”。虽然它们在一个单独 Python 进程中会保持不变，但它们的值在重复运行的 Python 间是不可预测的。

这是为了防止通过精心选择输入来利用字典插入操作在最坏情况下的执行效率即 $O(n^2)$ 复杂度制度的拒绝服务攻击。请参阅 http://ocert.org/advisories/ocert-2011-003.html 了解详情。

改变哈希值会影响集合的迭代次序。Python 也从不保证这个次序不会被改变（通常它在 32 位和 64 位构建上是不一致的）。

另见 `PYTHONHASHSEED`.

在 3.3 版本发生变更: 默认启用哈希随机化。

`object.__bool__`(*self*)

Called to implement truth value testing and the built-in operation `bool()`; should return `False` or `True`. 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 `__bool__()` (which is true of the `object` class itself), all its instances are considered true.

3.3.2 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 the default attribute access fails with an `AttributeError` (either *__getattribute__()* raises an `AttributeError` because *name* is not an instance attribute or an attribute in the class tree for `self`; or *__get__()* of a *name* property raises `AttributeError`). This method should either return the (computed) attribute value or raise an `AttributeError` exception. The `object` class itself does not provide this method.

请注意如果属性是通过正常机制找到的，则*__getattr__()* 不会被调用。(这是在*__getattr__()* 和*__setattr__()* 之间故意设置的不对称性。）这既是出于执行效率理由也是因为不这样做的话*__getattr__()* 将无法访问实例的其他属性。要注意至少对于实例变量来说，你不必在实例属性字典中插入任何值（而是通过插入到其他对象）就可以实现对它的完全控制。请参阅下面的*__getattribute__()* 方法了解真正获取对属性访问的完全控制权的办法。

`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)`。

备注

此方法在作为通过特定语法或内置函数 隐式地调用的结果的情况下查找特殊方法时仍可能会被跳过。参见特殊方法查找。

对于特定的敏感属性访问，引发一个 审计事件 `object.__getattr__`，附带参数 `obj` 和 `name`。

`object.__setattr__`(*self*, *name*, *value*)

此方法在一个属性被尝试赋值时被调用。这个调用会取代正常机制（即将值保存到实例字典）。*name* 为属性名称，*value* 为要赋给属性的值。

如果*__setattr__()* 想要赋值给一个实例属性，它应该调用同名的基类方法，例如 `object.__setattr__(self, name, value)`。

对特定敏感属性的赋值，会引发一个 审计事件 `object.__setattr__`，附带参数 `obj`, `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。

对于特定的敏感属性删除，引发一个 审计事件 `object.__delattr__`，附带参数 `obj` 和 `name`。

object.**__dir__**(*self*)

此方法会在针对相应对象调用 dir() 时被调用。返回值必须为一个可迭代对象。dir() 会把返回的可迭代对象转换为列表并对其排序。

自定义模块属性访问

特殊名称 __getattr__ 和 __dir__ 还可被用来自定义对模块属性的访问。模块层级的 __getattr__ 函数应当接受一个参数，其名称为一个属性名，并返回计算结果值或引发一个 AttributeError。如果通过正常查找即*object.__getattribute__()* 未在模块对象中找到某个属性，则 __getattr__ 会在模块的 __dict__ 中查找，未找到时会引发一个 AttributeError。如果找到，它会以属性名被调用并返回结果值。

__dir__ 函数应当不接受任何参数，并且返回一个表示模块中可访问名称的字符串可迭代对象。此函数如果存在，将会重写一个模块中的标准 dir() 搜索操作。

想要更细致地自定义模块的行为（设置属性和特性属性等待），可以将模块对象的 __class__ 属性设置为一个 types.ModuleType 的子类。例如:

```
import sys
from types import ModuleType

class VerboseModule(ModuleType):
    def __repr__(self):
        return f'Verbose {self.__name__}'

    def __setattr__(self, attr, value):
        print(f'Setting {attr}...')
        super().__setattr__(attr, value)

sys.modules[__name__].__class__ = VerboseModule
```

备注

定义模块的 __getattr__ 和设置模块的 __class__ 只会影响使用属性访问语法进行的查找 -- 直接访问模块全局变量（不论是通过模块内的代码还是通过对模块全局字典的引用）是不受影响的。

在 3.5 版本发生变更: __class__ 模块属性改为可写。

Added in version 3.7: __getattr__ 和 __dir__ 模块属性。

参见

PEP 562 - 模块 __getattr__ 和 __dir__
描述用于模块的 __getattr__ 和 __dir__ 函数。

实现描述器

The following methods only apply when an instance of the class containing the method (a so-called *descriptor* class) appears in an *owner* class (the descriptor must be in either the owner's class dictionary or in the class dictionary for one of its parents). In the examples below, "the attribute" refers to the attribute whose name is the key of the property in the owner class' *__dict__*. The object class itself does not implement any of these protocols.

object.**__get__**(*self*, *instance*, *owner=None*)

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）。可选的 *owner* 参数是所有者类而 *instance* 是被用来访问属性的实例，如果通过 *owner* 来访问属性则返回 None。

此方法应当返回计算得到的属性值或是引发 AttributeError 异常。

PEP 252 指明*__get__()* 为带有一至二个参数的可调用对象。Python 自身内置的描述器支持此规格定义；但是，某些第三方工具可能要求必须带两个参数。Python 自身的*__getattribute__()* 实现总是会传入两个参数，无论它们是否被要求提供。

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*。

请注意，添加*__set__()* 或*__delete__()* 会将描述器变成“数据描述器”。更多细节请参阅调用描述器。

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。

描述器的实例也可能存在 __objclass__ 属性：

object.**__objclass__**

属性 __objclass__ 会被 inspect 模块解读为指定此对象定义所在的类（正确设置此属性有助于动态类属性的运行时内省）。对于可调用对象来说，它可以指明预期或要求提供一个特定类型（或子类）的实例作为第一个位置参数（例如，CPython 会为在 C 中实现的未绑定方法设置此属性）。

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: *__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。

但是，如果找到的值是定义了某个描述器方法的对象，则 Python 可能会重载默认行为并转而发起调用描述器方法。这具体发生在优先级链的哪个环节则要根据所定义的描述器方法及其被调用的方式来决定。

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
如果绑定到一个对象实例，a.x 会被转换为调用: type(a).__dict__['x'].__get__(a, type(a))。

类绑定
如果绑定到一个类，A.x 会被转换为调用: A.__dict__['x'].__get__(None, A)。

超绑定
类似 super(A, a).x 这样的带点号查找将在 a.__class__.__mro__ 中搜索紧接在 A 之后的基类 B 并返回 B.__dict__['x'].__get__(a, A)。如果 x 不是描述器，则不加改变地返回它。

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__()，则访问属性将返回描述器对象自身，除非对象的实例字典中有相应的属性值。如果描述器定义了 __set__() 和/或 __delete__()，则它是一个数据描述器；如果两者均未定义，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__()，而非数据描述器则只有 __get__() 方法。定义了 __get__() 和 __set__() (和/或 __delete__()) 的数据描述器总是会重载实例字典中的定义。与之相对地，非数据描述器则可被实例所重载。

Python 方法（包括用 @staticmethod 和 @classmethod 装饰的方法）都是作为非数据描述器来实现的。因而，实例可以重定义和重写方法。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__

__slots__ 允许我们显式地声明数据成员（如特征属性）并禁止创建*__dict__* 和 *__weakref__* (除非是在 *__slots__* 中显式地声明或是在父类中可用。)

相比使用*__dict__* 可以显著节省空间。属性查找速度也可得到显著的提升。

object.**__slots__**

这个类变量可赋值为字符串、可迭代对象或由实例使用的变量名组成的字符串序列。*__slots__* 会为已声明的变量保留空间并阻止自动为每个实例创建*__dict__* 和 *__weakref__*。

使用 *__slots__* 的注意事项:

- 当继承自一个没有 *__slots__* 的类时，实例的*__dict__* 和 *__weakref__* 属性将总是可访问的。
- 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`。如果需要动态地给新变量赋值，则要将 `'__dict__'` 加入到在 *__slots__* 中声明的字符串序列中。
- 如果未给每个实例设置 *__weakref__* 变量，则定义了 *__slots__* 的类就不支持对其实例的 弱引用。如果需要支持弱引用，则要将 `'__weakref__'` 加入到在 *__slots__* 中声明的字符串序列中。
- *__slots__* 是通过为每个变量名创建描述器 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，类属性将会覆盖描述器赋值。
- The action of a *__slots__* declaration is not limited to the class where it is defined. *__slots__* declared in parents are available in child classes. However, instances of a child subclass will get a *__dict__* and *__weakref__* unless the subclass also defines *__slots__* (which should 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。
- 如果为派生自 `"variable-length"` 内置类型如 `int`, `bytes` 和 `tuple` 的类定义了非空的 *__slots__* 则将引发 `TypeError`。
- 任何非字符串的*iterable* 都可以被赋值给 *__slots__*。
- 如果是使用一个 字典来给 *__slots__* 赋值，则该字典的键将被用作槽位名称。字典的值可被用来为每个属性提供将被 `inspect.getdoc()` 识别并在 and displayed in the output of `help()` 的输出中显示的文档字符串。
- *__class__* assignment works only if both classes have the same *__slots__*.
- 带有多槽位父类的 多重继承也是可用的，但仅允许一个父类具有由槽位创建的属性（其他基类必须具有空的槽位布局）——违反此规则将引发 `TypeError`。
- 如果将*iterator* 用于 *__slots__* 则会为该迭代器的每个值创建一个*descriptor*。但是，*__slots__* 属性将为一个空迭代器。

3.3.3 自定义类创建

当一个类继承另一个类时，会在这个父类上调用*__init_subclass__()*。这样，就使得编写改变子类行为的类成为可能。这与类装饰器有很密切的关联，但类装饰器只能影响它们所应用的特定类，而 `__init_subclass__` 则只作用于定义了该方法的类在未来的子类。

classmethod object.**__init_subclass__**(*cls*)

当所在类派生子类时此方法就会被调用。*cls* 将指向新的子类。如果定义为一个普通实例方法，此方法将被隐式地转换为类方法。

传给一个新类的关键字参数会被传给上级类的 `__init_subclass__`。为了与其他使用 `__init_subclass__` 的类兼容，应当去掉需要的关键字参数再将其他参数传给基类，例如:

```
class Philosopher:
    def __init_subclass__(cls, /, default_name, **kwargs):
        super().__init_subclass__(**kwargs)
```

(续下页)

(接上页)

```
        cls.default_name = default_name

class AustralianPhilosopher(Philosopher, default_name="Bruce"):
    pass
```

`object.__init_subclass__` 的默认实现什么都不做，只在带任意参数调用时引发一个错误。

> **备注**
>
> 元类提示 `metaclass` 将被其它类型机制消耗掉，并不会被传给 `__init_subclass__` 的实现。实际的元类（而非显式的提示）可通过 `type(cls)` 访问。

Added in version 3.6.

当一个类被创建时，`type.__new__()` 会扫描类变量并对其中带有*__set_name__()* 钩子的对象执行回调。

`object.`**`__set_name__`**(*self*, *owner*, *name*)

在所有者类 *owner* 被创建时自动调用。此对象已被赋值给该类中的 *name*:

```
class A:
    x = C()  # 自动调用: x.__set_name__(A, 'x')
```

如果类变量赋值是在类被创建之后进行的，*__set_name__()* 将不会被自动调用。如有必要，可以直接调用*__set_name__()*:

```
class A:
   pass

c = C()
A.x = c                  # 钩子未被调用
c.__set_name__(A, 'x')   # 手动发起调用钩子
```

详情参见创建类对象。

Added in version 3.6.

元类

默认情况下，类是使用 `type()` 来构建的。类体会在一个新的命名空间内执行，类名会被局部绑定到 `type(name, bases, namespace)` 的结果。

类创建过程可通过在定义行传入 `metaclass` 关键字参数，或是通过继承一个包含此参数的现有类来进行定制。在以下示例中，`MyClass` 和 `MySubclass` 都是 `Meta` 的实例:

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

在类定义内指定的任何其他关键字参数都会在下面所描述的所有元类操作中进行传递。

当一个类定义被执行时，将发生以下步骤:

- 解析 MRO 条目；
- 确定适当的元类；

- 准备类命名空间；
- 执行类主体；
- 创建类对象。

解析 MRO 条目

object.**__mro_entries__**(*self*, *bases*)

如果一个出现在类定义中的基类不是 type 的实例，则会在该基类中搜索 __mro_entries__() 方法。如果找到了 __mro_entries__() 方法，则在创建类时该基类会被替换为调用 __mro_entries__() 的结果。该方法被调用时将附带传给 *bases* 形参的原始基类元组，并且必须返回一个由将被用来替代该基类的类组成的元组。返回的元组可能为空：在此情况下，原始基类将被忽略。

参见

types.resolve_bases()
动态地解析不属于 type 实例的基类。

types.get_original_bases()
在类被 *__mro_entries__()* 修改之前提取其“原始基类”。

PEP 560
对 typing 模块和泛用类型的核心支持。

确定适当的元类

为一个类定义确定适当的元类是根据以下规则:

- 如果没有基类且没有显式指定元类，则使用 type()；
- 如果给出一个显式元类而且 *不是* type() 的实例，则其会被直接用作元类；
- 如果给出一个 type() 的实例作为显式元类，或是定义了基类，则使用最近派生的元类。

最近派生的元类会从显式指定的元类（如果有）以及所有指定的基类的元类（即 type(cls)）中选取。最近派生的元类应为 *所有* 这些候选元类的一个子类型。如果没有一个候选元类符合该条件，则类定义将失败并抛出 TypeError。

准备类命名空间

一旦确定了适当的元类，就开始准备类的命名空间。如果元类具有 __prepare__ 属性，它将以 namespace = metaclass.__prepare__(name, bases, **kwds) 的形式被调用（其中如果存在任何额外关键字参数，则应来自类定义）。__prepare__ 方法应当被实现为 *类方法*。__prepare__ 所返回的命名空间会被传入 __new__，但是当最终的类对象被创建时该命名空间会被拷贝到一个新的 dict 中。

如果元类没有 __prepare__ 属性，则类命名空间将初始化为一个空的有序映射。

参见

PEP 3115 - Python 3000 中的元类
引入 __prepare__ 命名空间钩子

执行类主体

类主体会以（类似于）exec(body, globals(), namespace) 的形式被执行。普通调用与 exec() 的关键区别在于当类定义发生于函数内部时，词法作用域允许类主体（包括任何方法）引用来自当前和外部作用域的名称。

但是，即使当类定义发生于函数内部时，在类内部定义的方法仍然无法看到在类作用域层次上定义的名称。类变量必须通过实例的第一个形参或类方法来访问，或者是通过下一节中描述的隐式词法作用域的 `__class__` 引用。

创建类对象

一旦执行类主体完成填充类命名空间，将通过调用 `metaclass(name, bases, namespace, **kwds)` 创建类对象（此处的附加关键字参数与传入 `__prepare__` 的相同）。

如果类主体中有任何方法引用了 `__class__` 或 `super`，这个类对象会通过零参数形式的 `super().__class__` 所引用，这是由编译器所创建的隐式闭包引用。这使用零参数形式的 `super()` 能够正确标识正在基于词法作用域来定义的类，而被用于进行当前调用的类或实例则是基于传递给方法的第一个参数来标识的。

CPython 实现细节： 在 CPython 3.6 及之后的版本中，`__class__` 单元会作为类命名空间中的 `__classcell__` 条目被传给元类。如果存在，它必须被向上传播给 `type.__new__` 调用，以便能正确地初始化该类。如果不这样做，在 Python 3.8 中将引发 `RuntimeError`。

当使用默认的元类 `type`，或者任何最终会调用 `type.__new__` 的元类时，以下额外的自定义步骤将在创建类对象之后被发起调用：

1) `type.__new__` 方法会收集类命名空间中所有定义了*`__set_name__()`* 方法的属性；
2) 这些 `__set_name__` 方法将附带所定义的类和指定的属性所赋的名称进行调用；
3) 在新类基于方法解析顺序所确定的直接父类上调用*`__init_subclass__()`* 钩子。

在类对象创建之后，它会被传给包含在类定义中的类装饰器（如果有的话），得到的对象将作为已定义的类绑定到局部命名空间。

When a new class is created by `type.__new__`, the object provided as the namespace parameter is copied to a new ordered mapping and the original object is discarded. The new copy is wrapped in a read-only proxy, which becomes the *`__dict__`* attribute of the class object.

参见

PEP 3135 - 新的超类型
描述隐式的 `__class__` 闭包引用

元类的作用

元类的潜在作用非常广泛。已经过尝试的设想包括枚举、日志、接口检查、自动委托、自动特征属性创建、代理、框架以及自动资源锁定/同步等等。

3.3.4 自定义实例及子类检查

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。

`type.__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)`。

`type.__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
Includes the specification for customizing `isinstance()` and `issubclass()` behavior through *`__instancecheck__()`* and *`__subclasscheck__()`*, with motivation for this functionality in the context of adding Abstract Base Classes (see the `abc` module) to the language.

3.3.5 模拟泛型类型

当使用类型标注时，使用 Python 的方括号标记来形参化一个*generic type* 往往会很有用处。例如，`list[int]` 这样的标注可以被用来表示一个 `list` 中的所有元素均为 `int` 类型。

参见

PEP 484 ——类型注解
介绍 Python 中用于类型标注的框架

泛用别名类型
代表形参化泛用类的对象的文档

Generics, 用户自定义泛型和 `typing.Generic`
有关如何实现可在运行时被形参化并能被静态类型检查器所识别的泛用类的文档。

一个类通常只有在定义了特殊的类方法 `__class_getitem__()` 时才能被形参化。

classmethod object.**__class_getitem__**(*cls*, *key*)
按照 *key* 参数指定的类型返回一个表示泛型类的专门化对象。

当在类上定义时，`__class_getitem__()` 会自动成为类方法。因此，当它被定义时没有必要使用 `@classmethod` 来装饰。

__class_getitem__ 的目的

`__class_getitem__()` 的目的是允许标准库泛型类的运行时形参化以更方便地对这些类应用类型提示。

要实现可以在运行时被形参化并可被静态类型检查所理解的自定义泛型类，用户应当从已经实现了*`__class_getitem__()`* 的标准库类继承，或是从 `typing.Generic` 继承，这个类拥有自己的 `__class_getitem__()` 实现。

标准库以外的类上的*`__class_getitem__()`* 自定义实现可能无法被第三方类型检查器如 mypy 所理解。不建议在任何类上出于类型提示以外的目的使用 `__class_getitem__()`。

__class_getitem__ 与 *__getitem__*

通常，使用方括号语法抽取一个对象将会调用在该对象的类上定义的*`__getitem__()`* 实例方法。不过，如果被拟抽取的对象本身是一个类，则可能会调用*`__class_getitem__()`* 类方法。`__class_getitem__()` 如果被正确地定义，则应当返回一个 GenericAlias 对象。

使用表达式 `obj[x]` 来呈现，Python 解释器会遵循下面这样的过程来确定应当调用*`__getitem__()`* 还是*`__class_getitem__()`*：

```
from inspect import isclass

def subscribe(obj, x):
    """返回表达式 'obj[x]' 的结果"""

    class_of_obj = type(obj)

    # 如果 obj 所属的类定义了 __getitem__,
    # 则调用 class_of_obj.__getitem__(obj, x)
```

(续下页)

(接上页)

```
    if hasattr(class_of_obj, '__getitem__'):
        return class_of_obj.__getitem__(obj, x)

    # 否则，如果 obj 是一个类并且定义了 __class_getitem__，
    # 则调用 obj.__class_getitem__(x)
    elif isclass(obj) and hasattr(obj, '__class_getitem__'):
        return obj.__class_getitem__(x)

    # 否则，引发一个异常
    else:
        raise TypeError(
            f"'{class_of_obj.__name__}' object is not subscriptable"
        )
```

在 Python 中，所有的类自身也是其他类的实例。一个类所属的类被称为该类的*metaclass*，并且大多数类都将 `type` 类作为它们的元类。`type` 没有定义 *`__getitem__()`*，这意味着 `list[int]`, `dict[str, float]` 和 `tuple[str, bytes]` 这样的表达式都将导致 *`__class_getitem__()`* 被调用:

```
>>> # list 以 "type" 类作为其元类，与大多数类一样：
>>> type(list)
<class 'type'>
>>> type(dict) == type(list) == type(tuple) == type(str) == type(bytes)
True
>>> # "list[int]" 将调用 "list.__class_getitem__(int)"
>>> list[int]
list[int]
>>> # list.__class_getitem__ 将返回一个 GenericAlias 对象：
>>> type(list[int])
<class 'types.GenericAlias'>
```

然而，如果一个类属于定义了 *`__getitem__()`* 的自定义元类，则抽取该类可能导致不同的行为。这方面的一个例子可以在 `enum` 模块中找到:

```
>>> from enum import Enum
>>> class Menu(Enum):
...     """A breakfast menu"""
...     SPAM = 'spam'
...     BACON = 'bacon'
...
>>> # 枚举类有一个自定义元类：
>>> type(Menu)
<class 'enum.EnumMeta'>
>>> # EnumMeta 定义了 __getitem__，
>>> # 因此 __class_getitem__ 不会被调用，
>>> # 并且结果不是一个 GenericAlias 对象：
>>> Menu['SPAM']
<Menu.SPAM: 'spam'>
>>> type(Menu['SPAM'])
<enum 'Menu'>
```

参见

PEP 560 - 对 typing 模块和泛型的核心支持
介绍 *`__class_getitem__()`*，并指明抽取在何时会导致 `__class_getitem__()` 而不是 *`__getitem__()`* 被调用

3.3.6 模拟可调用对象

object.**__call__**(*self*[, *args...*])

Called when the instance is "called" as a function; if this method is defined, `x(arg1, arg2, ...)` roughly translates to `type(x).__call__(x, arg1, ...)`. The `object` class itself does not provide this method.

3.3.7 模拟容器类型

The following methods can be defined to implement container objects. None of them are provided by the `object` class itself. 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. It is also recommended that mappings provide the methods `keys()`, `values()`, `items()`, `get()`, `clear()`, `setdefault()`, `pop()`, `popitem()`, `copy()`, and `update()` behaving similar to those for Python's standard `dictionary` objects. The `collections.abc` module provides a `MutableMapping` *abstract base 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 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 search the mapping's keys; 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 iterate through the object's keys; for sequences, it should iterate through the values.

object.**__len__**(*self*)

调用此方法以实现内置函数 `len()`。应该返回对象的长度，以一个 >= 0 的整数表示。此外，如果一个对象未定义*__bool__()* 方法而其 `__len__()` 方法返回值为零则它在布尔运算中将被视为具有假值。

CPython 实现细节：在 CPython 中，要求长度最大只能为 `sys.maxsize`。如果长度大于 `sys.maxsize` 则某些特性（如 `len()`）可能会引发 `OverflowError`。要防止真值测试引发 `OverflowError`，对象必须定义*__bool__()* 方法。

object.**__length_hint__**(*self*)

调用此方法以实现 `operator.length_hint()`。应该返回对象长度的估计值（可能大于或小于实际长度）。此长度应为一个 >= 0 的整数。返回值也可以为 `NotImplemented`，这会被视作与 `__length_hint__` 方法完全不存在时一样处理。此方法纯粹是为了优化性能，并不要求正确无误。

Added in version 3.4.

备注

切片是通过下述三个专门方法完成的。以下形式的调用

```
a[1:2] = b
```

会为转写为

```
a[slice(1, 2, None)] = b
```

其他形式以此类推。略去的切片项总是以 `None` 补全。

object.**__getitem__**(*self*, *key*)

调用此方法以实现 `self[key]` 的求值。对于*sequence* 类型，接受的键应为整数。作为可选项，它们也可能支持 `slice` 对象。对负数索引的支持也是可选项。如果 *key* 的类型不正确，则可能引发 `TypeError`。如果 *key* 为序列索引集合范围以外的值（在进行任何负数索引的特殊解读之后），则应当引发 `IndexError`。对于*mapping* 类型，如果 *key* 找不到（不在容器中），则应当引发 `KeyError`。

> **备注**
>
> *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。

> **备注**
>
> 当*抽取* 一个 *class* 时，可能会调用特殊类方法 *__class_getitem__()* 而不是 __getitem__()。请参阅 *__class_getitem__ 与 __getitem__* 了解详情。

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*)

此方法会在需要为一个容器创建*iterator* 时被调用。此方法应当返回一个新的迭代器对象，它可以对容器中的所有对象执行迭代。对于映射，它应当对窗口中的键执行迭代。

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。

成员检测运算符 (*in* 和 *not in*) 通常以对容器进行逐个迭代的方式来实现。不过，容器对象可以提供以下特殊方法并采用更有效率的实现，这样也不要求对象必须为可迭代对象。

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*。

3.3.8 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.**__matmul__**(*self*, *other*)

object.**__truediv__**(*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*)

调用这些方法来实现双目算术运算 (+, -, *, @, /, //, %, divmod(), pow(), **, <<, >>, &, ^, |)。例如，求表达式 x + y 的值，其中 *x* 是具有__add__() 方法的类的一个实例，则会调用 type(x).__add__(x, y)。__divmod__() 方法应该等价于使用__floordiv__() 和__mod__()；它不应该被关联到__truediv__()。请注意如果要支持三目版本的内置 pow() 函数则__pow__() 应当被定义为接受可选的第三个参数。

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.**__radd__**(*self*, *other*)

object.**__rsub__**(*self*, *other*)

object.**__rmul__**(*self*, *other*)

object.**__rmatmul__**(*self*, *other*)

object.**__rtruediv__**(*self*, *other*)

object.**__rfloordiv__**(*self*, *other*)

object.**__rmod__**(*self*, *other*)

object.**__rdivmod__**(*self*, *other*)

object.**__rpow__**(*self*, *other*[, *modulo*])

object.**__rlshift__**(*self*, *other*)

object.**__rrshift__**(*self*, *other*)

object.**__rand__**(*self*, *other*)

object.**__rxor__**(*self*, *other*)

object.**__ror__**(*self*, *other*)

调用这些方法来实现具有反射（交换）操作数的双目算术运算 (+, - ``, ``*, @, /, //, %, divmod(), pow(), **, <<, >>, &, ^, |)。这些函数仅会在左操作数不支持相应运算[3] 且两个操作数类型不同时被调用。[4] 例如，求表达式 x - y 的值，其中 *y* 是具有__rsub__() 方法的类的一个实例，则当 type(x).__sub__(x, y) 返回 NotImplemented 时将会调用 type(y).__rsub__(y, x)。

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)。

> **备注**
>
> 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，则此方法将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.**__imatmul__**(*self*, *other*)

object.**__itruediv__**(*self*, *other*)

object.**__ifloordiv__**(*self*, *other*)

object.**__imod__**(*self*, *other*)

object.**__ipow__**(*self*, *other*[, *modulo*])

[3] 这里的“不支持”是指该类无此方法，或方法返回 NotImplemented 。如果你想强制回退到右操作数的反射方法，请不要设置方法为 None —那会造成显式地 阻塞此种回退的相反效果。

[4] 对于相同类型的操作数，如果非返回方法 -- 例如__add__() -- 失败则会认为整个运算都不被支持，这就是反射方法不会被调用的原因。

object.**__ilshift__**(*self*, *other*)

object.**__irshift__**(*self*, *other*)

object.**__iand__**(*self*, *other*)

object.**__ixor__**(*self*, *other*)

object.**__ior__**(*self*, *other*)

调用这些方法来实现增强算术赋值 (+=, -=, *=, @=, /=, //=, %=, **=, <<=, >>=, &=, ^=, |=)。这些方法应当尝试原地执行操作 (对 *self* 进行修改) 并返回结果 (结果可以为 *self* 但这并非必须)。如果某个方法未被定义，或者如果该方法返回 `NotImplemented`，则相应的增强赋值将回退到普通方法。举例来说，如果 *x* 是一个具有*__iadd__()* 方法的类的实例，则 `x += y` 就等价于 `x = x.__iadd__(y)`。如果*__iadd__()* 不存在，或者如果 `x.__iadd__(y)` 返回 `NotImplemented`，则将使用 `x.__add__(y)` 和 `y.__radd__(x)`，如同对 `x + y` 求值一样。在某些情况下，增强赋值可能导致未预期的错误 (参见 faq-augmented-assignment-tuple-error)，但此行为实际上是数据模型的一部分。

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.**__float__**(*self*)

调用这些方法以实现内置函数 `complex()`, `int()` 和 `float()`。应当返回一个相应类型的值。

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

调用此方法以实现 `operator.index()` 以及 Python 需要无损地将数字对象转换为整数对象的场合 (例如切片或是内置的 `bin()`, `hex()` 和 `oct()` 函数)。存在此方法表明数字对象属于整数类型。必须返回一个整数。

如果未定义*__int__()*, *__float__()* 和*__complex__()* 则相应的内置函数 `int()`, `float()` 和 `complex()` 将回退为*__index__()*。

object.**__round__**(*self*[, *ndigits*])

object.**__trunc__**(*self*)

object.**__floor__**(*self*)

object.**__ceil__**(*self*)

调用这些方法以实现内置函数 `round()` 以及 `math` 函数 `trunc()`, `floor()` 和 `ceil()`。除了将 *ndigits* 传给 `__round__()` 的情况之外这些方法的返回值都应当是原对象截断为 `Integral` (通常为 `int`)。

如果*__int__()* 或*__index__()* 均未被定义则内置函数 `int()` 会回退至*__trunc__()*。

在 3.11 版本发生变更: 将 `int()` 委托给*__trunc__()* 的做法已被弃用。

3.3.9 with 语句上下文管理器

上下文管理器是一个对象，它定义了在执行*with* 语句时要建立的运行时上下文。上下文管理器处理进入和退出所需运行时上下文以执行代码块。通常使用 `with` 语句 (在*with 语句* 中描述)，但是也可以通过直接调用它们的方法来使用。

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。

For more information on context managers, see typecontextmanager. The `object` class itself does not provide the context manager methods.

object.**__enter__**(*self*)

进入与此对象相关的运行时上下文。*with* 语句将会绑定这个方法的返回值到 `as` 子句中指定的目标，如果有的话。

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。

3.3.10 定制类模式匹配中的位置参数

当在模式中使用类名称时，默认不允许模式中出现位置参数，例如在 MyClass 没有特别支持的情况下 case MyClass(x, y) 通常是无效的。要能使用这样的模式，类必须定义一个 *__match_args__* 属性。

object.**__match_args__**

该类变量可以被赋值为一个字符串元组。当该类被用于带位置参数的类模式时，每个位置参数都将被转换为关键字参数，并使用 *__match_args__* 中的对应值作为关键字。缺失此属性就等价于将其设为 ()。

举例来说，如果 MyClass.__match_args__ 为 ("left", "center", "right") 则意味着 case MyClass(x, y) 就等价于 case MyClass(left=x, center=y)。请注意模式中参数的数量必须小于等于 *__match_args__* 中元素的数量；如果前者大于后者，则尝试模式匹配时将引发 TypeError。

Added in version 3.10.

参见

PEP 634 - 结构化模式匹配
有关 Python match 语句的规范说明。

3.3.11 模拟缓冲区类型

缓冲区协议为 Python 对象提供了一种向低层级内存数组暴露高效访问的方式。该协议是通过内置类型如 bytes 和 memoryview 实现的，还可能由第三方库定义额外的缓冲区类型。

虽然缓冲区类型通常都是用 C 实现的，但用 Python 来实现该协议也是可能的。

object.**__buffer__**(*self*, *flags*)

当从 *self* 请求一个缓冲区时将被调用（例如，从 memoryview 构造器）。*flags* 参数是代表所请求缓冲区的类别的整数，例如这会影响返回的缓冲区是只读还是可写。inspect.BufferFlags 提供了解读旗标的便利方式。此方法必须返回一个 memoryview 对象。

object.**__release_buffer__**(*self*, *buffer*)

当一个缓冲区不再需要时将被调用。*buffer* 参数是在此之前由*__buffer__()* 返回的 memoryview 对象。此方法必须释放任何关联到该缓冲区的资源。此方法应当返回 None。不需要执行任何清理的缓冲区对象不要求实现此方法。

Added in version 3.12.

参见

PEP 688 - 使缓冲区协议在 Python 中可访问
引入 Python __buffer__ 和 __release_buffer__ 方法。

collections.abc.Buffer
缓冲区类型的 ABC。

3.3.12 特殊方法查找

对于自定义类来说，特殊方法的隐式发起调用仅保证在其定义于对象类型中时能正确地发挥作用，而不能定义在对象实例字典中。该行为就是以下代码会引发异常的原因。:

```
>>> class C:
...     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，则当它们在对类型对象自身发起调用时将会失败:

```
>>> 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
```

除了出于正确性考虑而会绕过任何实例属性，隐式特殊方法查找通常还会绕过*__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
10
>>> len(c)                      # Implicit lookup
10
```

以这种方式绕过__getattribute__() 机制为解释器内部的速度优化提供了显著的空间，其代价则是牺牲了一些处理特殊方法时的灵活性（特殊方法 *must* 必须设置在类对象自身上以便始终一致地由解释器发起调用）。

3.4 协程

3.4.1 可等待对象

awaitable 对象主要实现了__await__() 方法。从async def 函数返回的协程对象 即为可等待对象。

> **备注**
>
> 从带有 types.coroutine() 装饰器的生成器返回的*generator iterator* 对象也属于可等待对象，但它们并未实现__await__()。

object.**__await__**(*self*)

Must return an *iterator*. Should be used to implement *awaitable* objects. For instance, asyncio.Future implements this method to be compatible with the await expression. The object class itself is not awaitable and does not provide this method.

> **备注**
>
> 本语言不会对 __await__ 所返回的迭代器产生的对象的类型或值施加任何限制，因为这是负责管理*awaitable* 对象的异步执行框架的具体实现 (如 asyncio) 专属特性。

Added in version 3.5.

> **参见**
>
> **PEP 492** 了解有关可等待对象的详细信息。

3.4.2 协程对象

协程对象 属于*awaitable* 对象。协程的执行可以通过调用__await__() 并迭代其结果来控制。当协程结束执行并返回时，迭代器会引发 StopIteration，而该异常的 value 属性将存放返回值。如果协程引发了异常，它会被迭代器传播出去。协程不应当直接引发未被处理的 StopIteration 异常。

协程也具有下面列出的方法，它们类似于生成器的对应方法 (参见生成器-迭代器的方法)。但是，与生成器不同，协程并不直接支持迭代。

在 3.5.2 版本发生变更: 等待一个协程超过一次将引发 RuntimeError。

coroutine.**send**(*value*)

开始或恢复协程的执行。如果 *value* 为 None，这将等价于前往__await__() 所返回的迭代器的下一项。如果 *value* 不为 None，此方法将委托给导致协挂起的迭代器的send() 方法。其结果（返回值, StopIteration 或是其他异常）将与上述对 __await__() 返回值进行迭代的结果相同。

coroutine.**throw**(*value*)

coroutine.**throw**(*type*[, *value*[, *traceback*]])

在协程内引发指定的异常。此方法将委托给导致该协程挂起的迭代器的throw() 方法，如果存在此方法的话。否则，该异常将在挂起点被引发。其结果（返回值，StopIteration 或是其他异常）将与上述对__await__() 返回值进行迭代的结果相同。如果该异常未在协程内被捕获，则将回传给调用方。

在 3.12 版本发生变更: 第二个签名 (type[, value[, traceback]]) 已被弃用并可能在未来的 Python 版本中移除。

coroutine.**close**()

此方法会使得协程清理自身并退出。如果协程被挂起，此方法会先委托给导致协程挂起的迭代器的*close()* 方法，如果存在该方法。然后它会在挂起点引发 GeneratorExit，使得协程立即清理自身。最后，协程会被标记为已结束执行，即使它根本未被启动。

当协程对象将要被销毁时，会使用以上处理过程来自动关闭。

3.4.3 异步迭代器

异步迭代器可以在其 __anext__ 方法中调用异步代码。

异步迭代器可在*async for* 语句中使用。

The object class itself does not provide these methods.

object.**__aiter__**(*self*)

必须返回一个 异步迭代器对象。

object.**__anext__**(*self*)

必须返回一个 可等待对象输出迭代器的下一结果值。当迭代结束时应该引发 StopAsyncIteration 错误。

异步可迭代对象的一个示例:

```
class Reader:
    async def readline(self):
        ...

    def __aiter__(self):
        return self

    async def __anext__(self):
        val = await self.readline()
        if val == b'':
            raise StopAsyncIteration
        return val
```

Added in version 3.5.

在 3.7 版本发生变更: 在 Python 3.7 之前，*__aiter__()* 可以返回一个 可等待对象并将被解析为异步迭代器。

从 Python 3.7 开始，*__aiter__()* 必须返回一个异步迭代器对象。返回任何其他对象都将导致 TypeError 错误。

3.4.4 异步上下文管理器

异步上下文管理器是 上下文管理器的一种，它能够在其 __aenter__ 和 __aexit__ 方法中暂停执行。

异步上下文管理器可在*async with* 语句中使用。

The object class itself does not provide these methods.

object.**__aenter__**(*self*)

在语义上类似于*__enter__()*，仅有的区别在于它必须返回一个 可等待对象。

object.**__aexit__**(*self*, *exc_type*, *exc_value*, *traceback*)

在语义上类似于*__exit__()*，仅有的区别在于它必须返回一个 可等待对象。

异步上下文管理器类的一个示例:

```
class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')
```

(续下页)

(接上页)

```
    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')
```

Added in version 3.5.

备注

CHAPTER 4

执行模型

4.1 程序的结构

Python 程序是由代码块构成的。*代码块*是被作为一个单元来执行的一段 Python 程序文本。以下几个都属于代码块：模块、函数体和类定义。交互式输入的每条命令都是代码块。一个脚本文件（作为标准输入发送给解释器或是作为命令行参数发送给解释器的文件）也是代码块。一条脚本命令（通过 `-c` 选项在解释器命令行中指定的命令）也是代码块。通过在命令行中使用 `-m` 参数作为最高层级脚本（即 `__main__` 模块）运行的模块也是代码块。传递给内置函数 `eval()` 和 `exec()` 的字符串参数也是代码块。

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。

4.2 命名与绑定

4.2.1 名称的绑定

*名称*用于指代对象。名称是通过名称绑定操作来引入的。

下面的结构将名字绑定:

- 函数的正式参数，
- 类定义，
- 函数定义，
- 赋值表达式，
- 如果在一个赋值中出现，则为标识符的*目标*：
  - *for* 循环头，
  - 在*with* 语句，*except* 子句，*except** 子句，或格式化模式匹配的 as 模式的 `as` 之后，
  - 在结构模式匹配中的捕获模式
- *import* 语句。
- *type* 语句。
- *类型形参列表*。

形式为 `from ... import *` 的 `import` 语句绑定所有在导入的模块中定义的名字，除了那些以下划线开头的名字。这种形式只能在模块级别上使用。

del 语句的目标也被视作一种绑定（虽然其实际语义为解除名称绑定）。

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）。

如果某个名称绑定在一个代码块中，则它就是该代码块的局部变量，除非声明为*nonlocal* 或*global*。如果某个名称绑定在模块层级，则它就是全局变量。（模块代码块的变量既是局部变量又是全局变量。）如果某个变量在一个代码块中被使用但不是在其中定义的，则它是*free variable*。

每个在程序文本中出现的名称是指由以下名称解析规则所建立的对该名称的 *绑定*。

4.2.2 名称的解析

*作用域*定义了一个代码块中名称的可见性。如果代码块中定义了一个局部变量，则其作用域包含该代码块。如果定义发生于函数代码块中，则其作用域会扩展到该函数所包含的任何代码块，除非有某个被包含代码块引入了对该名称的不同绑定。

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*。

当一个名称完全找不到时，将会引发 `NameError` 异常。如果当前作用域为函数作用域，且该名称指向一个局部变量，而此变量在该名称被使用的时候尚未绑定到特定值，将会引发 `UnboundLocalError` 异常。`UnboundLocalError` 为 `NameError` 的一个子类。

如果一个代码块内的任何位置发生名称绑定操作，则代码块内所有对该名称的使用都会被视为对当前代码块的引用。当一个名称在其被绑定前就在代码块内被使用时将会导致错误。这个规则是很微妙的。Python 缺少声明语法并且允许名称绑定操作发生于代码块内的任何位置。一个代码块的局部变量可通过在整个代码块文本中扫描名称绑定操作来确定。请参阅 UnboundLocalError 的 FAQ 条目来获取示例。

如果*global* 语句出现在一个代码块中，则所有对该语句所指定名称的使用都是在最高层级命名空间内对该名称绑定的引用。名称在最高层级命名空间内的解析是通过搜索全局命名空间，也就是包含该代码块的模块的命名空间，以及内置命名空间即 `builtins` 模块的命名空间。全局命名空间会先被搜索。如果未在其中找到相应名称，将再搜索内置命名空间。如果未在内置命名空间中找到相应名称，将在全局命名空间中创建新变量。global 语句必须位于所有对其所列名称的使用之前。

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。

nonlocal 语句会使得相应的名称指向之前在最近包含函数作用域中绑定的变量。如果指定的名称不存在于任何包含函数作用域中则将在编译时引发 `SyntaxError`。*类型形参* 不能使用 `nonlocal` 语句来重新绑定。

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__`。

类定义代码块以及传给 `exec()` 和 `eval()` 的参数是名称解析的上下文中的特殊情况。类定义是可能使用并定义名称的可执行语句。这些引用遵循正常的名称解析规则，例外之处在于未绑定的局部变量会在全局命名空间中查找。类定义的命名空间会成为该类的属性字典。在类代码块中定义的名称的作用域会被限制在类代码块中；它不会扩展到方法的代码块中。这包括推导式和生成器表达式，但不包括*标注作用域*，因为它可以访问所包含的类作用域。这意味着以下代码将会失败:

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

但是，下面的代码将会成功:

```
class A:
    type Alias = Nested
    class Nested: pass

print(A.Alias.__value__)  # <type 'A.Nested'>
```

4.2.3 标注作用域

类型形参列表 和type 语句引入了 标注作用域，其行为很像函数作用域，但具有下述的几处例外。标注目前没有使用标注作用域，但它们预期会在实现了 **PEP 649** 的 Python 3.13 中使用标注作用域。

标注作用域将在下列情况中使用:

- 针对泛型类型别名 的类型形参列表。
- 针对泛型函数 的类型形参列表。泛型函数的标注会在标注作用域内执行，但其默认值和装饰器则不会。
- 针对泛型类 的类型形参列表。泛型类的基类和关键字参数会在标注作用域内执行，但其装饰器则不会。
- 针对类型形参的绑定、约束和默认值 (惰性求值)。
- 类型别名的值 (惰性求值)。

标注作用域在以下几个方面不同于函数作用域:

- 标注作用域能够访问其所包含的类命名空间。如果某个标注作用域紧接在一个类作用域之内，或是位于紧接一个类作用域的另一个标注作用域之内，则该标注作用域中的代码将能使用在该类作用域中定义的名称，就像它是在该类内部直接执行一样。这不同于在类中定义的常规函数，后者无法访问在类作用域中定义的名称。
- 标注作用域中的表达式不能包含*yield*, `yield from`, *await* 或 *:=* 表达式。(这些表达式在包含于标注作用域之内的其他作用域中则是允许的。)
- 在标注作用域中定义的名称不能在内部作用域中通过*nonlocal* 语句来重新绑定。这只包括类型形参，因为没有其他可以在标注作用域内部出现的语法元素能够引入新的名称。
- 虽然标注作用域具有一个内部名称，但该名称不会反映在作用域内定义的对象的*qualified name* 中。相反，这些对象的 `__qualname__` 就像它们是定义在包含作用域中的对象一样。

Added in version 3.12: 标注作用域是在 Python 3.12 中作为 **PEP 695** 的一部分引入的。

在 3.13 版本发生变更: 标注作用域也被用于类型形参默认值，这是由 **PEP 696** 引入的。

4.2.4 惰性求值

通过type 语句创建的类型别名的值将被 惰性求值。此特性也适用于通过类型形参语法 创建的类型变量的绑定、约束和默认值。这意味着它们在创建类型别名或类型变量时不会被求值。相反，它们只有在需要处理属性访问时才会被求值。

示例:

```
>>> type Alias = 1/0
>>> Alias.__value__
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
>>> def func[T: 1/0](): pass
>>> T = func.__type_params__[0]
>>> T.__bound__
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
```

此处的异常只有在类型别名的 `__value__` 属性或类型变量的 `__bound__` 属性被访问时才会被引发。

此行为主要适用于当创建类型别名或类型变量时对尚未被定义的类型进行引用。例如，惰性求值将允许创建相互递归的类型别名:

```
from typing import Literal

type SimpleExpr = int | Parenthesized
```

(续下页)

(接上页)

```
type Parenthesized = tuple[Literal["("], Expr, Literal[")"]]
type Expr = SimpleExpr | tuple[SimpleExpr, Literal["+", "-"], Expr]
```

被惰性求值的值是在标记作用域内进行求值的，这意味着出现在被惰性求值的值内部的名称的查找范围就相当于它们是在紧邻的作用域中被使用。

Added in version 3.12.

4.2.5 内置命名空间和受限的执行

CPython 实现细节： 用户不应该接触 `__builtins__`，严格说来它属于实现细节。用户如果要重载内置命名空间中的值则应该 *import* `builtins` 并相应地修改该模块中的属性。

与一个代码块的执行相关联的内置命名空间实际上是通过在其全局命名空间中搜索名称 `__builtins__` 来找到的；这应该是一个字典或一个模块（在后一种情况下会使用该模块的字典）。默认情况下，当在 `__main__` 模块中时，`__builtins__` 就是内置模块 `builtins`；当在任何其他模块中时，`__builtins__` 则是 `builtins` 模块自身的字典的一个别名。

4.2.6 与动态特性的交互

自由变量的名称解析发生于运行时而不是编译时。这意味着以下代码将打印出 42:

```
i = 10
def f():
    print(i)
i = 42
f()
```

`eval()` 和 `exec()` 函数没有对完整环境的访问权限来解析名称。名称可以在调用者的局部和全局命名空间中被解析。自由变量的解析不是在最近包含命名空间中，而是在全局命名空间中。[1] `exec()` 和 `eval()` 函数有可选参数用来重载全局和局部命名空间。如果只指定一个命名空间，则它会同时作用于两者。

4.3 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）。

当一个异常完全未被处理时，解释器会终止程序的执行，或者返回交互模式的主循环。无论是哪种情况，它都会打印栈回溯信息，除非是当异常为 `SystemExit` 的时候。

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非虚基类。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。

> **备注**
>
> 异常消息不是 Python API 的组成部分。其内容可能在 Python 升级到新版本时不经警告地发生改变，不应该被需要在多版本解释器中运行的代码所依赖。

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

[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。

备注

CHAPTER 5

导入系统

一个*module* 内的 Python 代码通过*importing* 操作就能够访问另一个模块内的代码。*import* 语句是发起调用导入机制的最常用方式，但不是唯一的方式。`importlib.import_module()` 以及内置的 `__import__()` 等函数也可以被用来发起调用导入机制。

import 语句结合了两个操作；它先搜索指定名称的模块，然后将搜索结果绑定到当前作用域中的名称。`import` 语句的搜索操作被定义为对 `__import__()` 函数的调用并带有适当的参数。`__import__()` 的返回值会被用于执行 `import` 语句的名称绑定操作。请参阅 `import` 语句了解名称绑定操作的更多细节。

对 `__import__()` 的直接调用将仅执行模块搜索以及在找到时的模块创建操作。不过也可能产生某些副作用，例如导入父包和更新各种缓存 (包括 `sys.modules`)，只有*import* 语句会执行名称绑定操作。

当*import* 语句被执行时，标准的内置 `__import__()` 函数会被调用。其他发起调用导入系统的机制 (例如 `importlib.import_module()`) 可能会选择绕过 `__import__()` 并使用它们自己的解决方案来实现导入机制。

当一个模块首次被导入时，Python 会搜索该模块，如果找到就创建一个 module 对象[1] 并初始化它。如果指定名称的模块未找到，则会引发 `ModuleNotFoundError`。当发起调用导入机制时，Python 会实现多种策略来搜索指定名称的模块。这些策略可以通过使用使用下文所描述的多种钩子来加以修改和扩展。

在 3.3 版本发生变更: 导入系统已被更新以完全实现 **PEP 302** 中的第二阶段要求。不会再有任何隐式的导入机制——整个导入系统都通过 `sys.meta_path` 暴露出来。此外，对原生命名空间包的支持也已被实现 (参见 **PEP 420**)。

5.1 importlib

`importlib` 模块提供了一个丰富的 API 用来与导入系统进行交互。例如 `importlib.import_module()` 提供了相比内置的 `__import__()` 更推荐、更简单的 API 用来发起调用导入机制。更多细节请参看 `importlib` 库文档。

5.2 包

Python 只有一种模块对象类型，所有模块都属于该类型，无论模块是用 Python、C 还是别的语言实现。为了帮助组织模块并提供名称层次结构，Python 还引入了包 的概念。

[1] 参见 `types.ModuleType`。

你可以把包看成是文件系统中的目录，并把模块看成是目录中的文件，但请不要对这个类比做过于字面的理解，因为包和模块不是必须来自于文件系统。为了方便理解本文档，我们将继续使用这种目录和文件的类比。与文件系统一样，包通过层次结构进行组织，在包之内除了一般的模块，还可以有子包。

要注意的一个重点概念是所有包都是模块，但并非所有模块都是包。或者换句话说，包只是一种特殊的模块。特别地，任何具有 `__path__` 属性的模块都会被当作是包。

所有模块都有自己的名字。子包名与其父包名会以点号分隔，与 Python 的标准属性访问语法一致。因此你可能会有一个名为 `email` 的包，这个包中又有一个名为 `email.mime` 的子包以及该子包中的名为 `email.mime.text` 的子包。

5.2.1 常规包

Python 定义了两种类型的包，*常规包* 和*命名空间包*。常规包是传统的包类型，它们在 Python 3.2 及之前就已存在。常规包通常以一个包含 `__init__.py` 文件的目录形式实现。当一个常规包被导入时，这个 `__init__.py` 文件会隐式地被执行，它所定义的对象会被绑定到该包命名空间中的名称。`__init__.py` 文件可以包含与任何其他模块中所包含的 Python 代码相似的代码，Python 将在模块被导入时为其添加额外的属性。

例如，以下文件系统布局定义了一个最高层级的 `parent` 包和三个子包:

```
parent/
    __init__.py
    one/
        __init__.py
    two/
        __init__.py
    three/
        __init__.py
```

导入 `parent.one` 将隐式地执行 `parent/__init__.py` 和 `parent/one/__init__.py`。后续导入 `parent.two` 或 `parent.three` 则将分别执行 `parent/two/__init__.py` 和 `parent/three/__init__.py`。

5.2.2 命名空间包

命名空间包是由多个*部分* 构成的，每个部分为父包增加一个子包。各个部分可能处于文件系统的不同位置。部分也可能处于 zip 文件中、网络上，或者 Python 在导入期间可以搜索的其他地方。命名空间包并不一定会直接对应到文件系统中的对象；它们有可能是无实体表示的虚拟模块。

命名空间包的 `__path__` 属性不使用普通的列表。而是使用定制的可迭代类型，如果其父包的路径 (或者最高层级包的 `sys.path`) 发生改变，这种对象会在该包内的下一次导入尝试时自动执行新的对包部分的搜索。

命名空间包没有 `parent/__init__.py` 文件。实际上，在导入搜索期间可能找到多个 `parent` 目录，每个都由不同的部分所提供。因此 `parent/one` 的物理位置不一定与 `parent/two` 相邻。在这种情况下，Python 将为顶级的 `parent` 包创建一个命名空间包，无论是它本身还是它的某个子包被导入。

另请参阅 **PEP 420** 了解对命名空间包的规格描述。

5.3 搜索

为了开始搜索，Python 需要被导入模块（或者包，对于当前讨论来说两者没有差别）的完整*限定名称*。此名称可以来自 *import* 语句所带的各种参数，或者来自传给 `importlib.import_module()` 或 `__import__()` 函数的形参。

此名称会在导入搜索的各个阶段被使用，它也可以是指向一个子模块的带点号路径，例如 `foo.bar.baz`。在这种情况下，Python 会先尝试导入 `foo`，然后是 `foo.bar`，最后是 `foo.bar.baz`。如果这些导入中的任何一个失败，都会引发 `ModuleNotFoundError`。

5.3.1 模块缓存

在导入搜索期间首先会被检查的地方是 sys.modules。这个映射起到缓存之前导入的所有模块的作用（包括其中间路径）。因此如果之前导入过 foo.bar.baz，则 sys.modules 将包含 foo, foo.bar 和 foo.bar.baz 条目。每个键的值就是相应的模块对象。

在导入期间，会在 sys.modules 查找模块名称，如存在则其关联的值就是需要导入的模块，导入过程完成。然而，如果值为 None，则会引发 ModuleNotFoundError。如果找不到指定模块名称，Python 将继续搜索该模块。

sys.modules 是可写的。删除键可能不会破坏关联的模块（因为其他模块可能会保留对它的引用），但它会使命名模块的缓存条目无效，导致 Python 在下次导入时重新搜索命名模块。键也可以赋值为 None，强制下一次导入模块导致 ModuleNotFoundError 。

但是要小心，因为如果你还保有对某个模块对象的引用，同时停用其在 sys.modules 中的缓存条目，然后又再次导入该名称的模块，则前后两个模块对象将 不是同一个。相反地，importlib.reload() 将重用 同一个模块对象，并简单地通过重新运行模块的代码来重新初始化模块内容。

5.3.2 查找器和加载器

如果指定名称的模块在 sys.modules 找不到，则将发起调用 Python 的导入协议以查找和加载该模块。此协议由两个概念性模块构成，即查找器 和加载器。查找器的任务是确定是否能使用其所知的策略找到该名称的模块。同时实现这两种接口的对象称为导入器 ——它们在确定能加载所需的模块时会返回其自身。

Python 包含了多个默认查找器和导入器。第一个知道如何定位内置模块，第二个知道如何定位冻结模块。第三个默认查找器会在*import path* 中搜索模块。*import path* 是一个由文件系统路径或 zip 文件组成的位置列表。它还可以扩展为搜索任意可定位资源，例如由 URL 指定的资源。

导入机制是可扩展的，因此可以加入新的查找器以扩展模块搜索的范围和作用域。

查找器并不真正加载模块。如果它们能找到指定名称的模块，会返回一个 模块规格说明，这是对模块导入相关信息的封装，供后续导入机制用于在加载模块时使用。

以下各节描述了有关查找器和加载器协议的更多细节，包括你应该如何创建并注册新的此类对象来扩展导入机制。

在 3.4 版本发生变更: 在之前的 Python 版本中，查找器会直接返回加载器，现在它们则返回模块规格说明，其中 包含加载器。加载器仍然在导入期间被使用，但负担的任务有所减少。

5.3.3 导入钩子

导入机制被设计为可扩展；其中的基本机制是 导入钩子。导入钩子有两种类型: 元钩子和 导入路径钩子。

元钩子在导入过程开始时被调用，此时任何其他导入过程尚未发生，但 sys.modules 缓存查找除外。这允许元钩子重载 sys.path 过程、冻结模块甚至内置模块。元钩子的注册是通过向 sys.meta_path 添加新的查找器对象，具体如下所述。

导入路径钩子是作为 sys.path (或 package.__path__) 过程的一部分，在遇到它们所关联的路径项的时候被调用。导入路径钩子的注册是通过向 sys.path_hooks 添加新的可调用对象，具体如下所述。

5.3.4 元路径

当指定名称的模块在 sys.modules 中找不到时，Python 会接着搜索 sys.meta_path，其中包含元路径查找器对象列表。这些查找器将按顺序被查询以确定它们是否知道如何处理该名称的模块。元路径查找器必须实现名为 find_spec() 的方法，它接受三个参数：名称、导入路径和（可选的）目标模块。元路径查找器可使用任何策略来确定它是否能处理指定名称的模块。

如果元路径查找器知道如何处理指定名称的模块，它将返回一个说明对象。如果它不能处理该名称的模块，则会返回 None。如果 sys.meta_path 处理过程到达列表末尾仍未返回说明对象，则将引发 ModuleNotFoundError。任何其他被引发异常将直接向上传播，并放弃导入过程。

元路径查找器的 find_spec() 方法调用将附带两个或三个参数。第一个是被导入模块的完整限定名称，例如 foo.bar.baz。第二个参数是供模块搜索使用的路径条目。对于最高层级模块，第二个参数为 None，

但对于子模块或子包，第二个参数为父包的 `__path__` 属性的值。如果相应折 `__path__` 属性无法访问，则会引发 `ModuleNotFoundError`。第三个参数是将被作为稍后加载目标的现有模块对象。导入系统仅会在重加载期间传入一个目标模块。

对于单个导入请求可以多次遍历元路径。例如，假设所涉及的模块都尚未被缓存，则导入 `foo.bar.baz` 将首先执行顶级的导入，在每个元路径查找器 (`mpf`) 上调用 `mpf.find_spec("foo", None, None)`。在导入 `foo` 之后，`foo.bar` 将通过第二次遍历元路径来导入，调用 `mpf.find_spec("foo.bar", foo.__path__, None)`。一旦 `foo.bar` 完成导入，最后一次遍历将调用 `mpf.find_spec("foo.bar.baz", foo.bar.__path__, None)`。

有些元路径查找器只支持顶级导入。当把 `None` 以外的对象作为第三个参数传入时，这些导入器将总是返回 `None`。

Python 的默认 `sys.meta_path` 具有三种元路径查找器，一种知道如何导入内置模块，一种知道如何导入冻结模块，还有一种知道如何导入来自*import path* 的模块 (即*path based finder*)。

在 3.4 版本发生变更: 元路径查找器的 `find_spec()` 方法替代了 `find_module()`，后者现已被弃用。虽然它仍将可以不加修改地继续使用，但导入机制仅会在查找器未实现 `find_spec()` 时尝试使用它。

在 3.10 版本发生变更: 导入系统使用 `find_module()` 现在将引发 `ImportWarning`。

在 3.12 版本发生变更: `find_module()` 已被移除。请改用 `find_spec()`。

5.4 加载

当一个模块说明被找到时，导入机制将在加载该模块时使用它（及其所包含的加载器）。下面是导入的加载部分所发生过程的简要说明:

```
module = None
if spec.loader is not None and hasattr(spec.loader, 'create_module'):
    # 假定 'exec_module' 也将在该加载器上定义。
    module = spec.loader.create_module(spec)
if module is None:
    module = ModuleType(spec.name)
# 导入相关的模块属性在此设置:
_init_module_attrs(spec, module)

if spec.loader is None:
    # 不受支持
    raise ImportError
if spec.origin is None and spec.submodule_search_locations is not None:
    # 命名空间包
    sys.modules[spec.name] = module
elif not hasattr(spec.loader, 'exec_module'):
    module = spec.loader.load_module(spec.name)
else:
    sys.modules[spec.name] = module
    try:
        spec.loader.exec_module(module)
    except BaseException:
        try:
            del sys.modules[spec.name]
        except KeyError:
            pass
        raise
return sys.modules[spec.name]
```

请注意以下细节:

- 如果在 `sys.modules` 中存在指定名称的模块对象，导入操作会已经将其返回。
- 在加载器执行模块代码之前，该模块将存在于 `sys.modules` 中。这一点很关键，因为该模块代码可能（直接或间接地）导入其自身；预先将其添加到 `sys.modules` 可防止在最坏情况下的无限递归和最好情况下的多次加载。

- 如果加载失败，则该模块 -- 只限加载失败的模块 -- 将从 sys.modules 中移除。任何已存在于 sys.modules 缓存的模块，以及任何作为附带影响被成功加载的模块仍会保留在缓存中。这与重新加载不同，后者会把即使加载失败的模块也保留在 sys.modules 中。
- 在模块创建完成但还未执行之前，导入机制会设置导入相关模块属性（在上面的示例伪代码中为“_init_module_attrs”），详情参见后续部分。
- 模块执行是加载的关键时刻，在此期间将填充模块的命名空间。执行会完全委托给加载器，由加载器决定要填充的内容和方式。
- 在加载过程中创建并传递给 exec_module() 的模块并不一定就是在导入结束时返回的模块[2]。

在 3.4 版本发生变更: 导入系统已经接管了加载器建立样板的责任。这些在以前是由 importlib.abc.Loader.load_module() 方法来执行的。

5.4.1 加载器

模块加载器提供关键的加载功能：模块执行。导入机制调用 importlib.abc.Loader.exec_module() 方法并传入一个参数来执行模块对象。从 exec_module() 返回的任何值都将被忽略。

加载器必须满足下列要求:

- 如果模块是一个 Python 模块（而非内置模块或动态加载的扩展），加载器应该在模块的全局命名空间 (module.__dict__) 中执行模块的代码。
- 如果加载器无法执行指定模块，它应该引发 ImportError，不过在 exec_module() 期间引发的任何其他异常也会被传播。

在许多情况下，查找器和加载器可以是同一对象；在此情况下 find_spec() 方法将返回一个规格说明，其中加载器会被设为 self。

模块加载器可以选择通过实现 create_module() 方法在加载期间创建模块对象。它接受一个参数，即模块规格说明，并返回新的模块对象供加载期间使用。create_module() 不需要在模块对象上设置任何属性。如果模块返回 None，导入机制将自行创建新模块。

Added in version 3.4: 加载器的 create_module() 方法。

在 3.4 版本发生变更: load_module() 方法被 exec_module() 所替代，导入机制会对加载的所有样板责任作出假定。

为了与现有的加载器兼容，导入机制会使用导入器的 load_module() 方法，如果它存在且导入器也未实现 exec_module()。但是，load_module() 现已弃用，加载器应该转而实现 exec_module()。

除了执行模块之外，load_module() 方法必须实现上文描述的所有样板加载功能。所有相同的限制仍然适用，并带有一些附加规定:

- 如果 sys.modules 中存在指定名称的模块对象，加载器必须使用已存在的模块。(否则 importlib.reload() 将无法正确工作。) 如果该名称模块不存在于 sys.modules 中，加载器必须创建一个新的模块对象并将其加入 sys.modules。
- 在加载器执行模块代码之前，模块 *必须* 存在于 sys.modules 之中，以防止无限递归或多次加载。
- 如果加载失败，加载器必须移除任何它已加入到 sys.modules 中的模块，但它必须 **仅限**移除加载失败的模块，且所移除的模块应为加载器自身显式加载的。

在 3.5 版本发生变更: 当 exec_module() 已定义但 create_module() 未定义时将引发 DeprecationWarning。

在 3.6 版本发生变更: 当 exec_module() 已定义但 create_module() 未定义时将引发 ImportError。

在 3.10 版本发生变更: 使用 load_module() 将引发 ImportWarning。

[2] importlib 实现避免直接使用返回值。而是通过在 sys.modules 中查找模块名称来获取模块对象。这种方式的间接影响是被导入的模块可能在 sys.modules 中替换其自身。这属于具体实现的特定行为，不保证能在其他 Python 实现中起作用。

5.4.2 子模块

当使用任意机制 (例如 `importlib` API, `import` 及 `import-from` 语句或者内置的 `__import__()`) 加载一个子模块时，父模块的命名空间中会添加一个对子模块对象的绑定。例如，如果包 `spam` 有一个子模块 `foo`，则在导入 `spam.foo` 之后，`spam` 将具有一个绑定到相应子模块的 `foo` 属性。假如现在有如下的目录结构:

```
spam/
    __init__.py
    foo.py
```

并且 `spam/__init__.py` 中有如下几行内容:

```
from .foo import Foo
```

那么执行如下代码将把 `foo` 和 `Foo` 的名称绑定添加到 `spam` 模块中:

```
>>> import spam
>>> spam.foo
<module 'spam.foo' from '/tmp/imports/spam/foo.py'>
>>> spam.Foo
<class 'spam.foo.Foo'>
```

按照通常的 Python 名称绑定规则，这看起来可能会令人惊讶，但它实际上是导入系统的一个基本特性。保持不变的一点是如果你有 `sys.modules['spam']` 和 `sys.modules['spam.foo']` (例如在上述导入之后就是如此)，则后者必须显示为前者的 `foo` 属性。

5.4.3 模块规格说明

导入机制在导入期间会使用有关每个模块的多种信息，特别是加载之前。大多数信息都是所有模块通用的。模块规格说明的目的是基于每个模块来封装这些导入相关信息。

在导入期间使用规格说明可允许状态在导入系统各组件之间传递，例如在创建模块规格说明的查找器和执行模块的加载器之间。最重要的一点是，它允许导入机制执行加载的样板操作，在没有模块规格说明的情况下这是加载器的责任。

模块的规格说明将作为*module.__spec__* 公开。正确设置 `__spec__` 将同时应用于*解释器启动期间初始化的模块*。唯一的例外是 `__main__`，其中的 `__spec__` 会*在某些情况下设为 None*。

请参阅 `ModuleSpec` 了解有关模块规格的详细内容。

Added in version 3.4.

5.4.4 模块的 __path__ 属性

__path__ 属性应当是一个（可能为空的）枚举将用于查找包的子模块的位置的字符串*sequence*。根据定义，如果一个模块具有 `__path__` 属性，它就是一个*package*。

包的*__path__* 属性会在导入其子包期间被使用。在导入机制内部，它的功能与 `sys.path` 基本相同，即在导入期间提供一个模拟搜索位置列表。但是，`__path__` 相比 `sys.path` 通常要受到更多约束。

作用于 `sys.path` 的规则同样适用于包的 `__path__`。`sys.path_hooks` (见下文) 会在遍历包的 `__path__` 时被查询。

包的 `__init__.py` 文件可以设置或更改包的*__path__* 属性，而且这是在 **PEP 420** 之前实现命名空间包的典型方式。随着 **PEP 420** 的引入，命名空间包不再需要提供仅包含 `__path__` 操控代码的 `__init__.py` 文件；导入机制会自动为命名空间包正确地设置 `__path__`。

5.4.5 模块的 repr

默认情况下，全部模块都具有一个可用的 repr，但是你可以依据上述的属性设置，在模块的规格说明中更为显式地控制模块对象的 repr。

如果模块具有 spec (__spec__)，导入机制将尝试用它来生成一个 repr。如果生成失败或找不到 spec，导入系统将使用模块中的各种可用信息来制作一个默认 repr。它将尝试使用 module.__name__, module.__file__ 以及 module.__loader__ 作为 repr 的输入，并将任何丢失的信息赋为默认值。

以下是所使用的确切规则:

- 如果模块具有 __spec__ 属性，其中的规格信息会被用来生成 repr。被查询的属性有"name", "loader", "origin" 和"has_location" 等等。
- 如果模块具有 __file__ 属性，这会被用作模块 repr 的一部分。
- 如果模块没有 __file__ 但是有 __loader__ 且取值不为 None，则加载器的 repr 会被用作模块 repr 的一部分。
- 对于其他情况，仅在 repr 中使用模块的 __name__。

在 3.12 版本发生变更: module_repr() 自 Python 3.4 起已被弃用，在 Python 3.12 中已被移除且不会在模块的 repr 计算期间被调用。

5.4.6 已缓存字节码的失效

在 Python 从 .pyc 文件加载已缓存字节码之前，它会检查缓存是否由最新的 .py 源文件所生成。默认情况下，Python 通过在所写入缓存文件中保存源文件的最新修改时间戳和大小来实现这一点。在运行时，导入系统会通过比对缓存文件中保存的元数据和源文件的元数据确定该缓存的有效性。

Python 也支持"基于哈希的"缓存文件，即保存源文件内容的哈希值而不是其元数据。存在两种基于哈希的 .pyc 文件：检查型和非检查型。对于检查型基于哈希的 .pyc 文件，Python 会通过求哈希源文件并将结果哈希值与缓存文件中的哈希值比对来确定缓存有效性。如果检查型基于哈希的缓存文件被确定为失效，Python 会重新生成并写入一个新的检查型基于哈希的缓存文件。对于非检查型 .pyc 文件，只要其存在 Python 就会直接认定缓存文件有效。确定基于哈希的 .pyc 文件有效性的行为可通过 --check-hash-based-pycs 旗标来重载。

在 3.7 版本发生变更: 增加了基于哈希的 .pyc 文件。在此之前，Python 只支持基于时间戳来确定字节码缓存的有效性。

5.5 基于路径的查找器

在之前已经提及，Python 带有几种默认的元路径查找器。其中之一是*path based finder* (PathFinder)，它会搜索包含一个路径条目 列表的*import path*。每个路径条目指定一个用于搜索模块的位置。

基于路径的查找器自身并不知道如何进行导入。它只是遍历单独的路径条目，将它们各自关联到某个知道如何处理特定类型路径的路径条目查找器。

默认的路径条目查找器集合实现了在文件系统中查找模块的所有语义，可处理多种特殊文件类型例如 Python 源码 (.py 文件)，Python 字节码 (.pyc 文件) 以及共享库 (例如 .so 文件)。在标准库中 zipimport 模块的支持下，默认路径条目查找器还能处理所有来自 zip 文件的上述文件类型。

路径条目不必仅限于文件系统位置。它们可以指向 URL、数据库查询或可以用字符串指定的任何其他位置。

基于路径的查找器还提供了额外的钩子和协议以便能扩展和定制可搜索路径条目的类型。例如，如果你想要支持网络 URL 形式的路径条目，你可以编写一个实现 HTTP 语义在网络上查找模块的钩子。这个钩子（可调用对象）应当返回一个支持下述协议的*path entry finder*，以被用来获取一个专门针对来自网络的模块的加载器。

预先的警告：本节和上节都使用了 查找器这一术语，并通过*meta path finder* 和*path entry finder* 两个术语来明确区分它们。这两种类型的查找器非常相似，支持相似的协议，且在导入过程中以相似的方式运作，但关键的一点是要记住它们是有微妙差异的。特别地，元路径查找器作用于导入过程的开始，主要是启动 sys.meta_path 遍历。

相比之下，路径条目查找器在某种意义上说是基于路径的查找器的实现细节，实际上，如果需要从 sys.meta_path 移除基于路径的查找器，并不会有任何路径条目查找器被发起调用。

5.5.1 路径条目查找器

path based finder 会负责查找和加载通过*path entry* 字符串来指定位置的 Python 模块和包。多数路径条目所指定的是文件系统中的位置，但它们并不必受限于此。

作为一种元路径查找器，*path based finder* 实现了上文描述的 `find_spec()` 协议，但是它还对外公开了一些附加钩子，可被用来定制模块如何从*import path* 查找和加载。

有三个变量由*path based finder*, `sys.path`, `sys.path_hooks` 和 `sys.path_importer_cache` 所使用。包对象的 `__path__` 属性也会被使用。它们提供了可用于定制导入机制的额外方式。

`sys.path` 包含一个提供模块和包搜索位置的字符串列表。它初始化自 `PYTHONPATH` 环境变量以及多种其他特定安装和实现专属的默认位置。`sys.path` 中的条目可指定文件系统中的目录、zip 文件及其他可用于搜索模块的潜在“位置“(参见 `site` 模块)，例如 URL 或数据库查询等。在 `sys.path` 中应当只有字符串；所有其他数据类型都会被忽略。

path based finder 是一种*meta path finder*，因此导入机制会通过调用上文描述的基于路径的查找器的 `find_spec()` 方法来启动*import path* 搜索。当要向 `find_spec()` 传入 `path` 参数时，它将是一个可遍历的字符串列表——通常为用来在其内部进行导入的包的 `__path__` 属性。如果 `path` 参数为 `None`，这表示最高层级的导入，将会使用 `sys.path`。

基于路径的查找器会迭代搜索路径中的每个条目，并且每次都查找与路径条目对应的*path entry finder* (`PathEntryFinder`)。因为这种操作可能很耗费资源 (例如搜索会有 `stat()` 调用的开销)，基于路径的查找器会维持一个将路径条目映射到路径条目查找器的缓存。这个缓存是在 `sys.path_importer_cache` 中维护的 (尽管如此命名，但这个缓存实际存放的是查找器对象而非仅限于*importer* 对象)。通过这种方式，对特定*path entry* 位置的*path entry finder* 的高耗费搜索只需进行一次。用户代码可以自由地从 `sys.path_importer_cache` 移除缓存条目以强制基于路径的查找器再次执行路径条目搜索。

如果路径条目不存在于缓存中，基于路径的查找器会迭代 `sys.path_hooks` 中的每个可调用对象。对此列表中的每个*路径条目钩子* 的调用会带有一个参数，即要搜索的路径条目。每个可调用对象或是返回可处理路径条目的*path entry finder*，或是引发 `ImportError`。基于路径的查找器使用 `ImportError` 来表示钩子无法找到与*path entry* 相对应的*path entry finder*。该异常会被忽略并继续进行*import path* 的迭代。每个钩子应该期待接收一个字符串或字节串对象；字节串对象的编码由钩子决定（例如可以是文件系统使用的编码 UTF-8 或其它编码)，如果钩子无法解码参数，它应该引发 `ImportError`。

如果 `sys.path_hooks` 迭代结束时没有返回*path entry finder*，则基于路径的查找器 `find_spec()` 方法将在 `sys.path_importer_cache` 中存入 `None` (表示此路径条目没有对应的查找器) 并返回 `None`，表示此*meta path finder* 无法找到该模块。

如果 `sys.path_hooks` 中的某个*path entry hook* 可调用对象的返回值 是一个*path entry finder*，则以下协议会被用来向查找器请求一个模块的规格说明，并在加载该模块时被使用。

当前工作目录 -- 由一个空字符串表示 -- 的处理方式与 `sys.path` 中的其他条目略有不同。首先，如果发现当前工作目录不存在，则 `sys.path_importer_cache` 中不会存放任何值。其次，每个模块查找会对当前工作目录的值进行全新查找。第三，由 `sys.path_importer_cache` 所使用并由 `importlib.machinery.PathFinder.find_spec()` 所返回的路径将是实际的当前工作目录而非空字符串。

5.5.2 路径条目查找器协议

为了支持模块和已初始化包的导入，也为了给命名空间包提供组成部分，路径条目查找器必须实现 `find_spec()` 方法。

`find_spec()` 接受两个参数，即要导入模块的完整限定名称，以及（可选的）目标模块。`find_spec()` 返回模块的完全填充好的规格说明。这个规格说明总是包含“加载器”集合（但有一个例外)。

为了向导入机制提示该规格说明代表一个命名空间*portion*，路径条目查找器会将 `submodule_search_locations` 设为一个包含该部分的列表。

在 3.4 版本发生变更: `find_spec()` 替代了 `find_loader()` 和 `find_module()`，这两者现在都已被弃用，但会在 `find_spec()` 未定义时被使用。

较旧的路径条目查找器可能会实现这两个已弃用的方法中的一个而没有实现 `find_spec()`。为保持向后兼容，这两个方法仍会被接受。但是，如果在路径条目查找器上实现了 `find_spec()`，这两个遗留方法就会被忽略。

`find_loader()` 接受一个参数，即要导入模块的完整限定名称。`find_loader()` 返回一个 2 元组，其中第一项是加载器而第二项是命名空间*portion*。

为了向后兼容其他导入协议的实现，许多路径条目查找器也同样支持元路径查找器所支持的传统 `find_module()` 方法。但是路径条目查找器 `find_module()` 方法的调用绝不会带有 `path` 参数（它们被期望记录来自对路径钩子初始调用的恰当路径信息）。

路径条目查找器的 `find_module()` 方法已弃用，因为它不允许路径条目查找器为命名空间包提供部分。如果 `find_loader()` 和 `find_module()` 同时存在于一个路径条目查找器中，导入系统将总是调用 `find_loader()` 而不选择 `find_module()`。

在 3.10 版本发生变更: 导入系统对 `find_module()` 和 `find_loader()` 的调用将引发 `ImportWarning`。

在 3.12 版本发生变更: `find_module()` 和 `find_loader()` 已被移除。

5.6 替换标准导入系统

替换整个导入系统的最可靠机制是移除 `sys.meta_path` 的默认内容，，将其完全替换为自定义的元路径钩子。

一个可行的方式是仅改变导入语句的行为而不影响访问导入系统的其他 API，那么替换内置的 `__import__()` 函数可能就够了。这种技巧也可以在模块层级上运用，即只在某个模块内部改变导入语句的行为。

想要选择性地预先防止在元路径上从一个钩子导入某些模块（而不是完全禁用标准导入系统），只需直接从 `find_spec()` 引发 `ModuleNotFoundError` 而非返回 `None` 就足够了。返回后者表示元路径搜索应当继续，而引发异常则会立即终止搜索。

5.7 包相对导入

相对导入使用前缀点号。一个前缀点号表示相对导入从当前包开始。两个或更多前缀点号表示对当前包的上级包的相对导入，第一个点号之后的每个点号代表一级。例如，给定以下的包布局结构:

```
package/
    __init__.py
    subpackage1/
        __init__.py
        moduleX.py
        moduleY.py
    subpackage2/
        __init__.py
        moduleZ.py
    moduleA.py
```

不论是在 `subpackage1/moduleX.py` 还是 `subpackage1/__init__.py` 中，以下导入都是有效的:

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
```

绝对导入可以使用 `import <>` 或 `from <> import <>` 语法，但相对导入只能使用第二种形式；其中的原因在于:

```
import XXX.YYY.ZZZ
```

应当提供 `XXX.YYY.ZZZ` 作为可用表达式，但.moduleY 不是一个有效的表达式。

5.8 有关 __main__ 的特殊事项

对于 Python 的导入系统来说 __main__ 模块是一个特殊情况。正如在另一节 中所述，__main__ 模块是在解释器启动时直接初始化的，与 sys 和 builtins 很类似。但是，与那两者不同，它并不被严格归类为内置模块。这是因为 __main__ 被初始化的方式依赖于发起调用解释器所附带的旗标和其他选项。

5.8.1 __main__.__spec__

根据 __main__ 被初始化的方式，__main__.__spec__ 会被设置相应值或是 None。

当 Python 附加 -m 选项启动时，__spec__ 会被设为相应模块或包的模块规格说明。__spec__ 也会在 __main__ 模块作为执行某个目录，zip 文件或其它 sys.path 条目的一部分加载时被填充。

在 其余的情况下 __main__.__spec__ 会被设为 None，因为用于填充 __main__ 的代码不直接与可导入的模块相对应:

- 交互型提示
- -c 选项
- 从 stdin 运行
- 直接从源码或字节码文件运行

请注意在最后一种情况中 __main__.__spec__ 总是为 None，即使文件从技术上说可以作为一个模块被导入。如果想要让 __main__ 中的元数据生效，请使用 -m 开关。

还要注意即使是在 __main__ 对应于一个可导入模块且 __main__.__spec__ 被相应地设定时，它们仍会被视为 不同的模块。这是由于以下事实：使用 if __name__ == "__main__": 检测来保护的代码块仅会在模块被用来填充 __main__ 命名空间时而非普通的导入时被执行。

5.9 参考文献

导入机制自 Python 诞生之初至今已发生了很大的变化。原始的 包规格说明 仍然可以查阅，但在撰写该文档之后许多相关细节已被修改。

原始的 sys.meta_path 规格说明见 **PEP 302**，后续的扩展说明见 **PEP 420**。

PEP 420 为 Python 3.3 引入了命名空间包。**PEP 420** 还引入了 find_loader() 协议作为 find_module() 的替代。

PEP 366 描述了新增的 __package__ 属性，用于在模块中的显式相对导入。

PEP 328 引入了绝对和显式相对导入，并初次提出了 __name__ 语义，最终由 **PEP 366** 为 __package__ 加入规范描述。

PEP 338 定义了将模块作为脚本执行。

PEP 451 在 spec 对象中增加了对每个模块导入状态的封装。它还将加载器的大部分样板责任移交回导入机制中。这些改变允许弃用导入系统中的一些 API 并为查找器和加载器增加一些新的方法。

备注

CHAPTER 6

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。

6.1 算术转换

当对下述某个算术运算符的描述中使用了“数值参数被转换为普通类型”这样的说法，这意味着内置类型的运算符实现采用了如下运作方式:

- 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，另一参数将被转换为浮点数。
- 否则，两者应该都为整数，不需要进行转换。

某些附加规则会作用于特定运算符（例如，字符串作为'%' 运算符的左运算参数）。扩展必须定义它们自己的转换行为。

6.2 原子

“原子”指表达式的最基本构成元素。最简单的原子是标识符和字面值。以圆括号、方括号或花括号包括的形式在语法上也被归类为原子。原子的句法为:

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

6.2.1 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。

私有名称的 mangling

当类定义中出现的标识符，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，就称它为类的 *private name* 。

参见

类规范说明。

更具体地，私有名称在其字节码生成之前即被转为更长的名字。如果转换后的名字长于 255 字符，实现可以决定缩短。

这一转换过程和标识符使用的语法上下文无关，仅有以下几种私有标识符会被 mangle：

- 用作被分配或读取的变量的名字的，或者用作被访问的属性的名字的。

  但是嵌套的函数、类和类型别名的 `__name__` 属性不会被 mangle。

- 导入的模块的名称，例如 “import __spam“中的 “__spam“。若模块属于一个包（即它的名称中有点号），这个名称 不会被 mangle，比如 “import __foo.bar“中的 “__foo“不会被 mangle。
- 导入的成员的名称，比如 “from spam import __f“中的 “__f“。

转换规则的定义如下：

- 类名称，先移除全部的开头下划线并插入一个开头下划线，再插入到标识符的前面，例如出现在名为 `Foo`, `_Foo` 或 `__Foo` 类中的标识符 `__spam` 将被转换为 `_Foo__spam`。
- 如果类名称仅由下划线组成，则转换为标识符本身，例如出现在名为 `_` 或 `__` 类中的标识符 `__spam` 将保持原样。

6.2.2 字面值

Python 支持字符串和字节串字面值，以及几种数字字面值:

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

对字面值求值将返回一个该值所对应类型的对象（字符串、字节串、整数、浮点数、复数）。对于浮点数和虚数（复数）的情况，该值可能为近似值。详情参见字面值。

所有字面值都对应于不可变数据类型，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。

6.2.3 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   ::=   "(" [starred_expression] ")"
```

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。

一对内容为空的圆括号将产生一个空的元组对象。由于元组是不可变对象，因此适用与字面值相同的规则（即两次出现的空元组产生的对象可能相同也可能不同）。

请注意元组并不是由圆括号构建的，实际起作用的是逗号。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。

6.2.4 列表、集合与字典的显示

为了构建列表、集合或字典，Python 提供了名为“显示”的特殊句法，每个类型各有两种形式:

- 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   ::=   assignment_expression comp_for
comp_for        ::=   ["async"] "for" target_list "in" or_test [comp_iter]
comp_iter       ::=   comp_for | comp_if
comp_if         ::=   "if" or_test [comp_iter]
```

推导式的结构是一个单独表达式后面加至少一个 `for` 子句以及零个或更多个 `for` 或 `if` 子句。在这种情况下，新容器的元素产生方式是将每个 `for` 或 `if` 子句视为一个代码块，按从左至右的顺序嵌套，然后每次到达最内层代码块时就对表达式进行求值以产生一个元素。

不过，除了最左边 `for` 子句中的可迭代表达式，推导式是在另一个隐式嵌套的作用域内执行的。这能确保赋给目标列表的名称不会“泄露”到外层的作用域。

最左边的 `for` 子句中的可迭代对象表达式会直接在外层作用域中被求值，然后作为一个参数被传给隐式嵌套的作用域。后续的 `for` 子句以及最左侧 `for` 子句中的任何筛选条件不能在外层作用域中被求值，因为它们可能依赖于从最左侧可迭代对象中获得的值。例如: `[x*y for x in range(10) for y in range(x, x+10)]`。

为了确保推导式得出的结果总是一个类型正确的容器，在隐式嵌套作用域内禁止使用 `yield` 和 `yield from` 表达式。

从 Python 3.6 开始，在 *`async def`* 函数中，可以使用 `async for` 子句来迭代 *asynchronous iterator*。在 `async def` 函数中的推导式可以由打头的表达式后跟一个 `for` 或 `async for` 子句组成，并可能包含附加的 `for` 或 `async for` 子句，还可能使用 *`await`* 表达式。

如果一个推导式包含 `async for` 子句，或者如果它在最左侧的 `for` 子句中可迭代对象表达式以外的任何地方包含 `await` 表达式或其他异步推导式，那它就被称为 *asynchronous comprehension*。异步推导式可以挂起它所在的协程函数的执行。另请参阅 **PEP 530**。

Added in version 3.6: 引入了异步推导式。

在 3.8 版本发生变更: `yield` 和 `yield from` 在隐式嵌套的作用域中已被禁用。

在 3.11 版本发生变更: 现在允许在异步函数的推导式中使用异步推导式。外部推导式将隐式地转为异步的。

6.2.5 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   ::=   "[" [flexible_expression_list | comprehension] "]"
```

列表显示会产生一个新的列表对象，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按此顺序放入列表对象。当提供一个推导式时，列表会根据推导式所产生的结果元素进行构建。

6.2.6 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  ::=  "{" (flexible_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。

6.2.7 字典显示

字典显示是一个用花括号括起来的可能为空的字典条目（键/值对）系列:

```
dict_display        ::=  "{" [dict_item_list | dict_comprehension] "}"
dict_item_list      ::=  dict_item ("," dict_item)* [","]
dict_item           ::=  expression ":" expression | "**" or_expr
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字典条目序列，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字典中存放相应值的键。这意味着你可以在字典条目列表中多次指定相同的键，而最终字典的值将由最后一次给出的键决定。

双星号 `**` 表示 *字典拆包*。它的操作数必须是一个*mapping*。每个映射项会被加入到新的字典。后续的值会替换先前的字典项和先前的字典拆包所设置的值。

Added in version 3.5: 拆包到字典显示，最初由 **PEP 448** 提出。

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。

对键的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，键的类型应为*hashable*，这就排除了所有可变对象。) 重复键之间的冲突不会被检测；指定键所保存的最后一个值（即在显示中排最右边的文本）将为最终的值。

在 3.8 版本发生变更: 在 Python 3.8 之前的字典推导式中，并没有定义好键和值的求值顺序。在 CPython 中，值会先于键被求值。根据 **PEP 572** 的提议，从 3.8 开始，键会先于值被求值。

6.2.8 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。

在生成器表达式中使用的变量会在为生成器对象调用 *__next__()* 方法的时候以惰性方式被求值（即与普通生成器相同的方式）。但是，最左侧 `for` 子句内的可迭代对象是会被立即求值的，因此它所造成的错误会在生成器表达式被定义时被检测到，而不是在获取第一个值时才出错。后续的 `for` 子句以及最左侧 `for` 子句内的任何筛选条件无法在外层作用域内被求值，因为它们可能会依赖于从最左侧可迭代对象获取的值。例如: `(x*y for x in range(10) for y in range(x, x+10))`.

圆括号在只附带一个参数的调用中可以被省略。详情参见*调用* 一节。

为了避免干扰到生成器表达式本身的预期操作，禁止在隐式定义的生成器中使用 `yield` 和 `yield from` 表达式。

如果生成器表达式包含 `async for` 子句或*await* 表达式，则称为 异步生成器表达式。异步生成器表达式会返回一个新的异步生成器对象，此对象属于异步迭代器 (参见异步迭代器)。

Added in version 3.6: 引入了异步生成器表达式。

在 3.7 版本发生变更: 在 Python 3.7 之前，异步生成器表达式只能在*async def* 协和中出现。从 3.7 开始，任何函数都可以使用异步生成器表达式。

在 3.8 版本发生变更: `yield` 和 `yield from` 在隐式嵌套的作用域中已被禁用。

6.2.9 yield 表达式

```
yield_atom       ::=  "(" yield_expression ")"
yield_from       ::=  "yield" "from" expression
yield_expression ::=  "yield" yield_list | yield_from
```

yield 表达式在定义*generator* 函数或*asynchronous generator* 函数时才会用到因此只能在函数定义的内部使用。在一个函数体内使用 yield 表达式会使这个函数变成一个生成器函数，而在一个*async def* 函数的内部使用它则会让这个协程函数变成一个异步生成器函数。例如:

```
def gen():  # 定义一个生成器函数
    yield 123

async def agen(): # 定义一个异步生成器函数
    yield 123
```

由于它们会对外层作用域造成附带影响，`yield` 表达式不被允许作为用于实现推导式和生成器表达式的隐式定义作用域的一部分。

在 3.8 版本发生变更: 禁止在实现推导式和生成器表达式的隐式嵌套作用域中使用 yield 表达式。

下面是对生成器函数的描述，异步生成器函数会在异步生成器函数 一节中单独介绍。

当一个生成器函数被调用时，它将返回一个名为生成器的迭代器。然后这个生成器将控制生成器函数的执行。执行过程会在这个生成器的某个方法被调用时开始。这时，函数会执行到第一个 yield 表达式，在那里它将再次被挂起，向生成器的调用方返回 *yield_list* 的值，或者如果 *yield_list* 被省略则返回 `None`。所谓的挂起，就是说所有局部状态都会被保留，包括局部变量的当前绑定、指令指针、内部求值栈及任何异常处理等等。当通过调用生成器的某个方法恢复执行时，这个函数的运行就与 yield 表达式只是一个外部调用的情况完全一样。在恢复执行后 yield 表达式的值取决于恢复执行所调用的方法。如果是用*__next__()* (一般是通过*for* 或者 `next()` 内置函数) 则结果为 `None`。在其他情况下，如果是用*send()*，则结果将为传给该方法的值。

所有这些使生成器函数与协程非常相似；它们 yield 多次，它们具有多个入口点，并且它们的执行可以被挂起。唯一的区别是生成器函数不能控制在它在 yield 后交给哪里继续执行；控制权总是转移到生成器的调用者。

在*try* 结构中的任何位置都允许 yield 表达式。如果生成器在 (因为引用计数到零或是因为被垃圾回收) 销毁之前没有恢复执行，将调用生成器-迭代器的*close()* 方法. close 方法允许任何挂起的*finally* 子句执行。

当使用 `yield from <expr>` 时，所提供的表达式必须是一个可迭代对象。迭代该可迭代对象所产生的值会被直接传递给当前生成器方法的调用者。任何通过*send()* 传入的值以及任何通过*throw()* 传入的异常如果有适当的方法则会被传给下层迭代器。如果不是这种情况，那么*send()* 将引发 `AttributeError` 或 `TypeError`，而*throw()* 将立即引发所转入的异常。

当下层迭代器完成时，被引发的 `StopIteration` 实例的 `value` 属性会成为 yield 表达式的值。它可以在引发 `StopIteration` 时被显式地设置，也可以在子迭代器是一个生成器时自动地设置（通过从子生成器返回一个值）。

在 3.3 版本发生变更: 添加 `yield from <expr>` 以委托控制流给一个子迭代器。

当 yield 表达式是赋值语句右侧的唯一表达式时，括号可以省略。

参见

PEP 255 - 简单生成器
在 Python 中加入生成器和*yield* 语句的提议。

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。

PEP 380 - 委托给子生成器的语法
引入 *yield_from* 语法的提议，以方便地委托给子生成器。

PEP 525 - 异步生成器
通过给协程函数加入生成器功能对 **PEP 492** 进行扩展的提议。

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__()`

开始一个生成器函数的执行或是从上次执行 yield 表达式的位置恢复执行。当一个生成器函数通过*__next__()* 方法恢复执行时，当前的 yield 表达式总是取值为 `None`。随后会继续执行到下一个 yield 表达式，这时生成器将再次挂起，而 *yield_list* 的值会被返回给*__next__()* 的调用方。如果生成器没有产生下一个值就退出，则将引发 `StopIteration` 异常。

此方法通常是隐式地调用，例如通过*for* 循环或是内置的 `next()` 函数。

`generator.send(`*value*`)`

恢复执行并向生成器函数“发送”一个值。*value* 参数将成为当前 yield 表达式的结果。*send()* 方法会返回生成器所产生的下一个值，或者如果生成器没有产生下一个值就退出则会引发 `StopIteration`。当调用*send()* 来启动生成器时，它必须以 `None` 作为调用参数，因为这时没有可以接收值的 yield 表达式。

`generator.throw(`*value*`)`

`generator.throw(`*type*`[,` *value*`[,` *traceback*`]])`

在生成器暂停的位置引发一个异常，并返回该生成器函数所产生的下一个值。如果生成器没有产生下一个值就退出，则将引发 `StopIteration` 异常。如果生成器函数没有捕获传入的异常，或是引发了另一个异常，则该异常会被传播给调用方。

在典型的使用场景下，其调用将附带单个异常实例，类似于使用*raise* 关键字的方式。

但是为了向下兼容，也支持第二种签名方式，遵循来自旧版本 Python 的惯例。*type* 参数应为一个异常类，而 *value* 应为一个异常实例。如果未提供 *value*，则将调用 *type* 构造器来获取一个实例。如果提供了 *traceback*，它将被设置到异常上，否则任何存储在 *value* 中的现有 `__traceback__` 属性都会被清空。

在 3.12 版本发生变更: 第二个签名 (type[, value[, traceback]]) 已被弃用并可能在未来的 Python 版本中移除。

`generator.close()`

在生成器函数暂停的位置引发 `GeneratorExit`。如果生成器函数捕获该异常并返回一个值，这个值将从*close()* 返回。如果生成器函数已经关闭，或者引发了 `GeneratorExit` (由于未捕获异常)，*close()* 将返回 `None`。如果生成器产生了一个值，则将引发 `RuntimeError`。如果生成器引发了任何其他异常，它将被传播给调用方。如果生成器已经由于异常或以正常退出方式结束执行，*close()* 将返回 `None` 并且不会造成其他影响。

在 3.13 版本发生变更: 如果生成器在被关闭时返回了一个值，这个值将从*close()* 返回。

例子

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

对于 `yield from` 的例子, 参见 “Python 有什么新变化” 中的 pep-380。

异步生成器函数

在一个使用*async def* 定义的函数或方法中出现的 yield 表达式会进一步将该函数定义为一个*asynchronous generator* 函数。

当一个异步生成器函数被调用时，它会返回一个名为异步生成器对象的异步迭代器。此对象将在之后控制该生成器函数的执行。异步生成器对象通常被用在协程函数的*async for* 语句中，类似于在*for* 语句中使用生成器对象。

调用某个异步生成器的方法将返回一个*awaitable* 对象，执行会在此对象被等待时启动。到那时，将执行至第一个 yield 表达式，在那里它会再次挂起，将 *yield_list* 的值返回给等待中的协程。与生成器一样，挂起意味着所有局部状态会被保留，包括局部变量的当前绑定、指令指针、内部求值栈以及任何异常处理的状态。当执行在等待异步生成器的方法返回下一个对象后恢复时，该函数可以从原状态继续执行，就仿佛 yield 表达式只是另一个外部调用那样。恢复执行后 yield 表达式的值取决于恢复执行所用的方法。如果是使用*__anext__()* 则结果为 `None`。否则的话，如果是使用*asend()*，则结果将是传递给该方法的值。

如果一个异步生成器恰好因*break*、调用方任务被取消，或是其他异常而提前退出，生成器的异步清理代码将会运行并可能引发异常或访问意外上下文中的上下文变量 -- 也许是在它所依赖的任务的生命周期之后，或是在异步生成器垃圾回收钩子被调用时的事件循环关闭期间。为了防止这种情况，调用方必须通过调用*aclose()* 方法来显式地关闭异步生成器以终结生成器并最终从事件循环中将其分离。

在异步生成器函数中，yield 表达式允许出现在*try* 结构的任何位置。但是，如果一个异步生成器在其被终结（由于引用计数达到零或被作为垃圾回收）之前未被恢复，则 `try` 结构中的 yield 表达式可能导致挂起的*finally* 子句执行失败。在此情况下，应由运行该异步生成器的事件循环或任务调度器来负责调用异步生成器-迭代器的*aclose()* 方法并运行所返回的协程对象，从而允许任何挂起的 `finally` 子句得以执行。

为了能在事件循环终结时执行最终化处理，事件循环应当定义一个 *终结器*函数，它接受一个异步生成器迭代器并将调用*aclose()* 且执行该协程。这个 *终结器*可以通过调用 `sys.set_asyncgen_hooks()` 来注册。当首次迭代时，异步生成器迭代器将保存已注册的 *终结器*以便在最终化时调用。有关 *终结器*方法的参考示例请查看在 Lib/asyncio/base_events.py 的中的 `asyncio.Loop.shutdown_asyncgens` 实现。

`yield from <expr>` 表达式如果在异步生成器函数中使用会引发语法错误。

异步生成器-迭代器方法

这个子小节描述了异步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。

coroutine agen.**__anext__**()

返回一个可等待对象，它在运行时会开始执行该异步生成器或是从上次执行的 yield 表达式位置恢复执行。当一个异步生成器通过*__anext__()* 方法恢复执行时，当前的 yield 表达或所返回的可等待对象总是取值为 None，它在运行时将继续执行到下一个 yield 表达式。该 yield 表达式的 *yield_list* 的值会是完成的协程所引发的 StopIteration 异步的值。如果异步生成器没有产生下一个值就退出，则该可等待对象将引发 StopAsyncIteration 异常，提示该异步迭代操作已完成。

此方法通常是通过*async for* 循环隐式地调用。

coroutine agen.**asend**(*value*)

返回一个可等待对象，它在运行时会恢复该异步生成器的执行。与生成器的*send()* 方法一样，此方法会“发送”一个值给异步生成器函数，其 *value* 参数会成为当前 yield 表达式的结果值。*asend()* 方法所返回的可等待对象会将所引发的 StopIteration 作为生成器产生的下一个值返回，或者如果异步生成器没有产生下一个值就退出则引发 StopAsyncIteration。当调用*asend()* 来启动异步生成器时，它必须以 None 作为参数被调用，因为这时没有可以接收值的 yield 表达式。

coroutine agen.**athrow**(*value*)

coroutine agen.**athrow**(*type*[, *value*[, *traceback*]])

返回一个可等待对象，它会在异步生成器暂停的位置引发 type 类型的异常，并返回该生成器函数所产生的下一个值，其值为所引发的 StopIteration 异常。如果异步生成器没有产生下一个值就退出，则将由该可等待对象引发 StopAsyncIteration 异步。如果生成器函数没有捕获传入的异常，或引发了另一个异常，则当可等待对象运行时该异常会被传播给可等待对象的调用者。

在 3.12 版本发生变更: 第二个签名 (type[, value[, traceback]]) 已被弃用并可能在未来的 Python 版本中移除。

coroutine agen.**aclose**()

返回一个可等待对象，它会在运行时向异步生成器函数暂停的位置抛入一个 GeneratorExit。如果该异步生成器函数正常退出、关闭或引发 GeneratorExit (由于未捕获该异常) 则返回的可等待对象将引发 StopIteration 异常。后续调用异步生成器所返回的任何其他可等待对象将引发 StopAsyncIteration 异常。如果异步生成器产生了一个值，该可等待对象会引发 RuntimeError。如果异步生成器引发任何其他异常，它会被传播给可等待对象的调用者。如果异步生成器已经由于异常或正常退出则后续调用*aclose()* 将返回一个不会做任何事的可等待对象。

6.3 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
```

6.3.1 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
```

此原型必须求值为一个支持属性引用的类型的对象，多数对象都支持此特性。随后该对象会被要求产生以指定标识符为名称的属性。所产生对象的类型和值会根据该对象来确定。对同一属性引用的多次求值可能产生不同的对象。

产生过程可通过重载*__getattribute__()* 方法或*__getattr__()* 方法来自定义。将会先调用 __getattribute__() 方法并返回一个值或者如果属性不可用则会引发 AttributeError。

如果引发了 AttributeError 并且对象具有 __getattr__() 方法，则将调用该方法作为回退项。

6.3.2 抽取

对一个容器类 的实例执行抽取操作通常将会从该容器中选取一个元素。而对一个泛型类 执行抽取操作通常将会返回一个 GenericAlias 对象。

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

当一个对象被抽取时，解释器将对原型和表达式列表进行求值。

原型必须可被求值为一个支持抽取操作的对象。一个对象可通过同时定义 __getitem__() 和 __class_getitem__() 或其中之一来支持抽取操作。当原型被抽取时，表达式列表的求值结果将被传给以上方法中的一个。对于在何时会调用 __class_getitem__ 而不是 __getitem__ 的更多细节，请参阅 __class_getitem__ 与 __getitem__。

如果表达式列表包含至少一个逗号，或者如果某个表达式带有星号，该表达式列表将求值为包含该表达式列表中所有条目的 tuple。在其他情况下，表达式列表将被求值为列表中唯一成员的值。

在 3.11 版本发生变更: 一个表达式列表中的表达式可以带星号。参见 **PEP 646**。

对于内置对象，有两种类型的对象支持通过 __getitem__() 执行抽取操作:

1. 映射。如果原型是一个*mapping*，则表达式列表必须求值为一个以该映射的某个键为值的对象，而抽取操作会在映射中选取该键所对应的值。内置映射类的一个例子是 dict 类。
2. 序列。如果原型是一个*sequence*，则表达式列表必须求值为一个 int 或一个 slice (如下面的小节所讨论的)。内置序列类的例子包括 str, list 和 tuple 等类。

正式语法规则并未设置针对序列 中负索引号的特殊保留条款。不过，内置序列都提供了通过给索引号加上序列长度来解读负索引号的 __getitem__() 方法，因此举例来说，x[-1] 将选取 x 的最后一项。结果值必须为一个小于序列中条目数的非负整数，抽取操作会选取索引号为该值的条目（从零开始计数）。由于对负索引号和切片的支持是在 __getitem__() 方法中实现的，因而重写此方法的子类将需要显式地添加这种支持。

字符串是一种特殊的序列，其中的项是 字符。字符并不是一种单独的数据类型而是长度恰好为一个字符的字符串。

6.3.3 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      ::=  primary "[" slice_list "]"
slice_list   ::=  slice_item ("," slice_item)* [","]
slice_item   ::=  expression | proper_slice
proper_slice ::=  [lower_bound] ":" [upper_bound] [ ":" [stride] ]
lower_bound  ::=  expression
upper_bound  ::=  expression
stride       ::=  expression
```

此处的正式句法中存在一点歧义：任何形似表达式列表的东西同样也会形似切片列表，因此任何抽取操作也可以被解析为切片。为了不使句法更加复杂，于是通过定义将此情况解析为抽取优先于解析为切片来消除这种歧义（切片列表未包含正确的切片就属于此情况）。

切片的语义如下所述。原型通过一个根据所下所示的切片列表来构造的键进行索引（与普通的抽取一样使用 __getitem__() 方法）。如果切片列表包含至少一个逗号，则键将是一个包含切片项转换形式的元组；否则的话，键将是单个切片项的转换形式。切片项如为一个表达式，则其转换形式就是该表达式。一个正确的切片的转换形式就是一个切片对象（参见标准类型层级结构 一节），该对象的 start, stop 和 step 属性将分别为表达式所给出的下界、上界和步长值，省略的表达式将用 None 来替换。

6.3.4 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 [","] | comprehension] ")"
argument_list        ::=  positional_arguments ["," starred_and_keywords]
                          ["," keywords_arguments]
                          | starred_and_keywords ["," keywords_arguments]
                          | keywords_arguments
positional_arguments ::=  positional_item ("," positional_item)*
positional_item      ::=  assignment_expression | "*" expression
starred_and_keywords ::=  ("*" expression | keyword_item)
                          ("," "*" expression | "," keyword_item)*
keywords_arguments   ::=  (keyword_item | "**" expression)
                          ("," keyword_item | "," "**" expression)*
keyword_item         ::=  identifier "=" expression
```

一个可选项为在位置和关键字参数后加上逗号而不影响语义。

此原型必须被求值为一个可调用对象（用户自定义函数、内置函数、内置对象的方法、类对象、类实例的方法以及任何具有`__call__()`方法的对象都是可调用对象）。所有参数表达式将在尝试调用前被求值）。请参阅函数定义 一节了解正式的*parameter* 列表的语法。

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例表。如果有 N 个位置参数，则它们会被放入前 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`，它也会填充空位）。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。（函数一旦被定义，其默认值就会被计算；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；这种情况通常应当被避免。）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。在其他情况下，已填充空位的列表会被作为调用的参数列表。

CPython 实现细节：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。

如果函数调用中出现了 `*expression` 句法，`expression` 必须求值为一个*iterable*。来自该可迭代对象的元素会被当作是额外的位置参数。对于 `f(x1, x2, *y, x3, x4)` 调用，如果 *y* 求值为一个序列 *y1*, ..., *yM*，则它就等价于一个带有 M+4 个位置参数 *x1*, *x2*, *y1*, ..., *yM*, *x3*, *x4* 的调用。

这样做的一个后果是虽然 `*expression` 句法可能出现于显式的关键字参数 *之后*，但它会在关键字参数（以及任何 `**expression` 参数 -- 见下文）*之前*被处理。因此:

```
>>> 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 语句并不常见，因此实际上这样的混淆不会发生。

如果函数调用中出现了 **expression，则 expression 必须求值为一个*mapping*，其内容会被当作是额外的关键字参数。如果一个形参与一个已给定值关键字相匹配（通过显式的关键字参数，或通过另一个解包），则会引发 TypeError 异常。

当使用 **expression 时，该映射中的每个键都必须为字符串。该映射中的每个值将被赋值给名称与键相同的适用于关键字赋值的第一个正式形参。键名不需要是 Python 标识符（例如 "max-temp °F" 也是可接受的，但它将不能与可被声明的任何正式形参相匹配）。如果键值对未与某个正式形参相匹配则将被 ** 形参所收集，或者如果没有此形参，则会引发 TypeError 异常。

使用 *identifier 或 **identifier 句法的正式参数不能被用作位置参数空位或关键字参数名称。

在 3.5 版本发生变更: 函数调用接受任意数量的 * 和 ** 拆包，位置参数可能跟在可迭代对象拆包 (*) 之后，而关键字参数可能跟在字典拆包 (**) 之后。由 **PEP 448** 发起最初提议。

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:
该类定义定义*__call__()* 方法；其效果将等价于调用该方法。

6.4 await 表达式

挂起*coroutine* 的执行以等待一个*awaitable* 对象。只能在*coroutine function* 内部使用。

```
await_expr   ::=   "await" primary
```

Added in version 3.5.

6.5 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   ::=   (await_expr | 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。

幂运算符与附带两个参数调用内置 pow() 函数具有相同的语义：结果为对其左参数进行其右参数所指定幂次的乘方运算。数值参数会先转换为相同类型，结果也为转换后的类型。

对于 int 类型的操作数，结果将具有与操作数相同的类型，除非第二个参数为负数；在那种情况下，所有参数会被转换为 float 类型并输出 float 类型的结果。例如，10**2 返回 100，而 10**-2 返回 0.01。

对 0.0 进行负数幂次运算将导致 ZeroDivisionError。对负数进行分数幂次运算将返回 complex 数值。(在早期版本中这将引发 ValueError。)

此运算可使用特殊的__pow__() 和__rpow__() 方法来自定义。

6.6 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
```

单目运算符 - (负值) 将输出对数字参数的负值；该运算可通过__neg__() 特殊方法来重写。

单目运算符 + (正值) 将不加修改地输出其数字参数；该运算可通过__pos__() 特殊方法来重写。

单目运算符 ~ (取反) 将输出对其整数参数按位取反的结果。对 x 按位取反被定义为 -(x+1)。它只作用于整数或是重写了__invert__() 特殊方法的自定义对象。

在所有三种情况下，如果参数的类型不正确，将引发 TypeError 异常。

6.7 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 "@" m_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
```

运算符 * (乘) 将输出其参数的乘积。两个参数或者必须都为数字，或者一个参数必须为整数而另一个参数必须为序列。在前一种情况下，两个数字将被转换为相同类型然后相乘。在后一种情况下，将执行序列的重复；重复因子为负数将输出空序列。

此运算可使用特殊的__mul__() 和__rmul__() 方法来自定义。

运算符 @ (at) 的目标是用于矩阵乘法。没有内置 Python 类型实现此运算符。

此运算可使用特殊的__matmul__() 和__rmatmul__() 方法来自定义。

Added in version 3.5.

运算符 / (除) 和 // (整除) 将输出其参数的商。两个数字参数将先被转换为相同类型。整数相除会输出一个 float 值，整数相整除的结果仍是整数；整除的结果就是使用'floor' 函数进行算术除法的结果。除以零的运算将引发 ZeroDivisionError 异常。

除法运算可使用特殊的__truediv__() 和__rtruediv__() 方法来自定义。向下整除运算可使用特殊的__floordiv__() 和__rfloordiv__() 方法来自定义。

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[1]。

[1] 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。

整除与模运算符的联系可通过以下等式说明: `x == (x//y)*y + (x%y)`。此外整除与模也可通过内置函数 `divmod()` 来同时进行: `divmod(x, y) == (x//y, x%y)`。[2]。

除了对数字执行模运算，运算符 `%` 还被字符串对象重载用于执行旧式的字符串格式化（又称插值）。字符串格式化句法的描述参见 Python 库参考的 old-string-formatting 一节。

modulo 运算可使用特殊的*__mod__()* 和*__rmod__()* 方法来自定义。

整除运算符，模运算符和 `divmod()` 函数未被定义用于复数。如果有必要可以使用 `abs()` 函数将其转换为浮点数。

运算符 + (addition) 将输出其参数的和。两个参数或者必须都为数字，或者都为相同类型的序列。在前一种情况下，两个数字将被转换为相同类型然后相加。在后一种情况下，将执行序列拼接操作。

此运算可使用特殊的*__add__()* 和*__radd__()* 方法来自定义。

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。

此运算可使用特殊的*__sub__()* 和*__rsub__()* 方法来自定义。

6.8 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
```

这些运算符接受整数参数。它们会将第一个参数左移或右移第二个参数所指定的比特位数。

左移位运算可使用特殊的*__lshift__()* 和*__rlshift__()* 方法来自定义。右移位运算可使用特殊的*__rshift__()* 和*__rrshift__()* 方法来自定义。

右移 *n* 位被定义为被 `pow(2,n)` 整除。左移 *n* 位被定义为乘以 `pow(2,n)`。

6.9 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
```

`&` 运算符将输出对其参数按位 AND 的结果，参数必须都为整数或者其中之一必须为重写了*__and__()* 或*__rand__()* 特殊方法的自定义对象。

`^` 运算符将输出对其参数按位 XOR (异或) 的结果，参数必须都为整数或者其中之一必须为重写了*__xor__()* 或*__rxor__()* 特殊方法的自定义对象。

`|` 运算符将输出对其参数按位 OR (非异或) 的结果，参数必须都为整数或者其中之一为重写了*__or__()* 或*__ror__()* 特殊方法的自定义对象。

6.10 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:

[2] 如果 x 恰好非常接近于 y 的整数倍，则由于舍入的存在 `x//y` 可能会比 `(x-x%y)//y` 大。在这种情况下，Python 会返回后一个结果，以便保持令 `divmod(x,y)[0] * y + x % y` 尽量接近 `x`.

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

比较运算会产生布尔值: `True` 或 `False`。自定义的 *富比较方法* 可能返回非布尔值。在此情况下 Python 将在布尔运算上下文中对该值调用 `bool()`。

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`，不同点在于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）。

6.10.1 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。

由于所有类型都是 `object` 的（直接或间接）的子类型，因此它们都从 `object` 继承了默认的比较行为。类型可以通过实现 *rich comparison methods* 如 *`__lt__()`* 来自定义它们的比较行为，详情参见基本定制。

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`)。

次序比较 (<, >, <= 和 >=) 默认没有提供；如果尝试比较会引发 `TypeError`。规定这种默认行为的原因是缺少与一致性比较类似的固定值。

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。

  非数字值 `float('NaN')` 和 `decimal.Decimal('NaN')` 属于特例。任何数字与非数字值的排序比较均返回假值。还有一个反直觉的结果是非数字值不等于其自身。举例来说，如果 `x = float('NaN')` 则 `3 < x`, `x < 3` 和 `x == x` 均为假值，而 `x != x` 则为真值。此行为是遵循 IEEE 754 标准的。

- `None` 和 `NotImplemented` 都是单例对象。**PEP 8** 建议单例对象的比较应当总是通过 `is` 或 `is not` 来进行，绝不要使用等于运算符。

- 二进制码序列 (`bytes` 或 `bytearray` 的实例) 可进行类型内部和跨类型的比较。它们使用其元素的数字值按字典顺序进行比较。

- 字符串 (`str` 的实例) 使用其字符的 Unicode 码位数字值 (内置函数 `ord()` 的结果) 按字典顺序进行比较。[3]

  字符串和二进制码序列不能直接比较。

[3] 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。

对于字符串，比较运算符会按 Unicode 码位级别进行比较。这可能会违反人类的直觉。例如，`"\u00C7" == "\u0043\u0327"` 为 `False`，虽然两个字符串都代表同一个抽象字符" 带有下加符的大写拉丁字母 C"。

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()`。

- 序列 (tuple, list 或 range 的实例) 只可进行类型内部的比较，range 还有一个限制是不支持次序比较。以上对象的跨类型一致性比较结果将是不相等，跨类型次序比较将引发 TypeError。

  序列比较是按字典序对相应元素进行逐个比较。内置容器通常设定同一对象与其自身是相等的。这使得它们能跳过同一对象的相等性检测以提升运行效率并保持它们的内部不变性。

  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）。
  - 对于支持次序比较的多项集，排序与其第一个不相等元素的排序相同（例如 [1,2,x] <= [1,2,y] 的值与 x <= y 相同）。如果对应元素不存在，较短的多项集排序在前（例如 [1,2] < [1,2,3] 为真值）。

- 两个映射 (dict 的实例) 若要相等则必须当且仅当它们具有相等的 (key, value) 对。键和值的相相等性比较强制要求自反射性。

  次序比较 (<, >, <= 和 >=) 将引发 TypeError。

- 集合 (set 或 frozenset 的实例) 可进行类型内部和跨类型的比较。

  它们将比较运算符定义为子集和超集检测。这类关系没有定义完全排序（例如 {1,2} 和 {2,3} 两个集合不相等，即不为彼此的子集，也不为彼此的超集。相应地，集合不适宜作为依赖于完全排序的函数的参数（例如如果给出一个集合列表作为 min(), max() 和 sorted() 的输入将产生未定义的结果）。

  集合的比较强制规定其元素的自反射性。

- 大多数其他内置类型没有实现比较方法，因此它们会继承默认的比较行为。

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 并不强制要求这些一致性规则。实际上，非数字值就是一个不遵循这些规则的例子。

6.10.2 成员检测运算

运算符*in* 和*not in* 用于成员检测。如果 *x* 是 *s* 的成员则 `x in s` 求值为 `True`，否则为 `False`。`x not in s` 返回 `x in s` 取反后的值。所有内置序列和集合类型以及字典都支持此运算，对于字典来说 `in` 检测其是否有给定的键。对于 list, tuple, set, frozenset, dict 或 collections.deque 这样的容器类型，表达式 `x in y` 等价于 `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 For user-defined classes which define the *__contains__()* 方法来用户自定义类来说，如果 `y.__contains__(x)` 返回真值则 `x in y` 将返回 `True`，否则返回 `False`。

对于未定义*__contains__()* 但定义了*__iter__()* 的用户自定义类来说，如果在迭代 `y` 期间产生了值 `z` 使得表达式 `x is z or x == z` 为真值，则 `x in y` 将为 `True`。如果在迭代期间引发了异常，则将等同于*in* 引发了该异常。

最后，将会尝试旧式的迭代协议：如果一个类定义了*__getitem__()*，则当且仅当存在非负整数索引号 *i* 使得 `x is y[i] or x == y[i]` 并且没有更小的索引号引发 `IndexError` 异常时 `x in y` 才为 `True`。(如果引发了任何其他异常，则等同于*in* 引发了该异常。)

运算符*not in* 被定义为具有与*in* 相反的逻辑值。

6.10.3 标识号比较

运算符*is* 和*is not* 用于检测对象的标识号：当且仅当 *x* 和 *y* 是同一对象时 `x is y` 为真。一个对象的标识号可使用 `id()` 函数来确定。`x is not y` 会产生相反的逻辑值。[4]

6.11 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
```

在执行布尔运算的情况下，或是当表达式被用于流程控制语句时，以下值会被解读为假值: `False`, `None`, 所有类型的数字零，以及空字符串和空容器（包括字符串、元组、列表、字典、集合与冻结集合）。所有其他值都会被解读为真值。用户自定义对象可通过提供*__bool__()* 方法来定制其逻辑值。

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。

请注意*and* 和*or* 都不限制其返回的值和类型必须为 `False` 和 `True`，而是返回最后被求值的操作数。此行为是有必要的，例如假设 `s` 为一个当其为空时应被替换为某个默认值的字符串，表达式 `s or 'foo'` 将产生希望的值。由于*not* 必须创建一个新值，不论其参数为何种类型它都会返回一个布尔值（例如，`not 'foo'` 结果为 `False` 而非 `''`。)

6.12 赋值表达式

```
assignment_expression  ::=  [identifier ":="] expression
```

赋值表达式(有时又被称为“命名表达式”或“海象表达式”)将一个 *expression* 赋值给一个 *identifier*，同时还会返回 *expression* 的值。

一个常见用例是在处理匹配的正则表达式的时候:

[4] 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。

```
if matching := pattern.search(data):
    do_something(matching)
```

或者是在处理分块的文件流的时候:

```
while chunk := file.read(9000):
    process(chunk)
```

赋值表达式在被用作表达式语句及在被用作切片、条件表达式、lambda 表达式、关键字参数和推导式中的 if 表达式以及在 `assert`, `with` 和 `assignment` 语句中的子表达式时必须用圆括号括起来。在其可使用的其他场合，圆括号则不是必须的，包括在 `if` 和 `while` 语句中。

Added in version 3.8: 请参阅 **PEP 572** 了解有关赋值表达式的详情。

6.13 条件表达式

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

条件表达式（有时称为“三目运算符”）在所有 Python 运算中具有最低的优先级。

表达式 `x if C else y` 首先是对条件 *C* 而非 *x* 求值。如果 *C* 为真，*x* 将被求值并返回其值；否则将对 *y* 求值并返回其值。

请参阅 **PEP 308** 了解有关条件表达式的详情。

6.14 lambda 表达式

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

lambda 表达式（有时称为 lambda 构型）被用于创建匿名函数。表达式 `lambda parameters: expression` 会产生一个函数对象。该未命名对象的行为类似于用以下方式定义的函数:

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

请参阅*函数定义* 了解有关参数列表的句法。请注意通过 lambda 表达式创建的函数不能包含语句或标注。

6.15 表达式列表

```
starred_expression         ::=  ["*"] or_expr
flexible_expression        ::=  assignment_expression | starred_expression
flexible_expression_list   ::=  flexible_expression ("," flexible_expression)* [","]
starred_expression_list    ::=  starred_expression ("," starred_expression)* [","]
expression_list            ::=  expression ("," expression)* [","]
yield_list                 ::=  expression_list | starred_expression "," [starred_expression_list
```

除了作为列表或集合显示的一部分，包含至少一个逗号的表达式列表将生成一个元组。元组的长度就是列表中表达式的数量。表达式将从左至右被求值。

一个星号 `*` 表示 *可迭代拆包*。其操作数必须为一个*iterable*。该可迭代对象将被拆解为迭代项的序列，并被包含于在拆包位置上新建的元组、列表或集合之中。

Added in version 3.5: 表达式列表中的可迭代对象拆包，最初由 **PEP 448** 提出。

Added in version 3.11: 一个表达式列表中的任何条目都可以带星号。参见 **PEP 646**。

末尾的逗号仅在创建单条目元组，比如 `1,` 时才是必需的；在所有其他情况下它都是可选项。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: `()`。)

6.16 求值顺序

Python 按从左至右的顺序对表达式求值。但注意在对赋值操作求值时，右侧会先于左侧被求值。

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
```

6.17 运算符优先级

下表对 Python 中运算符的优先顺序进行了总结，从最高优先级（最先绑定）到最低优先级（最后绑定）。相同单元格内的运算符具有相同优先级。除非语法显式地指明，否则运算符均为双目运算符。相同单元格内的运算符从左至右组合的（只有幂运算符是从右至左组合的）。

请注意比较、成员检测和标识号检测均为相同优先级，并具有如比较运算 一节所描述的从左至右串连特性。

运算符	描述
`(expressions...)`, `[expressions...]`, `{key: value...}`, `{expressions...}`	绑定或加圆括号的表达式，列表显示，字典显示，集合显示
`x[index]`, `x[index:index]`, `x(arguments...)`, `x.attribute`	抽取，切片，调用，属性引用
await x	await 表达式
`**`	乘方[5]
`+x`, `-x`, `~x`	正，负，按位非 NOT
`*`, `@`, `/`, `//`, `%`	乘，矩阵乘，除，整除，取余[6]
`+`, `-`	加和减
`<<`, `>>`	移位
`&`	按位与 AND
`^`	按位异或 XOR
`|`	按位或 OR
in, *not in*, *is*, *is not*, `<`, `<=`, `>`, `>=`, `!=`, `==`	比较运算，包括成员检测和标识号检测
not x	布尔逻辑非 NOT
and	布尔逻辑与 AND
or	布尔逻辑或 OR
if -- *else*	条件表达式
lambda	lambda 表达式
`:=`	赋值表达式

[5] 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`。
[6] `%` 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。

备注

CHAPTER 7

简单语句

简单语句由一个单独的逻辑行构成。多条简单语句可以存在于同一行内并以分号分隔。简单语句的句法为:

```
simple_stmt   ::=   expression_stmt
                  | assert_stmt
                  | assignment_stmt
                  | augmented_assignment_stmt
                  | annotated_assignment_stmt
                  | pass_stmt
                  | del_stmt
                  | return_stmt
                  | yield_stmt
                  | raise_stmt
                  | break_stmt
                  | continue_stmt
                  | import_stmt
                  | future_stmt
                  | global_stmt
                  | nonlocal_stmt
                  | type_stmt
```

7.1 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   ::=   starred_expression
```

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。

在交互模式下，如果结果值不为 `None`，它会通过内置的 `repr()` 函数转换为一个字符串，该结果字符串将以单独一行的形式写入标准输出（例外情况是如果结果为 `None`，则该过程调用不产生任何输出。）

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

(请参阅原型 一节了解 属性引用, 抽取和 切片的句法定义。)

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）。

对象赋值的目标对象可以包含于圆括号或方括号内，具体操作按以下方式递归地定义。

- 如果目标列表为后面不带逗号、可以包含于圆括号内的单一目标，则将对象赋值给该目标。
- 否则:
  - 如果目标列表包含一个带有星号前缀的目标，这称为“加星”目标：则该对象至少必须为与目标列表项数减一相同项数的可迭代对象。该可迭代对象前面的项将按从左至右的顺序被赋值给加星目标之前的目标。该可迭代对象末尾的项将被赋值给加星目标之后的目标。然后该可迭代对象中剩余项的列表将被赋值给加星目标（该列表可以为空）。
  - 否则：该对象必须为具有与目标列表相同项数的可迭代对象，这些项将按从左至右的顺序被赋值给对应的目标。

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）：
  - 如果该名称未出现于当前代码块的*global* 或*nonlocal* 语句中：该名称将被绑定到当前局部命名空间的对象。
  - 否则：该名称将被分别绑定到全局命名空间或由*nonlocal* 所确定的外层命名空间的对象。

  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）。
- 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，则左侧会创建一个新的实例属性作为赋值的目标:

  ```
  class Cls:
      x = 3               # 类变量
  inst = Cls()
  inst.x = inst.x + 1   # 将 inst.x 改为 4 而 Cls.x 仍为 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。

- 如果目标为一个抽取项：引用中的原型表达式会被求值。它应当产生一个可变序列对象（例如列表）或一个映射对象（例如字典）。接下来，该抽取表达式会被求值。

  如果原型为一个可变序列对象（例如列表），抽取应产生一个整数。如其为负值，则再加上序列长度。结果值必须为一个小于序列长度的非负整数，序列将把被赋值对象赋值给该整数指定索引号的项。如果索引超出范围，将会引发 `IndexError`（给被抽取序列赋值不能向列表添加新项）。

  如果原型为一个映射对象（例如字典），下标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下标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）。

  对于用户自定义对象，会调用 *__setitem__()* 方法并附带适当的参数。

- 如果目标为一个切片：引用中的原型表达式会被求值。它应当产生一个可变序列对象（例如列表）。被赋值对象应当是一个相同类型的序列对象。接下来，下界与上界表达式如果存在的话将被求值；默认值分别为零和序列长度。上下边界值应当为整数。如果某一边界为负值，则会加上序列长度。求出的边界会被裁剪至介于零和序列长度的开区间中。最后，将要求序列对象以被赋值序列的项替换该切片。切片的长度可能与被赋值序列的长度不同，这会在目标序列允许的情况下改变目标序列的长度。

CPython 实现细节： 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。

虽然赋值的定义意味着左手边与右手边的重叠是“同时”进行的（例如 `a, b = b, a` 会交换两个变量的值），但在赋值给变量的多项集之内的重叠是从左至右进行的，这有时会令人混淆。例如，以下程序将会打印出 `[0, 2]`：

```
x = [0, 1]
i = 0
i, x[i] = 1, 2          # 先更新 i, 再更新 x[i]
print(x)
```

参见

PEP 3132 - 扩展的可迭代对象拆包
对 `*target` 特性的规范说明。

7.2.1 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                       ::=   "+=" | "-=" | "*=" | "@=" | "/=" | "//=" | "%=" | "**="
                                  | ">>=" | "<<=" | "&=" | "^=" | "|="
```

（请参阅原型 一节了解最后三种符号的句法定义。）

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。

增强赋值语句如 `x += 1` 可以被改写为 `x = x + 1` 以获得类似的、但并非完全等价的效果。在增强赋值版本中，`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，这意味着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。

不同于普通赋值，增强赋值会在对右手边求值之前对左手边求值。例如，`a[i] += f(x)` 首先查找 `a[i]`，然后对 `f(x)` 求值并执行加法操作，最后将结果写回到 `a[i]`。

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。

7.2.2 带标注的赋值语句

标注 赋值就是在单个语句中将变量或属性标注和可选的赋值语句合为一体:

```
annotated_assignment_stmt ::= augtarget ":" expression
                              ["=" (starred_expression | yield_expression)]
```

与普通赋值语句 的差别在于仅允许单个目标。

如果赋值目标由不带圆括号的单个名称组成则称为“简单”赋值目标。对于简单赋值目标，如果处在类或模块作用域中，标注将被求值并存储到一个特殊的类或模块属性 __annotations__ 中，该属性是一个将变量名称（如为私有则将移除）映射到被求值标注的字典。此属性为可写属性并且在类或模块体开始执行时自动创建，如果静态地发现标注的话。

如果赋值目标不是简单赋值目标（属性、下标节点或带圆括号的名称），则如果标注处在类或模块作用域中则会被求值，但不会被存储。

如果一个名称在函数作用域内被标注，则该名称为该作用域的局部变量。标注绝不会在函数作用域内被求值和保存。

如果存在右手边，带标注的赋值会在对标注求值之前（如果适用）执行实际的赋值。如果用作表达式目标的右手边不存在，则解释器会对目标求值，但最后的*__setitem__()* 或*__setattr__()* 调用除外。

参见

PEP 526 - 变量标注的语法
该提议增加了标注变量（也包括类变量和实例变量）类型的语法，而不再是通过注释来进行表达。

PEP 484 - 类型提示
该提议增加了 typing 模块以便为类型标注提供标准句法，可被静态分析工具和 IDE 所使用。

在 3.8 版本发生变更: 现在带有标注的赋值允许在右边以同样的表达式作为常规赋值。之前某些表达式（例如未加圆括号的元组表达式）会导致语法错误。

7.3 assert 语句

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)
```

以上等价形式假定 __debug__ 和 AssertionError 指向具有指定名称的内置变量。在当前实现中，内置变量 __debug__ 在正常情况下为 True，在请求优化时为 False (对应命令行选项为 -O)。如果在编译时请求优化，当前代码生成器不会为 assert 语句发出任何代码。请注意不必在错误信息中包含失败表达式的源代码；它会被作为栈追踪的一部分被显示。

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。

7.4 pass 语句

```
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    # 一个（目前）不做任何事的函数

class C: pass       # 一个（目前）没有任何方法的类
```

7.5 del 语句

```
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。

名称的删除将从局部或全局命名空间中移除该名称的绑定，具体作用域的确定是看该名称是否有在同一代码块的*global* 语句中出现。如果该名称未被绑定，将会引发 NameError。

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）。

在 3.2 版本发生变更: 在之前版本中，如果一个名称作为被嵌套代码块中的自由变量出现，则将其从局部命名空间中删除是非法的。

7.6 return 语句

```
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。

当*return* 将控制流传出一个带有*finally* 子句的*try* 语句时，该 finally 子句会先被执行然后再真正离开该函数。

在一个生成器函数中，*return* 语句表示生成器已完成并将导致 StopIteration 被引发。返回值（如果有的话）会被当作一个参数用来构建 StopIteration 并成为 StopIteration.value 属性。

在一个异步生成器函数中，一个空的*return* 语句表示异步生成器已完成并将导致 StopAsyncIteration 被引发。一个非空的 return 语句在异步生成器函数中会导致语法错误。

7.7 yield 语句

```
yield_stmt   ::=   yield_expression
```

yield 语句在语义上等同于*yield 表达式*。yield 语句可用来省略在使用等效的 yield 表达式语句时所必须的圆括号。例如，以下 yield 语句

```
yield <expr>
yield from <expr>
```

等同于以下 yield 表达式语句

```
(yield <expr>)
(yield from <expr>)
```

yield 表达式和语句仅在定义*generator* 函数时使用，并且仅被用于生成器函数的函数体内部。在函数定义中使用 yield 就足以使得该定义创建的是生成器函数而非普通函数。

有关*yield* 语义的完整细节请参看*yield 表达式* 一节。

7.8 raise 语句

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

如果没有提供表达式，则*raise* 会重新引发当前正在处理的异常，它也被称为 *活动的异常*。如果当前没有活动的异常，则会引发 RuntimeError 来提示发生了错误。

否则的话，*raise* 会将第一个表达式求值为异常对象。它必须为 BaseException 的子类或实例。如果它是一个类，当需要时会通过不带参数地实例化该类来获得异常的实例。

异常的 *类型* 为异常实例的类，*值* 为实例本身。

当有异常被引发时通常会自动创建一个回溯对象并将其关联到它的 __traceback__ 属性。你可以创建一个异常并使用 with_traceback() 异常方法直接设置你的回溯对象（该方法将返回同一异常实例，并将回溯对象设为其参数），就像这样：

```
raise Exception("foo occurred").with_traceback(tracebackobj)
```

from 子句用于异常串连：如果给出该子句，则第二个 *表达式* 必须为另一个异常类或实例。如果第二个表达式是一个异常实例，它将作为 __cause__ 属性（为一个可写属性）被关联到所引发的异常。如果该表达式是一个异常类，这个类将被实例化且所生成的异常实例将作为 __cause__ 属性被关联到所引发的异常。如果所引发的异常未被处理，则两个异常都将被打印：

```
>>> try:
...     print(1 / 0)
... except Exception as exc:
...     raise RuntimeError("Something bad happened") from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(1 / 0)
          ~~^~~
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("Something bad happened") from exc
RuntimeError: Something bad happened
```

当已经有一个异常在处理时如果有新的异常被引发则类似的机制会隐式地起作用。异常可以通过使用*except* 或*finally* 子句或者*with* 语句来处理。之前的异常将被关联至新异常的 __context__ 属性：

```
>>> try:
...     print(1 / 0)
... except:
```

(续下页)

(接上页)

```
...     raise RuntimeError("Something bad happened")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(1 / 0)
          ~~^~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("Something bad happened")
RuntimeError: Something bad happened
```

异常串连可通过在 `from` 子句中指定 `None` 来显式地加以抑制:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened") from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

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*try 语句* 一节查看。

在 3.3 版本发生变更: `None` 现在允许被用作 `raise X from Y` 中的 `Y`。

增加了 `__suppress_context__` 属性向来抑制异常上下文的自动显示。

在 3.11 版本发生变更: 如果活动异常的回溯在*except* 子句中被修改，则会有后续的 `raise` 语句重新引发该异常并附带被修改的回溯。在之前版本中，重新引发该异常则会附带它被捕获时的回溯。

7.9 break 语句

```
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。

它会终结最近的外层循环，如果循环有可选的 `else` 子句，也会跳过该子句。

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。

当*break* 将控制流传出一个带有*finally* 子句的*try* 语句时，该 `finally` 子句会先被执行然后再真正离开该循环。

7.10 continue 语句

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

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

当*continue* 将控制流传出一个带有*finally* 子句的*try* 语句时，该 `finally` 子句会先被执行然后再真正开始循环的下一个轮次。

7.11 import 语句

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

基本的 import 语句（不带*from*子句）会分两步执行:

1. 查找一个模块，如果有必要还会加载并初始化模块。
2. 在局部命名空间中为*import*语句发生位置所处的作用域定义一个或多个名称。

当语句包含多个子句（由逗号分隔）时这两个步骤将对每个子句分别执行，如同这些子句被分成独立的 import 语句一样。

第一个步骤，即查找和加载模块的细节在导入系统一节中有更详细的描述，其中也描述了可被导入的多种类型的包和模块，以及可用于定制导入系统的所有钩子对象。请注意如果这一步失败，则可能说明模块无法找到，或者是在初始化模块，包括执行模块代码期间发生了错误。

如果成功获取到请求的模块，则可以通过以下三种方式一之在局部命名空间中使用它:

- 模块名后使用 `as` 时，直接把 `as` 后的名称与导入模块绑定。
- 如果没有指定其他名称，且被导入的模块为最高层级模块，则模块的名称将被绑定到局部命名空间作为对所导入模块的引用。
- 如果被导入的模块 *不是*最高层级模块，则包含该模块的最高层级包的名称将被绑定到局部命名空间作为对该最高层级包的引用。所导入的模块必须使用其完整限定名称来访问而不能直接访问。

*from*形式使用的过程略微繁复一些:

1. 查找*from*子句中指定的模块，如有必要还会加载并初始化模块;
2. 对于*import*子句中指定的每个标识符:
    1. 检查被导入模块是否有该名称的属性
    2. 如果没有，尝试导入具有该名称的子模块，然后再次检查被导入模块是否有该属性
    3. 如果未找到该属性，则引发 `ImportError`。
    4. 否则的话，将对该值的引用存入局部命名空间，如果有 `as` 子句则使用其指定的名称，否则使用该属性的名称

示例:

```
import foo                 # foo 被导入并且被局部绑定
import foo.bar.baz         # foo, foo.bar 和 foo.bar.baz 被导入，foo 被局部绑定
import foo.bar.baz as fbb  # foo, foo.bar 和 foo.bar.baz 被导入，foo.bar.baz 被绑定为 fbb
from foo.bar import baz    # foo, foo.bar 和 foo.bar.baz 被导入，foo.bar.baz 被绑定为 baz
from foo import attr       # foo 被导入并且 foo.attr 被绑定为 attr
```

如果标识符列表改为一个星号 (`'*'`)，则在模块中定义的全部公有名称都将按*import*语句所在的作用域被绑定到局部命名空间。

一个模块所定义的 *公有名称*是由在模块的命名空间中检测一个名为 `__all__` 的变量来确定的；如果有定义，它必须是一个字符串列表，其中的项为该模块所定义或导入的名称。在 `__all__` 中所给出的名称都会被视为公有并且应当存在。如果 `__all__` 没有被定义，则公有名称的集合将包含在模块的命名空间中找到的所有不以下划线字符 (`'_'`) 打头的名称。`__all__` 应当包括整个公有 API。它的目标是避免意外地导出不属于 API 的一部分的项（例如在模块内部被导入和使用的库模块）。

通配符形式的导入 --- `from module import *` --- 仅在模块层级上被允许。尝试在类或函数定义中使用它将引发 `SyntaxError`。

当指定要导入哪个模块时，你不必指定模块的绝对名称。当一个模块或包被包含在另一个包之中时，可以在同一个最高层级包中进行相对导入，而不必提及包名称。通过在 *from* 之后指定的模块或包中使用前缀点号，你可以在不指定确切名称的情况下指明在当前包层级结构中要上溯多少级。一个前缀点号表示是执行导入的模块所在的当前包，两个点号表示上溯一个包层级。三个点号表示上溯两级，依此类推。因此如果你执行 `from . import mod` 时所处位置为 `pkg` 包内的一个模块，则最终你将导入 `pkg.mod`。如果你执行 `from ..subpkg2 import mod` 时所处位置为 `pkg.subpkg1` 则你将导入 `pkg.subpkg2.mod`。有关相对导入的规范说明包含在包相对导入 一节中。

`importlib.import_module()` 被提供用来为动态地确定要导入模块的应用提供支持。

引发一个 审计事件 `import` 并附带参数 `module`, `filename`, `sys.path`, `sys.meta_path`, `sys.path_hooks`。

7.11.1 future 语句

future 语句 是一种针对编译器的指令，指明某个特定模块应当使用在特定的未来某个 Python 发行版中成为标准特性的语法或语义。

future 语句的目的是使得向在语言中引入了不兼容改变的 Python 未来版本的迁移更为容易。它允许基于每个模块在某种新特性成为标准之前的发行版中使用该特性。

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

唯一需要使用 future 语句的特性是 标注 (参见 **PEP 563**)。

future 语句所启用的所有历史特性仍然为 Python 3 所认可。其中包括 `absolute_import`, `division`, `generators`, `generator_stop`, `unicode_literals`, `print_function`, `nested_scopes` 和 `with_statement`。它们都已成为冗余项，因为它们总是为已启用状态，保留它们只是为了向后兼容。

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。

在默认情况下，通过对内置函数 `exec()` 和 `compile()` 的调用编译的代码如果出现于一个包含有 future 语句的模块 `M` 之中，就会使用该 future 语句所关联的语法和语义。此行为可以通过传给 `compile()` 的可选参数来控制 --- 请参阅该函数的文档了解详情。

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。

7.12 global 语句

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

global 语句是作用于整个当前代码块的声明。它意味着所列出的标识符将被解读为全局变量。要给全局变量赋值不可能不用到 `global` 关键字，不过自由变量也可以指向全局变量而不必声明为全局变量。

在*global* 语句中列出的名称不得在同一代码块内该 `global` 语句之前的位置中使用。

在*global* 语句中列出的名称不能被定义为形式参数，也不能被作为*with* 语句或*except* 子句的目标，以及*for* 循环的目标列表、*class* 定义、函数定义、*import* 语句或变量标注等等。

CPython 实现细节： 当前的实现并未强制要求所有的上述限制，但程序不应当滥用这样的自由，因为未来的实现可能会改为强制要求，并静默地改变程序的含义。

程序员注意事项: *global* 是对解析器的指令。它仅对与 `global` 语句同时被解析的代码起作用。特别地，包含在提供给内置 `exec()` 函数字符串或代码对象中的 `global` 语句并不会影响 包含该函数调用的代码块，而包含在这种字符串中的代码也不会受到包含该函数调用的代码中的 `global` 语句影响。这同样适用于 `eval()` 和 `compile()` 函数。

7.13 nonlocal 语句

```
nonlocal_stmt  ::=  "nonlocal" identifier ("," identifier)*
```

当一个函数或类的定义嵌套（被包围）在其他函数的定义中时，其非局部作用域就是包围它的函数的局部作用域。*nonlocal* 语句会使其所列出的标识符指向之前在非局部作用域中绑定的名称。它允许封装的代码重新绑定这样的非局部标识符。如果一个名称在多个非局部作用域中都被绑定，则会使用最近的绑定。如果一个名称在任何非局部作用域中都未被绑定，或者不存在非局部作用域，则会引发 `SyntaxError`。

nonlocal 语句的作用范围是整个函数或类语句体。如果一个变量在本作用域的 nonlocal 声明之前被使用或赋值则会引发 `SyntaxError`。

参见

PEP 3104 - 访问外层作用域中的名称
有关*nonlocal* 语句的规范说明。

程序员注意事项: *nonlocal* 是对解析器的指令并且仅会在与其一同被解析的代码上应用。参见*global* 语句的相关注意事项。

7.14 type 语句

```
type_stmt   ::=   'type' identifier [type_params] "=" expression
```

type 语句声明一个类型别名，即 typing.TypeAliasType 的实例。

例如，以下语句创建了一个类型别名:

```
type Point = tuple[float, float]
```

此代码大致等价于:

```
annotation-def VALUE_OF_Point():
    return tuple[float, float]
Point = typing.TypeAliasType("Point", VALUE_OF_Point())
```

annotation-def 指定一个标注作用域，其行为很像是一个函数，但有几个小差别。

类型别名的值是在标注作用域中被求值的。当创建类型别名时它不会被求值，只有当通过该类型别名的 __value__ 属性访问时它才会被求值 (参见惰性求值)。这允许类型别名引用尚未被定义的名称。

类型别名可以通过在名称之后添加类型形参列表 来泛型化。请参阅泛型类型别名 了解详情。

type 是一个软关键字。

Added in version 3.12.

参见

PEP 695 - 类型形参语法
引入了 type 语句和用于泛型类和函数的语法。

CHAPTER 8

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。

if, *while* 和*for* 语句用来实现传统的控制流程构造。*try* 语句为一组语句指定异常处理和/和清理代码，而*with* 语句允许在一个代码块周围执行初始化和终结化代码。函数和类定义在语法上也属于复合语句。

一条复合语句由一个或多个‘子句’组成。一个子句则包含一个句头和一个‘句体’。特定复合语句的子句头都处于相同的缩进层级。每个子句头以一个作为唯一标识的关键字开始并以一个冒号结束。子句体是由一个子句控制的一组语句。子句体可以是在子句头的冒号之后与其同处一行的一条或由分号分隔的多条简单语句，或者也可以是在其之后缩进的一行或多行语句。只有后一种形式的子句体才能包含嵌套的复合语句；以下形式是不合法的，这主要是因为无法分清某个后续的*else* 子句应该属于哪个*if* 子句:

```
if test1: if test2: print(x)
```

还要注意的是在这种情形下分号的绑定比冒号更紧密，因此在以下示例中，所有 `print()` 调用或者都不执行，或者都执行:

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

总结:

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

请注意语句总是以 NEWLINE 结束，之后可能跟随一个 DEDENT。还要注意可选的后续子句总是以一个不能作为语句开头的关键字作为开头，因此不会产生歧义（‘悬空的*else*’问题在 Python 中是通过要求嵌套的*if* 语句必须缩进来解决的)。

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。

8.1 if 语句

if 语句用于有条件的执行:

```
if_stmt   ::=  "if" assignment_expression ":" suite
               ("elif" assignment_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。

8.2 while 语句

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" assignment_expression ":" suite
                  ["else" ":" suite]
```

这将重复地检验表达式，并且如果其值为真就执行第一个子句体；如果表达式值为假（这可能在第一次检验时就发生）则如果 else 子句体存在就会被执行并终止循环。

第一个子句体中的*break* 语句在执行时将终止循环且不执行 else 子句体。第一个子句体中的*continue* 语句在执行时将跳过子句体中的剩余部分并返回检验表达式。

8.3 for 语句

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" starred_list ":" suite
                ["else" ":" suite]
```

starred_list 表达式会被求值一次；它应当产生一个*iterable* 对象。将针对该可迭代对象创建一个*iterator*。随后该迭代器所提供的第一个条目将使用标准的赋值规则被赋值给目标列表 (参见赋值语句)，而代码块将被执行。此过程将针对该迭代器所提供每个条目重复进行。当迭代器被耗尽时，如果存在 else 子句中的代码块，则它将被执行，并终结循环。

第一个子句体中的*break* 语句在执行时将终止循环且不执行 else 子句体。第一个子句体中的*continue* 语句在执行时将跳过子句体中的剩余部分并转往下一项继续执行，或者在没有下一项时转往 else 子句执行。

for 循环会对目标列表中的变量进行赋值。这将覆盖之前对这些变量的所有赋值，包括在 for 循环体中的赋值:

```
for i in range(10):
    print(i)
    i = 5             # this will not affect the for-loop
                      # because i will be overwritten with the next
                      # index in the range
```

目标列表中的名称在循环结束时不会被删除，但是如果序列为空，则它们将根本不会被循环所赋值。提示：内置类型 `range()` 代表由整数组成的不可变算数序列。例如，迭代 `range(3)` 将依次产生 0, 1 和 2。

在 3.11 版本发生变更: 现在允许在表达式列表中使用带星号的元素。

8.4 try 语句

`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 | try3_stmt
try1_stmt  ::=  "try" ":" suite
                ("except" [expression ["as" identifier]] ":" suite)+
                ["else" ":" suite]
                ["finally" ":" suite]
try2_stmt  ::=  "try" ":" suite
                ("except" "*" expression ["as" identifier] ":" suite)+
                ["else" ":" suite]
                ["finally" ":" suite]
try3_stmt  ::=  "try" ":" suite
                "finally" ":" suite
```

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*raise 语句* 一节找到。

8.4.1 except 子句

`except` 子句指定一个或多个异常处理器。当在 *try* 子句中未发生异常时，将不会执行任何异常处理器。当在 `try` 语句块中发生异常时，将启动对异常处理器的搜索。此搜索会依次检查 `except` 子句直至找到与异常相匹配的处理器。不带表达式的 `except` 子句如果存在，则它必须是最后一个；它将匹配任何异常。

对于带有表达式的 `except` 子句，该表达式必须被求值为一个异常类型或是由异常类型组成的元组。被引发的异常将会匹配某个表达式被求值为该异常对象对应的类或其非虚基类，或者是包含该类的元组的 `except` 子句。

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` 子句头部的表达式求值时引发了异常，则对处理器的原始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）。

当代到一个匹配的 `except` 子句时，异常将被赋值给该 `except` 子句在 `as` 关键字之后指定的目标，如果存在此关键字的话，并且该 `except` 子句的代码块将被执行。所有 `except` 子句都必须有可执行的代码块。当到达此类代码块的末尾时，通常会转到整个 *try* 语句之后继续执行。(这意味着如果对同一异常存在两个嵌套的处理器，并且异常发生在内层处理器的 `try` 子句中，则外层处理器将不会处理该异常。)

当使用 `as target` 来为异常赋值时，它将在 `except` 子句结束时被清除。这就相当于

```
except E as N:
    foo
```

被转写为

```
except E as N:
    try:
        foo
    finally:
        del N
```

[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。

这意味着异常必须被赋值给一个不同的名称才能在 except 子句之后引用它。异常会被清除是因为在附加了回溯信息的情况下它们会形成栈帧的循环引用，使得帧中的所有局部变量保持存活直到发生下一次垃圾回收。

在 except 子句的代码块被执行之前，异常将保存在 sys 模块中，在那里它可以从 except 子句的语句体内部通过 sys.exception() 被访问。当离开一个异常处理器时，保存在 sys 模块中的异常将被重置为在此之前的值:

```
>>> print(sys.exception())
None
>>> try:
...     raise TypeError
... except:
...     print(repr(sys.exception()))
...     try:
...         raise ValueError
...     except:
...         print(repr(sys.exception()))
...     print(repr(sys.exception()))
...
TypeError()
ValueError()
TypeError()
>>> print(sys.exception())
None
```

8.4.2 except* 子句

except* 子句被用来处理 ExceptionGroup。要匹配的异常类型将按与*except* 中的相同的方式来解读，但在使用异常组的情况下当类型与组内的某些异常相匹配时我们可以有部分匹配。这意味着有多个 except* 子句可被执行，各自处理异常组的一部分。每个子句最多执行一次并处理所有匹配异常中的一个异常组。组内的每个异常将至多由一个 except* 子句来处理，即第一个与其匹配的子句。

```
>>> try:
...     raise ExceptionGroup("eg",
...         [ValueError(1), TypeError(2), OSError(3), OSError(4)])
... except* TypeError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
... except* OSError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
...
caught <class 'ExceptionGroup'> with nested (TypeError(2),)
caught <class 'ExceptionGroup'> with nested (OSError(3), OSError(4))
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  | ExceptionGroup: eg
  +-+---------------- 1 ----------------
    | ValueError: 1
    +------------------------------------
```

任何未被 except* 子句处理的剩余异常最后都会在 except* 子句中被重新引发。如果此列表包含一个以上的要被重新引发的异常，它们将被合并成一个异常组。

如果被引发的异常不是一个异常组并且其类型与某个 except* 子句相匹配，它将被捕获并由附带空消息字符串的异常组来包装。

```
>>> try:
...     raise BlockingIOError
... except* BlockingIOError as e:
...     print(repr(e))
...
ExceptionGroup('', (BlockingIOError()))
```

except* 子句必须有一个匹配的表达式；它不可为 except*:。并且，该表达式不可包括异常组类型，因为这将导致模糊的语义。

在同一个*try* 中不可以混用*except* 和 except*。*break*, *continue* 和*return* 不可以在 except* 子句中出现。

8.4.3 else 子句

如果控制流离开*try* 子句体时没有引发异常，并且没有执行*return*, *continue* 或*break* 语句，可选的 else 子句将被执行。else 语句中的异常不会由之前的*except* 子句处理。

8.4.4 finally 子句

如果存在 finally，它将指定一个‘清理’处理器。*try* 子句会被执行，包括任何*except* 和*else* 子句。如果在这些子句中发生任何未处理的异常，该异常会被临时保存。finally 子句将被执行。如果存在被保存的异常，它会在 finally 子句的末尾被重新引发。如果 finally 子句引发了另一个异常，被保存的异常会被设为新异常的上下文。如果 finally 子句执行了*return*, *break* 或*continue* 语句，则被保存的异常会被丢弃:

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

在 finally 子句执行期间程序将不能获取到异常信息。

当*return*, *break* 或*continue* 语句在一个 try...finally 语句的*try* 子句的代码块中被执行时，finally 子句也会在‘离开时’被执行。

函数的返回值是由最后被执行的*return* 语句来决定的。由于 finally 子句总是会被执行，因此在 finally 子句中被执行的 return 语句将总是最后被执行的:

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

在 3.8 版本发生变更: 在 Python 3.8 之前，*continue* 语句不允许在 finally 子句中使用，这是因为具体实现中存在一个问题。

8.5 with 语句

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_stmt_contents ","? ")" | with_stmt_contents ) ":" sui
with_stmt_contents  ::=  with_item (","  with_item)*
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）进行求值来获得上下文管理器。

2. 载入上下文管理器的*__enter__()* 以便后续使用。
3. 载入上下文管理器的*__exit__()* 以便后续使用。
4. 发起调用上下文管理器的*__enter__()* 方法。
5. 如果一个目标被包括在*with* 语句中，则把它赋值为*__enter__()* 的返回值。

> **备注**
>
> *with* 语句会保证如果*__enter__()* 方法未发生错误地返回，则*__exit__()* 将一定被调用。因此，如果在对目标列表赋值期间发生错误，它将被当作在语句体内部发生的错误来处理。参见下面的第 7 步。

6. 执行语句体。
7. 发起调用上下文管理器的*__exit__()* 方法。如果语句体的退出是由异常导致的，则其类型、值和回溯信息将被作为参数传递给*__exit__()*。否则的话，将提供三个 `None` 参数。

   如果语句体的退出是由异常导致的，并且来自*__exit__()* 方法的返回值为假，则该异常会被重新引发。如果返回值为真，则该异常会被抑制，并会继续执行*with* 语句之后的语句。

   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。

以下代码:

```
with EXPRESSION as TARGET:
    SUITE
```

在语义上等价于:

```
manager = (EXPRESSION)
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        exit(manager, None, None, None)
```

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
```

在语义上等价于:

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

也可以用圆括号包围的多行形式的多项目上下文管理器。例如:

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

在 3.1 版本发生变更: 支持多个上下文表达式。

在 3.10 版本发生变更: Support for using grouping parentheses to break the statement in multiple lines.

参见

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

8.6 match 语句

Added in version 3.10.

匹配语句用于进行模式匹配。语法如下:

```
match_stmt    ::=   'match' subject_expr ":" NEWLINE INDENT case_block+ DEDENT
subject_expr  ::=   star_named_expression "," star_named_expressions?
                  | named_expression
case_block    ::=   'case' patterns [guard] ":" block
```

备注

本节使用单引号来表示软关键字。

模式匹配接受一个模式作为输入（跟在 case 后），一个目标值（跟在 match 后）。该模式（可能包含子模式）将与目标值进行匹配。输出是:

- 匹配成功或失败（也被称为模式成功或失败）。
- 可能将匹配的值绑定到一个名字上。这方面的先决条件将在下面进一步讨论。

关键字 match 和 case 是*soft keywords* 。

参见

- **PEP 634** ——结构化模式匹配: 规范
- **PEP 636** ——结构化模式匹配: 教程

8.6.1 概述

匹配语句逻辑流程的概述如下:

1. 对目标表达式 subject_expr 求值后将结果作为匹配用的目标值。如果目标表达式包含逗号，则使用 the standard rules 构建一个元组。
2. 目标值将依次与 case_block 中的每个模式进行匹配。匹配成功或失败的具体规则在下面描述。匹配尝试也可以与模式中的一些或所有的独立名称绑定。准确的模式绑定规则因模式类型而异，具体规定见下文。**成功的模式匹配过程中产生的名称绑定将超越所执行的块的范围，可以在匹配语句之后使用**。

> **备注**
>
> 在模式匹配失败时，一些子模式可能会成功。不要依赖于失败匹配进行的绑定。反过来说，不要认为变量在匹配失败后保持不变。确切的行为取决于实现，可能会有所不同。这是一个有意的决定，允许不同的实现添加优化。

3. 如果该模式匹配成功，并且完成了对相应的约束项（如果存在）的求值。在这种情况下，保证完成所有的名称绑定。
   - 如果约束项求值为真或缺失，执行 `case_block` 中的 `block` 。
   - 否则，将按照上述方法尝试下一个 `case_block` 。
   - 如果没有进一步的 case 块，匹配语句终止。

> **备注**
>
> 用户一般不应依赖正在求值的模式。根据不同的实现方式，解释器可能会缓存数值或使用其他优化方法来避免重复求值。

匹配语句示例:

```
>>> flag = False
>>> match (100, 200):
...     case (100, 300):  # Mismatch: 200 != 300
...         print('Case 1')
...     case (100, 200) if flag:  # Successful match, but guard fails
...         print('Case 2')
...     case (100, y):  # Matches and binds y to 200
...         print(f'Case 3, y: {y}')
...     case _:  # Pattern not attempted
...         print('Case 4, I match anything!')
...
Case 3, y: 200
```

在这个示例中，`if flag` 是约束项。请阅读下一节以了解更多相关内容。

8.6.2 约束项

```
guard   ::=   "if" named_expression
```

`guard` (它是 `case` 的一部分) 必须成立才能让 `case` 语句块中的代码被执行。它所采用的形式为: *if* 之后跟一个表达式。

拥有 `guard` 的 `case` 块的逻辑流程如下:

1. 检查 `case` 块中的模式是否匹配成功。如果该模式匹配失败，则不对 `guard` 进行求值，检查下一个 `case` 块。
2. 如果该模式匹配成功，对 `guard` 求值。
   - 如果 `guard` 求值为真，则选用该 case 块。
   - 如果 `guard` 求值为假，则不选用该 case 块。
   - 如果在对 `guard` 求值过程中引发了异常，则异常将被抛出。

允许约束项产生副作用，因为他们是表达式。约束项求值必须从第一个 case 块到最后一个 case 块依次逐个进行，模式匹配失败的 case 块将被跳过。（也就是说，约束项求值必须按顺序进行。）一旦选用了一个 case 块，约束项求值必须由此终止。

8.6.3 必定匹配的 case 块

必定匹配的 case 块是能匹配所有情况的 case 块。一个匹配语句最多可以有一个必定匹配的 case 块，而且必须是最后一个。

如果一个 case 块没有约束项，并且其模式是必定匹配的，那么它就被认为是必定匹配的。如果我们可以仅从语法上证明一个模式总是能匹配成功，那么这个模式就被认为是必定匹配的。只有以下模式是必定匹配的：

- 左侧模式是必定匹配的*AS 模式*
- 包含至少一个必定匹配模式的或模式
- 捕获模式
- 通配符模式
- 括号内的必定匹配模式

8.6.4 模式

> **备注**
>
> 本节使用了超出标准 EBNF 的语法符号。
>
> - 符号 `SEP.RULE+` 是 `RULE (SEP RULE)*` 的简写
> - 符号 `!RULE` 是前向否定断言的简写

`patterns` 的顶层语法是：

```
patterns         ::=  open_sequence_pattern | pattern
pattern          ::=  as_pattern | or_pattern
closed_pattern   ::=  | literal_pattern
                      | capture_pattern
                      | wildcard_pattern
                      | value_pattern
                      | group_pattern
                      | sequence_pattern
                      | mapping_pattern
                      | class_pattern
```

下面的描述将包括一个“简而言之”以描述模式的作用，便于说明问题（感谢 Raymond Hettinger 提供的一份文件，大部分的描述受其启发）。请注意，这些描述纯粹是为了说明问题，**可能不**反映底层的实现。此外，它们并没有涵盖所有有效的形式。

或模式

或模式是由竖杠 `|` 分隔的两个或更多的模式。语法：

```
or_pattern   ::=  "|".closed_pattern+
```

只有最后的子模式可以是必定匹配的，且每个子模式必须绑定相同的名字集以避免歧义。

或模式将目标值依次与其每个子模式尝试匹配，直到有一个匹配成功，然后该或模式被视作匹配成功。否则，如果没有任何子模式匹配成功，则或模式匹配失败。

简而言之，`P1 | P2 | ...` 会首先尝试匹配 `P1`，如果失败将接着尝试匹配 `P2`，如果出现成功的匹配则立即结束且模式匹配成功，否则模式匹配失败。

AS 模式

AS 模式将关键字*as*左侧的或模式与目标值进行匹配。语法：

```
as_pattern   ::=   or_pattern "as" capture_pattern
```

如果 OR 模式匹配失败，则 AS 模式也会失败。在其他情况下，AS 模块会将目标与 as 关键字右边的名称绑定并匹配成功。capture_pattern 不可为 _。

简而言之，P as NAME 将与 P 匹配，成功后将设置 NAME = <subject> 。

字面值模式

字面值模式对应 Python 中的大多数字面值。语法为:

```
literal_pattern   ::=   signed_number
                      | signed_number "+" NUMBER
                      | signed_number "-" NUMBER
                      | strings
                      | "None"
                      | "True"
                      | "False"
signed_number     ::=   ["-"] NUMBER
```

规则 strings 和标记 NUMBER 是在*standard Python grammar* 中定义的。支持三引号的字符串。不支持原始字符串和字节字符串。也不支持*f* 字符串 。

signed_number '+' NUMBER 和 signed_number '-' NUMBER 形式是用于表示复数；它们要求左边是一个实数而右边是一个虚数。例如 3 + 4j。

简而言之，LITERAL 只会在 <subject> == LITERAL 时匹配成功。对于单例 None 、True 和 False ，会使用*is* 运算符。

捕获模式

捕获模式将目标值与一个名称绑定。语法：

```
capture_pattern   ::=   !'_' NAME
```

单独的一个下划线 _ 不是捕获模式（!'_' 表达的就是这个含义）。它会被当作 *wildcard_pattern* 。

在给定的模式中，一个名字只能被绑定一次。例如 case x, x: ... 时无效的，但 case [x] | x: ... 是被允许的。

捕获模式总是能匹配成功。绑定遵循 **PEP 572** 中赋值表达式运算符设立的作用域规则；名字在最接近的包含函数作用域内成为一个局部变量，除非有适用的*global* 或*nonlocal* 语句。

简而言之，NAME 总是会匹配成功且将设置 NAME = <subject> 。

通配符模式

通配符模式总是会匹配成功（匹配任何内容）并且不绑定任何名称。语法：

```
wildcard_pattern   ::=   '_'
```

在且仅在任何模式中 _ 是一个软关键字。通常情况下它是一个标识符，即使是在 match 的目标表达式、guard 和 case 代码块中也是如此。

简而言之，_ 总是会匹配成功。

值模式

值模式代表 Python 中具有名称的值。语法：

```
value_pattern  ::=  attr
attr           ::=  name_or_attr "." NAME
name_or_attr   ::=  attr | NAME
```

模式中带点的名称会使用标准的 Python 名称解析规则 来查找。如果找到的值与目标值比较结果相等则模式匹配成功（使用 == 相等运算符）。

简而言之，`NAME1.NAME2` 仅在 `<subject> == NAME1.NAME2` 时匹配成功。

> **备注**
>
> 如果相同的值在同一个匹配语句中出现多次，解释器可能会缓存找到的第一个值并重新使用它，而不是重复查找。这种缓存与特定匹配语句的执行严格挂钩。

组模式

组模式允许用户在模式周围添加括号，以强调预期的分组。除此之外，它没有额外的语法。语法：

```
group_pattern  ::=  "(" pattern ")"
```

简单来说 `(P)` 具有与 `P` 相同的效果。

序列模式

一个序列模式包含数个将与序列元素进行匹配的子模式。其语法类似于列表或元组的解包。

```
sequence_pattern         ::=  "[" [maybe_sequence_pattern] "]"
                              | "(" [open_sequence_pattern] ")"
open_sequence_pattern    ::=  maybe_star_pattern "," [maybe_sequence_pattern]
maybe_sequence_pattern   ::=  ",".maybe_star_pattern+ ","?
maybe_star_pattern       ::=  star_pattern | pattern
star_pattern             ::=  "*" (capture_pattern | wildcard_pattern)
```

序列模式中使用圆括号或方括号没有区别（例如 `(...)` 和 `[...]` ）。

> **备注**
>
> 用圆括号括起来且没有跟随逗号的单个模式 (例如 `(3 | 4)`) 是一个分组模式。而用方括号括起来的单个模式 (例如 `[3 | 4]`) 则仍是一个序列模式。

一个序列模式中最多可以有一个星号子模式。星号子模式可以出现在任何位置。如果没有星号子模式，该序列模式是固定长度的序列模式；否则，其是一个可变长度的序列模式。

下面是将一个序列模式与一个目标值相匹配的逻辑流程：

1. 如果目标值不是一个序列[2] ，该序列模式匹配失败。

[2] 在模式匹配中，序列被定义为以下几种之一：
- 继承自 `collections.abc.Sequence` 的类
- 注册为 `collections.abc.Sequence` 的 Python 类
- 设置了 (CPython) `Py_TPFLAGS_SEQUENCE` 比特位的内置类
- 继承自上述任何一个类的类

下列标准库中的类都是序列：

2. 如果目标值是 `str` 、`bytes` 或 `bytearray` 的实例，则该序列模式匹配失败。
3. 随后的步骤取决于序列模式是固定长度还是可变长度的。

   如果序列模式是固定长度的：

   1. 如果目标序列的长度与子模式的数量不相等，则该序列模式匹配失败
   2. 序列模式中的子模式与目标序列中的相应项目从左到右进行匹配。一旦一个子模式匹配失败，就停止匹配。如果所有的子模式都成功地与它们的对应项相匹配，那么该序列模式就匹配成功了。

   否则，如果序列模式是变长的：

   1. 如果目标序列的长度小于非星号子模式的数量，则该序列模式匹配失败。
   2. 与固定长度的序列一样，靠前的非星形子模式与其相应的项目进行匹配。
   3. 如果上一步成功，星号子模式与剩余的目标项形成的列表相匹配，不包括星号子模式之后的非星号子模式所对应的剩余项。
   4. 剩余的非星号子模式将与相应的目标项匹配，就像固定长度的序列一样。

   备注

   目标序列的长度可通过 `len()` (即通过 `__len__()` 协议) 获得。解释器可能会以类似于值模式的方式缓存这个长度信息。

简而言之，`[P1, P2, P3, … , P<N>]` 仅在满足以下情况时匹配成功：

- 检查 `<subject>` 是一个序列
- `len(subject) == <N>`
- 将 `P1` 与 `<subject>[0]` 进行匹配（请注意此匹配可以绑定名称）
- 将 `P2` 与 `<subject>[1]` 进行匹配（请注意此匹配可以绑定名称）
- ……剩余对应的模式/元素也以此类推。

映射模式

映射模式包含一个或多个键值模式。其语法类似于字典的构造。语法：

```
mapping_pattern     ::=  "{" [items_pattern] "}"
items_pattern       ::=  ",".key_value_pattern+ ","?
key_value_pattern   ::=  (literal_pattern | value_pattern) ":" pattern
                         | double_star_pattern
double_star_pattern ::=  "**" capture_pattern
```

一个映射模式中最多可以有一个双星号模式。双星号模式必须是映射模式中的最后一个子模式。

- `array.array`
- `collections.deque`
- `list`
- `memoryview`
- `range`
- `tuple`

备注

类型为 `str`, `bytes` 和 `bytearray` 的目标值不能匹配序列模式。

映射模式中不允许出现重复的键。重复的字面值键会引发 SyntaxError 。若是两个键有相同的值将会在运行时引发 ValueError 。

以下是映射模式与目标值匹配的逻辑流程：

1. 如果目标值不是一个映射[3]，则映射模式匹配失败。
2. 若映射模式中给出的每个键都存在于目标映射中，且每个键的模式都与目标映射的相应项匹配成功，则该映射模式匹配成功。
3. 如果在映射模式中检测到重复的键，该模式将被视作无效。对于重复的字面值，会引发 SyntaxError ；对于相同值的命名键，会引发 ValueError 。

备注

键值对使用映射目标的 get() 方法的双参数形式进行匹配。匹配的键值对必须已经存在于映射中，而不是通过 __missing__() 或 *__getitem__()* 即时创建。

简而言之，{KEY1: P1, KEY2: P2, ... } 仅在满足以下情况时匹配成功：

- 检查 <subject> 是映射
- KEY1 in <subject>
- P1 与 <subject>[KEY1] 相匹配
- ……剩余对应的键/模式对也以此类推。

类模式

类模式表示一个类以及它的位置参数和关键字参数（如果有的话）。语法：

```
class_pattern         ::=  name_or_attr "(" [pattern_arguments ","?] ")"
pattern_arguments     ::=  positional_patterns [","  keyword_patterns]
                         | keyword_patterns
positional_patterns   ::=  ",".pattern+
keyword_patterns      ::=  ",".keyword_pattern+
keyword_pattern       ::=  NAME "=" pattern
```

同一个关键词不应该在类模式中重复出现。

以下是类模式与目标值匹配的逻辑流程：

1. 如果 name_or_attr 不是内置 type 的实例，引发 TypeError 。
2. 如果目标值不是 name_or_attr 的实例（通过 isinstance() 测试），该类模式匹配失败。
3. 如果没有模式参数存在，则该模式匹配成功。否则，后面的步骤取决于是否有关键字或位置参数模式存在。

   对于一些内置的类型（将在后文详述），接受一个位置子模式，它将与整个目标值相匹配；对于这些类型，关键字模式也像其他类型一样工作。

   如果只存在关键词模式，它们将被逐一处理，如下所示：

   一. 该关键词被视作主体的一个属性进行查找。

   - 如果这引发了除 AttributeError 以外的异常，该异常会被抛出。

[3] 在模式匹配中，映射被定义为以下几种之一：
- 继承自 collections.abc.Mapping 的类
- 注册为 collections.abc.Mapping 的 Python 类
- 设置了 (CPython) Py_TPFLAGS_MAPPING 比特位的内置类
- 继承自上述任何一个类的类

标准库中的 dict 和 types.MappingProxyType 类都属于映射。

- 如果这引发了 AttributeError ，该类模式匹配失败。
- 否则，与关键词模式相关的子模式将与目标的属性值进行匹配。如果失败，则类模式匹配失败；如果成功，则继续对下一个关键词进行匹配。

二. 如果所有的关键词模式匹配成功，该类模式匹配成功。

如果存在位置模式，在匹配前会用类 name_or_attr 的*__match_args__* 属性将其转换为关键词模式。

一. 进行与 getattr(cls, "__match_args__", ()) 等价的调用。

- 如果这引发一个异常，该异常将被抛出。
- 如果返回值不是一个元组，则转换失败且引发 TypeError 。
- 若位置模式的数量超出 len(cls.__match_args__) ，将引发 TypeError 。
- 否则，位置模式 i 会使用 __match_args__[i] 转换为关键词。__match_args__[i] 必须是一个字符串；如果不是则引发 TypeError 。
- 如果有重复的关键词，引发 TypeError 。

> **参见**
>
> 定制类模式匹配中的位置参数

二. 若所有的位置模式都被转换为关键词模式，
匹配的过程就像只有关键词模式一样。

对于以下内置类型，位置子模式的处理是不同的：

- bool
- bytearray
- bytes
- dict
- float
- frozenset
- int
- list
- set
- str
- tuple

这些类接受一个位置参数，其模式是针对整个对象而不是某个属性进行匹配。例如，int(0|1) 匹配值 0，但不匹配值 0.0。

简而言之，CLS(P1, attr=P2) 仅在满足以下情况时匹配成功：

- isinstance(<subject>, CLS)
- 用 CLS.__match_args__ 将 P1 转换为关键词模式
- 对于每个关键词参数 attr=P2 ：
  - hasattr(<subject>, "attr")
  - 将 P2 与 <subject>.attr 进行匹配
- ……剩余对应的关键字参数/模式对也以此类推。

参见

- **PEP 634** ——结构化模式匹配：规范
- **PEP 636** ——结构化模式匹配：教程

8.7 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）：

```
funcdef                 ::=  [decorators] "def" funcname [type_params] "(" [parameter_list] "
                             ["->" expression] ":" suite
decorators              ::=  decorator+
decorator               ::=  "@" assignment_expression NEWLINE
parameter_list          ::=  defparameter ("," defparameter)* "," "/" ["," [parameter_list_no
                             | parameter_list_no_posonly
parameter_list_no_posonly ::=  defparameter ("," defparameter)* ["," [parameter_list_starargs]]
                             | parameter_list_starargs
parameter_list_starargs ::=  "*" [star_parameter] ("," defparameter)* ["," ["**" parameter ["
                             | "**" parameter [","]
parameter               ::=  identifier [":" expression]
star_parameter          ::=  identifier [":" ["*"] expression]
defparameter            ::=  parameter ["=" expression]
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。[4]

一个函数定义可以被一个或多个*decorator* 表达式所包装。当函数被定义时将在包含该函数定义的作用域中对装饰器表达式求值。求值结果必须是一个可调用对象，它会以该函数对象作为唯一参数被发起调用。其返回值将被绑定到函数名称而非函数对象。多个装饰器会以嵌套方式被应用。例如以下代码

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

大致等价于

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

不同之处在于原始函数并不会被临时绑定到名称 `func`。

在 3.9 版本发生变更: 函数可使用任何有效的 *assignment_expression* 来装饰。在之前版本中，此语法则更为受限，详情参见 **PEP 614**。

可以在函数名及其形参列表开头圆括号之间加方括号给出一个类型形参 的列表。这将向静态类型检查器指明该函数是泛型尾数。在运行时，类型形参可以从函数的*__type_params__* 属性中提取。请参阅泛型函数 了解详情。

在 3.12 版本发生变更: 类型形参列表是在 Python 3.12 中新增的。

当一个或多个形参 具有 *形参* = *表达式*这样的形式时，该函数就被称为具有“默认形参值”。对于一个具有默认值的形参，其对应的*argument* 可以在调用中被省略，在此情况下会用形参的默认值来替代。如果

[4] 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*。

一个形参具有默认值，后续所有在"*" 之前的形参也必须具有默认值 --- 这个句法限制并未在语法中明确表达。

默认形参值会在执行函数定义时按从左至右的顺序被求值。这意味着当函数被定义时将对表达式求值一次，相同的 "预计算" 值将在每次调用时被使用。这一点在默认形参为可变对象，例如列表或字典的时候尤其需要重点理解：如果函数修改了该对象（例如向列表添加了一项），则实际上默认值也会被修改。这通常不是人们所想要的。绕过此问题的一个方法是使用 `None` 作为默认值，并在函数体中显式地对其进测试，例如:

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

函数调用的语义在调用 一节中有更详细的描述。函数调用总是会给形参列表中列出的所有形参赋值，或是用位置参数，或是用关键字参数，或是用默认值。如果存在"`*identifier`" 这样的形式，它会被初始化为一个元组来接收任何额外的位置参数，默认为一个空元组。如果存在"`**identifier`" 这样的形式，它会被初始化为一个新的有序映射来接收任何额外的关键字参数，默认为一个相同类型的空映射。在"*" 或"`*identifier`" 之后的形参都是仅限关键字形参因而只能通过关键字参数传入。在"/" 之前的形参都是仅限位置形参因而只能通过位置参数传入。

在 3.8 版本发生变更: 可以使用 / 函数形参语法来标示仅限位置形参。请参阅 **PEP 570** 了解详情。

Parameters may have an *annotation* of the form "`: expression`" following the parameter name. Any parameter may have an annotation, even those of the form `*identifier` or `**identifier`. (As a special case, parameters of the form `*identifier` may have an annotation "`: *expression`".) Functions may have "return" annotation of the form "`-> expression`" after the parameter list. These annotations can be any valid Python expression. The presence of annotations does not change the semantics of a function. The annotation values are available as values of a dictionary keyed by the parameters' names in the `__annotations__` attribute of the function object. If the `annotations` import from `__future__` is used, annotations are preserved as strings at runtime which enables postponed evaluation. Otherwise, they are evaluated when the function definition is executed. In this case annotations may be evaluated in a different order than they appear in the source code.

在 3.11 版本发生变更: Parameters of the form "`*identifier`" may have an annotation "`: *expression`". See **PEP 646**.

创建匿名函数（未绑定到一个名称的函数）以便立即在表达式中使用也是可能的。这需要使用 lambda 表达式，具体描述见*lambda 表达式* 一节。请注意 lambda 只是简单函数定义的一种简化写法；在"*def*" 语句中定义的函数也可以像用 lambda 表达式定义的函数一样被传递或赋值给其他名称。"`def`" 形式实际上更为强大，因为它允许执行多条语句和使用标注。

程序员注意事项: 函数属于一类对象。在一个函数内部执行的"`def`" 语句会定义一个局部函数并可被返回或传递。在嵌套函数中使用的自由变量可以访问包含该 def 语句的函数的局部变量。详情参见*命名与绑定* 一节。

参见

PEP 3107 - 函数标注
最初的函数标注规范说明。

PEP 484 ——类型注解
标注的标准含意定义：类型提示。

PEP 526 - 变量标注的语法
变量声明的类型提示功能，包括类变量和实例变量。

PEP 563 - 延迟的标注求值
支持在运行时通过以字符串形式保存标注而非不是即求值来实现标注内部的向前引用。

PEP 318 - 函数和方法的装饰器
引入了函数和方法的装饰器。类装饰器是在 **PEP 3129** 中引入的。

8.8 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      ::=  [decorators] "class" classname [type_params] [inheritance] ":" suite
inheritance   ::=  "(" [argument_list] ")"
classname     ::=  identifier
```

类定义是一条可执行语句。其中继承列表通常给出基类的列表 (进阶用法请参见元类)，列表中的每一项都应当被求值为一个允许子类的类对象。没有继承列表的类默认继承自基类 `object`；因此，：

```
class Foo:
    pass
```

等价于

```
class Foo(object):
    pass
```

随后类体将在一个新的执行帧 (参见命名与绑定) 中被执行，使用新创建的局部命名空间和原有的全局命名空间。(通常，类体主要包含函数定义。) 当类体结束执行时，其执行帧将被丢弃而其局部命名空间会被保存。[5] 一个类对象随后会被创建，其基类使用给定的继承列表，属性字典使用保存的局部命名空间。类名称将在原有的全局命名空间中绑定到该类对象。

The order in which attributes are defined in the class body is preserved in the new class's *__dict__*. Note that this is reliable only right after the class is created and only for classes that were defined using the definition syntax.

类的创建可使用元类 进行重度定制。

类也可以被装饰：就像装饰函数一样，：

```
@f1(arg)
@f2
class Foo: pass
```

大致等价于

```
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

装饰器表达式的求值规则与函数装饰器相同。结果随后会被绑定到类名称。

在 3.9 版本发生变更: 类可使用任何有效的 *assignment_expression* 来装饰。在之前版本中，此语法则更为受限，详情参见 **PEP 614**。

A list of *type parameters* may be given in square brackets immediately after the class's name. This indicates to static type checkers that the class is generic. At runtime, the type parameters can be retrieved from the class's *__type_params__* attribute. See 泛型类 for more.

在 3.12 版本发生变更: 类型形参列表是在 Python 3.12 中新增的。

程序员注意事项: 在类定义内定义的变量是类属性；它们将被类实例所共享。实例属性可通过 `self.name = value` 在方法中设定。类和实例属性均可通过"`self.name`" 表示法来访问，当通过此方式访问时实例属性会隐藏同名的类属性。类属性可被用作实例属性的默认值，但在此场景下使用可变值可能导致未预期的结果。可以使用描述器 来创建具有不同实现细节的实例变量。

参见

[5] A string literal appearing as the first statement in the class body is transformed into the namespace's *__doc__* item and therefore the class's *docstring*.

PEP 3115 - Python 3000 中的元类
将元类声明修改为当前语法的提议，以及关于如何构建带有元类的类的语义描述。

PEP 3129 - 类装饰器
增加类装饰器的提议。函数和方法装饰器是在 **PEP 318** 中被引入的。

8.9 协程

Added in version 3.5.

8.9.1 协程函数定义

```
async_funcdef   ::=   [decorators] "async" "def" funcname "(" [parameter_list] ")"
                      ["->" expression] ":" suite
```

Python 协程的执行可以在多个位置上被挂起和恢复 (参见*coroutine*)。`await` 表达式，`async for` 以及`async with` 只能在协程函数体中使用。

使用 `async def` 语法定义的函数总是为协程函数，即使它们不包含 `await` 或 `async` 关键字。

在协程函数体中使用 `yield from` 表达式将引发 `SyntaxError`。

协程函数的例子:

```
async def func(param1, param2):
    do_stuff()
    await some_coroutine()
```

在 3.7 版本发生变更: `await` 和 `async` 现在是保留关键字；在之前版本中它们仅在协程函数内被当作保留关键字。

8.9.2 `async for` 语句

```
async_for_stmt   ::=   "async" for_stmt
```

asynchronous iterable 提供了 `__aiter__` 方法，该方法会直接返回*asynchronous iterator*，它可以在其 `__anext__` 方法中调用异步代码。

`async for` 语句允许方便地对异步可迭代对象进行迭代。

以下代码:

```
async for TARGET in ITER:
    SUITE
else:
    SUITE2
```

在语义上等价于:

```
iter = (ITER)
iter = type(iter).__aiter__(iter)
running = True

while running:
    try:
        TARGET = await type(iter).__anext__(iter)
    except StopAsyncIteration:
        running = False
    else:
```

(续下页)

(接上页)

```
        SUITE
else:
    SUITE2
```

另请参阅*__aiter__()* 和*__anext__()* 了解详情。

在协程函数体之外使用 `async for` 语句将引发 `SyntaxError`。

8.9.3 `async with` 语句

```
async_with_stmt   ::=   "async" with_stmt
```

asynchronous context manager 是一种*context manager*，能够在其 *enter* 和 *exit* 方法中暂停执行。

以下代码:

```
async with EXPRESSION as TARGET:
    SUITE
```

在语义上等价于:

```
manager = (EXPRESSION)
aenter = type(manager).__aenter__
aexit = type(manager).__aexit__
value = await aenter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not await aexit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        await aexit(manager, None, None, None)
```

另请参阅*__aenter__()* 和*__aexit__()* 了解详情。

在协程函数体之外使用 `async with` 语句将引发 `SyntaxError`。

参见

PEP 492 - 使用 async 和 await 语法实现协程
将协程作为 Python 中的一个正式的单独概念，并增加相应的支持语法。

8.10 类型形参列表

Added in version 3.12.

在 3.13 版本发生变更: 增加了对默认值的支持 (参见 **PEP 696**)。

```
type_params   ::=   "[" type_param ("," type_param)* "]"
type_param    ::=   typevar | typevartuple | paramspec
typevar       ::=   identifier (":" expression)? ("=" expression)?
typevartuple  ::=   "*" identifier ("=" expression)?
```

```
paramspec      ::=    "**" identifier ("=" expression)?
```

函数 (包括协程), 类 和类型别名 可能包含类型形参列表:

```
def max[T](args: list[T]) -> T:
    ...

async def amax[T](args: list[T]) -> T:
    ...

class Bag[T]:
    def __iter__(self) -> Iterator[T]:
        ...

    def add(self, arg: T) -> None:
        ...

type ListOrSet[T] = list[T] | set[T]
```

从语义上讲，这表明函数、类或类型别名是类型变量的泛型。此信息主要供静态类型检查器使用，并且在运行时，泛型对象的行为与其对应的非泛型对象非常相似。

类型参数是紧接在函数、类或类型别名的名称之后的方括号 (`[]`) 中声明的。类型参数可在泛型对象的作用域内访问，但不能在其他地方访问。因此，在声明 `def func[T](): pass` 之后，模块作用域中就不能再使用 `T` 这个名称。在下文中，将更精确地描述泛型对象的语义。类型形参的作用域是用一个特殊函数 (从技术上说，是一个标注作用域) 来模拟的，它封装了泛型对象的创建操作。

Generic functions, classes, and type aliases have a `__type_params__` attribute listing their type parameters.

类型形参可分为三种:

- `typing.TypeVar`，由一个普通名称 (例如 `T`) 引入。从语义上讲，这对类型检查器来说代表了一个单独类型。
- `typing.TypeVarTuple`，通过在前面添加一个星号的名称来引入 (例如 `*Ts`)。从语义上讲，它代表由任意多个类型组成的元组。
- `typing.ParamSpec`，通过在前面添加两个星号的名称来引入 (例如 `**P`)。从语义上讲，它代表一个可调用对象的形参。

`typing.TypeVar` 声明可以通过在冒号 (`:`) 后跟一个表达式来定义 *范围* 和 *约束*。冒号后的单独表达式表示一个范围 (例如 `T: int`)。从语义上讲，这意味着 `typing.TypeVar` 能表示的类型只能是该范围的子类型。冒号后在圆括号内的表达式元组指定了一组约束 (例如 `T: (str, bytes)`)。元组中的每个成员都应为一个类型 (同样，在运行时并不强制要求这一点)。约束的类型变量只能使用约束列表内的类型中选择一种。

对于使用类型形参列表语法声明的 `typing.TypeVar`，范围和约束在创建泛型对象时并不会被求值，只有在通过属性 `__bound__` 和 `__constraints__` 显式地访问它时才会被求值。要做到这一点，需要在单独的标注作用域 中对范围和约束进行求值。

`typing.TypeVarTuple` 和 `typing.ParamSpec` 不能拥有范围或约束。

所有三种风格的类型形参都还可以具有 *默认值*，它会在未显式提供类型形参值时被使用。这是通过添加单个等号 (`=`) 跟一个表达式来添加的。与类型变量的绑定和约束类似，默认值不是在创建对象时被求值的，而是在类型形参的 `__default__` 属性被访问的时候。为此，默认值将在单独的标注作用域 中被求值。如果没有为类型形参指定默认值，`__default__` 属性将被设为特殊的哨兵对象 `typing.NoDefault`。

下面的例子显示了所有被允许的类型形参声明:

```
def overly_generic[
   SimpleTypeVar,
   TypeVarWithDefault = int,
   TypeVarWithBound: int,
   TypeVarWithConstraints: (str, bytes),
   *SimpleTypeVarTuple = (int, float),
```

(续下页)

(接上页)

```
   **SimpleParamSpec = (str, bytearray),
](
   a: SimpleTypeVar,
   b: TypeVarWithDefault,
   c: TypeVarWithBound,
   d: Callable[SimpleParamSpec, TypeVarWithConstraints],
   *e: SimpleTypeVarTuple,
): ...
```

8.10.1 泛型函数

泛型函数的声明方式如下:

```
def func[T](arg: T): ...
```

该语法等价于:

```
annotation-def TYPE_PARAMS_OF_func():
    T = typing.TypeVar("T")
    def func(arg: T): ...
    func.__type_params__ = (T,)
    return func
func = TYPE_PARAMS_OF_func()
```

这里 annotation-def 指定了一个标注作用域，它在运行时并不会实际绑定到任何名称。(另一项自由是在翻译中达成的：该语法没有通过 typing 模块的属性访问，而是直接创建了一个 typing.TypeVar 的实例)。

泛型函数的标注会在用于声明类型形参的标注作用域内进行求值，但函数的默认值和装饰器则不会。

下面的例子演示了针对这些场景，以及类型形参的变化形式的作用域规则:

```
@decorator
def func[T: int, *Ts, **P](*args: *Ts, arg: Callable[P, T] = some_default):
    ...
```

除了 TypeVar 绑定的惰性求值 以外，这等同于:

```
DEFAULT_OF_arg = some_default

annotation-def TYPE_PARAMS_OF_func():

    annotation-def BOUND_OF_T():
        return int
    # In reality, BOUND_OF_T() is evaluated only on demand.
    T = typing.TypeVar("T", bound=BOUND_OF_T())

    Ts = typing.TypeVarTuple("Ts")
    P = typing.ParamSpec("P")

    def func(*args: *Ts, arg: Callable[P, T] = DEFAULT_OF_arg):
        ...

    func.__type_params__ = (T, Ts, P)
    return func
func = decorator(TYPE_PARAMS_OF_func())
```

大写形式的名称如 DEFAULT_OF_arg 在运行时不会被实际绑定。

8.10.2 泛型类

泛型类的声明方式如下:

```
class Bag[T]: ...
```

该语法等价于:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(typing.Generic[T]):
        __type_params__ = (T,)
        ...
    return Bag
Bag = TYPE_PARAMS_OF_Bag()
```

这里还是用 annotation-def (不是真正的关键字) 指明标注作用域，而名称 TYPE_PARAMS_OF_Bag 在不会运行时实际被绑定。

泛型类隐式地继承自 typing.Generic。泛型类的基类和关键字参数在类型形参的类型作用域内进行求值，而装饰器则在该作用域之外进行求值。以下示例对此进行了说明:

```
@decorator
class Bag(Base[T], arg=T): ...
```

这相当于:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(Base[T], typing.Generic[T], arg=T):
        __type_params__ = (T,)
        ...
    return Bag
Bag = decorator(TYPE_PARAMS_OF_Bag())
```

8.10.3 泛型类型别名

type 语句也可被用来创建泛型类型别名:

```
type ListOrSet[T] = list[T] | set[T]
```

除了会对值执行惰性求值 以外，这等同于:

```
annotation-def TYPE_PARAMS_OF_ListOrSet():
    T = typing.TypeVar("T")

    annotation-def VALUE_OF_ListOrSet():
        return list[T] | set[T]
    # In reality, the value is lazily evaluated
    return typing.TypeAliasType("ListOrSet", VALUE_OF_ListOrSet(), type_params=(T,))
ListOrSet = TYPE_PARAMS_OF_ListOrSet()
```

这里，annotation-def (不是一个真正的关键字) 指明标注作用域。像 TYPE_PARAMS_OF_ListOrSet 这样的大写名称不会在运行时实际被绑定。

备注

CHAPTER 9

顶级组件

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。

9.1 完整的 Python 程序

虽然语言规范描述不必规定如何发起调用语言解释器，但对完整的 Python 程序加以说明还是很有用的。一个完整的 Python 程序会在最小初始化环境中被执行：所有内置和标准模块均为可用，但均处于未初始化状态，只有 `sys` (各种系统服务), `builtins` (内置函数、异常以及 `None`) 和 `__main__` 除外。最后一个模块用于为完整程序的执行提供局部和全局命名空间。

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。

一个完整程序可通过三种形式被传递给解释器：使用 `-c` *字符串*命令行选项，使用一个文件作为第一个命令行参数，或者使用标准输入。如果文件或标准输入是一个 tty 设置，解释器会进入交互模式；否则的话，它会将文件当作一个完整程序来执行。

9.2 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;
- 解析一个传递给 `exec()` 函数的字符串时;

9.3 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。

9.4 表达式输入

`eval()` 被用于表达式输入。它会忽略开头的空白。传递给 `eval()` 的字符串参数必须具有以下形式:

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

CHAPTER 10

完整的语法规范

这是完整的 Python 语法规范，直接提取自用于生成 CPython 解析器的语法 (参见 Grammar/python.gram)。这里显示的版本省略了有关代码生成和错误恢复的细节。

该标记法是 EBNF 和 PEG 的混合体。特别地，& 后跟一个符号、形符或带括号的分组来表示肯定型前视（即要求匹配但不消耗字符）。而 ! 表示否定型前视（即要求 不匹配）。我们使用 | 分隔符来表示 PEG 的“有序选择”（在传统 PEG 语法中为 / 写法）。请参阅 **PEP 617** 了解有关该语法规则的更多细节。

```
# PEG grammar for Python



# ========================= START OF THE GRAMMAR =========================

# General grammatical elements and rules:
#
# * Strings with double quotes (") denote SOFT KEYWORDS
# * Strings with single quotes (') denote KEYWORDS
# * Upper case names (NAME) denote tokens in the Grammar/Tokens file
# * Rule names starting with "invalid_" are used for specialized syntax errors
#     - These rules are NOT used in the first pass of the parser.
#     - Only if the first pass fails to parse, a second pass including the invalid
#       rules will be executed.
#     - If the parser fails in the second phase with a generic syntax error, the
#       location of the generic failure of the first pass will be used (this avoids
#       reporting incorrect locations due to the invalid rules).
#     - The order of the alternatives involving invalid rules matter
#       (like any rule in PEG).
#
# Grammar Syntax (see PEP 617 for more information):
#
# rule_name: expression
#   Optionally, a type can be included right after the rule name, which
#   specifies the return type of the C or Python function corresponding to the
#   rule:
# rule_name[return_type]: expression
#   If the return type is omitted, then a void * is returned in C and an Any in
#   Python.
# e1 e2
#   Match e1, then match e2.
```

(续下页)

(接上页)

```
# e1 | e2
#   Match e1 or e2.
#   The first alternative can also appear on the line after the rule name for
#   formatting purposes. In that case, a | must be used before the first
#   alternative, like so:
#       rule_name[return_type]:
#            | first_alt
#            | second_alt
# ( e )
#   Match e (allows also to use other operators in the group like '(e)*')
# [ e ] or e?
#   Optionally match e.
# e*
#   Match zero or more occurrences of e.
# e+
#   Match one or more occurrences of e.
# s.e+
#   Match one or more occurrences of e, separated by s. The generated parse tree
#   does not include the separator. This is otherwise identical to (e (s e)*).
# &e
#   Succeed if e can be parsed, without consuming any input.
# !e
#   Fail if e can be parsed, without consuming any input.
# ~
#   Commit to the current alternative, even if it fails to parse.
# &&e
#   Eager parse e. The parser will not backtrack and will immediately
#   fail with SyntaxError if e cannot be parsed.
#

# STARTING RULES
# ==============

file: [statements] ENDMARKER
interactive: statement_newline
eval: expressions NEWLINE* ENDMARKER
func_type: '(' [type_expressions] ')' '->' expression NEWLINE* ENDMARKER

# GENERAL STATEMENTS
# ==================

statements: statement+

statement: compound_stmt  | simple_stmts

statement_newline:
    | compound_stmt NEWLINE
    | simple_stmts
    | NEWLINE
    | ENDMARKER

simple_stmts:
    | simple_stmt !';' NEWLINE  # Not needed, there for speedup
    | ';'.simple_stmt+ [';'] NEWLINE

# NOTE: assignment MUST precede expression, else parsing a simple assignment
# will throw a SyntaxError.
simple_stmt:
    | assignment
    | type_alias
    | star_expressions
```

(续下页)

(接上页)

```
    | return_stmt
    | import_stmt
    | raise_stmt
    | 'pass'
    | del_stmt
    | yield_stmt
    | assert_stmt
    | 'break'
    | 'continue'
    | global_stmt
    | nonlocal_stmt

compound_stmt:
    | function_def
    | if_stmt
    | class_def
    | with_stmt
    | for_stmt
    | try_stmt
    | while_stmt
    | match_stmt

# SIMPLE STATEMENTS
# =================

# NOTE: annotated_rhs may start with 'yield'; yield_expr must start with 'yield'
assignment:
    | NAME ':' expression ['=' annotated_rhs ]
    | ('(' single_target ')'
         | single_subscript_attribute_target) ':' expression ['=' annotated_rhs ]
    | (star_targets '=' )+ (yield_expr | star_expressions) !'=' [TYPE_COMMENT]
    | single_target augassign ~ (yield_expr | star_expressions)

annotated_rhs: yield_expr | star_expressions

augassign:
    | '+='
    | '-='
    | '*='
    | '@='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '<<='
    | '>>='
    | '**='
    | '//='

return_stmt:
    | 'return' [star_expressions]

raise_stmt:
    | 'raise' expression ['from' expression ]
    | 'raise'

global_stmt: 'global' ','.NAME+

nonlocal_stmt: 'nonlocal' ','.NAME+
```

(续下页)

(接上页)

```
del_stmt:
    | 'del' del_targets &(';' | NEWLINE)

yield_stmt: yield_expr

assert_stmt: 'assert' expression [',' expression ]

import_stmt:
    | import_name
    | import_from

# Import statements
# -----------------

import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
import_from:
    | 'from' ('.' | '...')* dotted_name 'import' import_from_targets
    | 'from' ('.' | '...')+ 'import' import_from_targets
import_from_targets:
    | '(' import_from_as_names [','] ')'
    | import_from_as_names !','
    | '*'
import_from_as_names:
    | ','.import_from_as_name+
import_from_as_name:
    | NAME ['as' NAME ]
dotted_as_names:
    | ','.dotted_as_name+
dotted_as_name:
    | dotted_name ['as' NAME ]
dotted_name:
    | dotted_name '.' NAME
    | NAME

# COMPOUND STATEMENTS
# ===================

# Common elements
# ---------------

block:
    | NEWLINE INDENT statements DEDENT
    | simple_stmts

decorators: ('@' named_expression NEWLINE )+

# Class definitions
# -----------------

class_def:
    | decorators class_def_raw
    | class_def_raw

class_def_raw:
    | 'class' NAME [type_params] ['(' [arguments] ')' ] ':' block

# Function definitions
# --------------------

function_def:
```

(续下页)

(接上页)

```
    | decorators function_def_raw
    | function_def_raw

function_def_raw:
    | 'def' NAME [type_params] '(' [params] ')' ['->' expression ] ':' [func_type_comment]␣
↪block
    | 'async' 'def' NAME [type_params] '(' [params] ')' ['->' expression ] ':' [func_type_
↪comment] block

# Function parameters
# --------------------

params:
    | parameters

parameters:
    | slash_no_default param_no_default* param_with_default* [star_etc]
    | slash_with_default param_with_default* [star_etc]
    | param_no_default+ param_with_default* [star_etc]
    | param_with_default+ [star_etc]
    | star_etc

# Some duplication here because we can't write (',' | &')'),
# which is because we don't support empty alternatives (yet).

slash_no_default:
    | param_no_default+ '/' ','
    | param_no_default+ '/' &')'
slash_with_default:
    | param_no_default* param_with_default+ '/' ','
    | param_no_default* param_with_default+ '/' &')'

star_etc:
    | '*' param_no_default param_maybe_default* [kwds]
    | '*' param_no_default_star_annotation param_maybe_default* [kwds]
    | '*' ',' param_maybe_default+ [kwds]
    | kwds

kwds:
    | '**' param_no_default

# One parameter.  This *includes* a following comma and type comment.
#
# There are three styles:
# - No default
# - With default
# - Maybe with default
#
# There are two alternative forms of each, to deal with type comments:
# - Ends in a comma followed by an optional type comment
# - No comma, optional type comment, must be followed by close paren
# The latter form is for a final parameter without trailing comma.
#

param_no_default:
    | param ',' TYPE_COMMENT?
    | param TYPE_COMMENT? &')'
param_no_default_star_annotation:
    | param_star_annotation ',' TYPE_COMMENT?
    | param_star_annotation TYPE_COMMENT? &')'
param_with_default:
```

(续下页)

(接上页)

```
    | param default ',' TYPE_COMMENT?
    | param default TYPE_COMMENT? &')'
param_maybe_default:
    | param default? ',' TYPE_COMMENT?
    | param default? TYPE_COMMENT? &')'
param: NAME annotation?
param_star_annotation: NAME star_annotation
annotation: ':' expression
star_annotation: ':' star_expression
default: '=' expression  | invalid_default

# If statement
# ------------

if_stmt:
    | 'if' named_expression ':' block elif_stmt
    | 'if' named_expression ':' block [else_block]
elif_stmt:
    | 'elif' named_expression ':' block elif_stmt
    | 'elif' named_expression ':' block [else_block]
else_block:
    | 'else' ':' block

# While statement
# ---------------

while_stmt:
    | 'while' named_expression ':' block [else_block]

# For statement
# -------------

for_stmt:
    | 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block [else_block]
    | 'async' 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block [else_
↪block]

# With statement
# --------------

with_stmt:
    | 'with' '(' ','.with_item+ ','? ')' ':' [TYPE_COMMENT] block
    | 'with' ','.with_item+ ':' [TYPE_COMMENT] block
    | 'async' 'with' '(' ','.with_item+ ','? ')' ':' block
    | 'async' 'with' ','.with_item+ ':' [TYPE_COMMENT] block

with_item:
    | expression 'as' star_target &(',' | ')' | ':')
    | expression

# Try statement
# -------------

try_stmt:
    | 'try' ':' block finally_block
    | 'try' ':' block except_block+ [else_block] [finally_block]
    | 'try' ':' block except_star_block+ [else_block] [finally_block]


# Except statement
# ----------------
```

(续下页)

(接上页)

```
except_block:
    | 'except' expression ['as' NAME ] ':' block
    | 'except' ':' block
except_star_block:
    | 'except' '*' expression ['as' NAME ] ':' block
finally_block:
    | 'finally' ':' block

# Match statement
# ---------------

match_stmt:
    | "match" subject_expr ':' NEWLINE INDENT case_block+ DEDENT

subject_expr:
    | star_named_expression ',' star_named_expressions?
    | named_expression

case_block:
    | "case" patterns guard? ':' block

guard: 'if' named_expression

patterns:
    | open_sequence_pattern
    | pattern

pattern:
    | as_pattern
    | or_pattern

as_pattern:
    | or_pattern 'as' pattern_capture_target

or_pattern:
    | '|'.closed_pattern+

closed_pattern:
    | literal_pattern
    | capture_pattern
    | wildcard_pattern
    | value_pattern
    | group_pattern
    | sequence_pattern
    | mapping_pattern
    | class_pattern

# Literal patterns are used for equality and identity constraints
literal_pattern:
    | signed_number !('+' | '-')
    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'

# Literal expressions are used to restrict permitted mapping pattern keys
literal_expr:
    | signed_number !('+' | '-')
    | complex_number
```

(续下页)

(接上页)

```
    | strings
    | 'None'
    | 'True'
    | 'False'

complex_number:
    | signed_real_number '+' imaginary_number
    | signed_real_number '-' imaginary_number

signed_number:
    | NUMBER
    | '-' NUMBER

signed_real_number:
    | real_number
    | '-' real_number

real_number:
    | NUMBER

imaginary_number:
    | NUMBER

capture_pattern:
    | pattern_capture_target

pattern_capture_target:
    | !"_" NAME !('.' | '(' | '=')

wildcard_pattern:
    | "_"

value_pattern:
    | attr !('.' | '(' | '=')

attr:
    | name_or_attr '.' NAME

name_or_attr:
    | attr
    | NAME

group_pattern:
    | '(' pattern ')'

sequence_pattern:
    | '[' maybe_sequence_pattern? ']'
    | '(' open_sequence_pattern? ')'

open_sequence_pattern:
    | maybe_star_pattern ',' maybe_sequence_pattern?

maybe_sequence_pattern:
    | ','.maybe_star_pattern+ ','?

maybe_star_pattern:
    | star_pattern
    | pattern

star_pattern:
    | '*' pattern_capture_target
```

(续下页)

(接上页)

```
    | '*' wildcard_pattern

mapping_pattern:
    | '{' '}'
    | '{' double_star_pattern ','? '}'
    | '{' items_pattern ',' double_star_pattern ','? '}'
    | '{' items_pattern ','? '}'

items_pattern:
    | ','.key_value_pattern+

key_value_pattern:
    | (literal_expr | attr) ':' pattern

double_star_pattern:
    | '**' pattern_capture_target

class_pattern:
    | name_or_attr '(' ')'
    | name_or_attr '(' positional_patterns ','? ')'
    | name_or_attr '(' keyword_patterns ','? ')'
    | name_or_attr '(' positional_patterns ',' keyword_patterns ','? ')'

positional_patterns:
    | ','.pattern+

keyword_patterns:
    | ','.keyword_pattern+

keyword_pattern:
    | NAME '=' pattern

# Type statement
# ---------------

type_alias:
    | "type" NAME [type_params] '=' expression

# Type parameter declaration
# --------------------------

type_params:
    | invalid_type_params
    | '[' type_param_seq ']'

type_param_seq: ','.type_param+ [',']

type_param:
    | NAME [type_param_bound] [type_param_default]
    | '*' NAME [type_param_starred_default]
    | '**' NAME [type_param_default]

type_param_bound: ':' expression
type_param_default: '=' expression
type_param_starred_default: '=' star_expression

# EXPRESSIONS
# -----------

expressions:
    | expression (',' expression )+ [',']
```

(续下页)

(接上页)

```
    | expression ','
    | expression

expression:
    | disjunction 'if' disjunction 'else' expression
    | disjunction
    | lambdef

yield_expr:
    | 'yield' 'from' expression
    | 'yield' [star_expressions]

star_expressions:
    | star_expression (',' star_expression )+ [',']
    | star_expression ','
    | star_expression

star_expression:
    | '*' bitwise_or
    | expression

star_named_expressions: ','.star_named_expression+ [',']

star_named_expression:
    | '*' bitwise_or
    | named_expression

assignment_expression:
    | NAME ':=' ~ expression

named_expression:
    | assignment_expression
    | expression !':='

disjunction:
    | conjunction ('or' conjunction )+
    | conjunction

conjunction:
    | inversion ('and' inversion )+
    | inversion

inversion:
    | 'not' inversion
    | comparison

# Comparison operators
# --------------------

comparison:
    | bitwise_or compare_op_bitwise_or_pair+
    | bitwise_or

compare_op_bitwise_or_pair:
    | eq_bitwise_or
    | noteq_bitwise_or
    | lte_bitwise_or
    | lt_bitwise_or
    | gte_bitwise_or
    | gt_bitwise_or
    | notin_bitwise_or
```

(续下页)

(接上页)

```
    | in_bitwise_or
    | isnot_bitwise_or
    | is_bitwise_or

eq_bitwise_or: '==' bitwise_or
noteq_bitwise_or:
    | ('!=' ) bitwise_or
lte_bitwise_or: '<=' bitwise_or
lt_bitwise_or: '<' bitwise_or
gte_bitwise_or: '>=' bitwise_or
gt_bitwise_or: '>' bitwise_or
notin_bitwise_or: 'not' 'in' bitwise_or
in_bitwise_or: 'in' bitwise_or
isnot_bitwise_or: 'is' 'not' bitwise_or
is_bitwise_or: 'is' bitwise_or

# Bitwise operators
# -----------------

bitwise_or:
    | bitwise_or '|' bitwise_xor
    | bitwise_xor

bitwise_xor:
    | bitwise_xor '^' bitwise_and
    | bitwise_and

bitwise_and:
    | bitwise_and '&' shift_expr
    | shift_expr

shift_expr:
    | shift_expr '<<' sum
    | shift_expr '>>' sum
    | sum

# Arithmetic operators
# --------------------

sum:
    | sum '+' term
    | sum '-' term
    | term

term:
    | term '*' factor
    | term '/' factor
    | term '//' factor
    | term '%' factor
    | term '@' factor
    | factor

factor:
    | '+' factor
    | '-' factor
    | '~' factor
    | power

power:
    | await_primary '**' factor
    | await_primary
```

(续下页)

(接上页)

```
# Primary elements
# ----------------

# Primary elements are things like "obj.something.something", "obj[something]",
↪"obj(something)", "obj" ...

await_primary:
    | 'await' primary
    | primary

primary:
    | primary '.' NAME
    | primary genexp
    | primary '(' [arguments] ')'
    | primary '[' slices ']'
    | atom

slices:
    | slice !','
    | ','.(slice | starred_expression)+ [',']

slice:
    | [expression] ':' [expression] [':' [expression] ]
    | named_expression

atom:
    | NAME
    | 'True'
    | 'False'
    | 'None'
    | strings
    | NUMBER
    | (tuple | group | genexp)
    | (list | listcomp)
    | (dict | set | dictcomp | setcomp)
    | '...'

group:
    | '(' (yield_expr | named_expression) ')'

# Lambda functions
# ----------------

lambdef:
    | 'lambda' [lambda_params] ':' expression

lambda_params:
    | lambda_parameters

# lambda_parameters etc. duplicates parameters but without annotations
# or type comments, and if there's no comma after a parameter, we expect
# a colon, not a close parenthesis.  (For more, see parameters above.)
#
lambda_parameters:
    | lambda_slash_no_default lambda_param_no_default* lambda_param_with_default* [lambda_
↪star_etc]
    | lambda_slash_with_default lambda_param_with_default* [lambda_star_etc]
    | lambda_param_no_default+ lambda_param_with_default* [lambda_star_etc]
    | lambda_param_with_default+ [lambda_star_etc]
    | lambda_star_etc
```

(续下页)

(接上页)

```
lambda_slash_no_default:
    | lambda_param_no_default+ '/' ','
    | lambda_param_no_default+ '/' &':'

lambda_slash_with_default:
    | lambda_param_no_default* lambda_param_with_default+ '/' ','
    | lambda_param_no_default* lambda_param_with_default+ '/' &':'

lambda_star_etc:
    | '*' lambda_param_no_default lambda_param_maybe_default* [lambda_kwds]
    | '*' ',' lambda_param_maybe_default+ [lambda_kwds]
    | lambda_kwds

lambda_kwds:
    | '**' lambda_param_no_default

lambda_param_no_default:
    | lambda_param ','
    | lambda_param &':'
lambda_param_with_default:
    | lambda_param default ','
    | lambda_param default &':'
lambda_param_maybe_default:
    | lambda_param default? ','
    | lambda_param default? &':'
lambda_param: NAME

# LITERALS
# ========

fstring_middle:
    | fstring_replacement_field
    | FSTRING_MIDDLE
fstring_replacement_field:
    | '{' annotated_rhs '='? [fstring_conversion] [fstring_full_format_spec] '}'
fstring_conversion:
    | "!" NAME
fstring_full_format_spec:
    | ':' fstring_format_spec*
fstring_format_spec:
    | FSTRING_MIDDLE
    | fstring_replacement_field
fstring:
    | FSTRING_START fstring_middle* FSTRING_END

string: STRING
strings: (fstring|string)+

list:
    | '[' [star_named_expressions] ']'

tuple:
    | '(' [star_named_expression ',' [star_named_expressions]  ] ')'

set: '{' star_named_expressions '}'

# Dicts
# -----

dict:
```

(续下页)

(接上页)

```
    | '{' [double_starred_kvpairs] '}'

double_starred_kvpairs: ','.double_starred_kvpair+ [',']

double_starred_kvpair:
    | '**' bitwise_or
    | kvpair

kvpair: expression ':' expression

# Comprehensions & Generators
# ---------------------------

for_if_clauses:
    | for_if_clause+

for_if_clause:
    | 'async' 'for' star_targets 'in' ~ disjunction ('if' disjunction )*
    | 'for' star_targets 'in' ~ disjunction ('if' disjunction )*

listcomp:
    | '[' named_expression for_if_clauses ']'

setcomp:
    | '{' named_expression for_if_clauses '}'

genexp:
    | '(' ( assignment_expression | expression !':=') for_if_clauses ')'

dictcomp:
    | '{' kvpair for_if_clauses '}'

# FUNCTION CALL ARGUMENTS
# =======================

arguments:
    | args [','] &')'

args:
    | ','.(starred_expression | ( assignment_expression | expression !':=') !'=')+ [',' ␣
↪kwargs ]
    | kwargs

kwargs:
    | ','.kwarg_or_starred+ ',' ','.kwarg_or_double_starred+
    | ','.kwarg_or_starred+
    | ','.kwarg_or_double_starred+

starred_expression:
    | '*' expression

kwarg_or_starred:
    | NAME '=' expression
    | starred_expression

kwarg_or_double_starred:
    | NAME '=' expression
    | '**' expression

# ASSIGNMENT TARGETS
# ==================
```

(续下页)

(接上页)

```
# Generic targets
# ---------------

# NOTE: star_targets may contain *bitwise_or, targets may not.
star_targets:
    | star_target !','
    | star_target (',' star_target )* [',']

star_targets_list_seq: ','.star_target+ [',']

star_targets_tuple_seq:
    | star_target (',' star_target )+ [',']
    | star_target ','

star_target:
    | '*' (!'*' star_target)
    | target_with_star_atom

target_with_star_atom:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | star_atom

star_atom:
    | NAME
    | '(' target_with_star_atom ')'
    | '(' [star_targets_tuple_seq] ')'
    | '[' [star_targets_list_seq] ']'

single_target:
    | single_subscript_attribute_target
    | NAME
    | '(' single_target ')'

single_subscript_attribute_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead

t_primary:
    | t_primary '.' NAME &t_lookahead
    | t_primary '[' slices ']' &t_lookahead
    | t_primary genexp &t_lookahead
    | t_primary '(' [arguments] ')' &t_lookahead
    | atom &t_lookahead

t_lookahead: '(' | '[' | '.'

# Targets for del statements
# --------------------------

del_targets: ','.del_target+ [',']

del_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | del_t_atom

del_t_atom:
    | NAME
    | '(' del_target ')'
```

(续下页)

(接上页)

```
    | '(' [del_targets] ')'
    | '[' [del_targets] ']'

# TYPING ELEMENTS
# ---------------

# type_expressions allow */** but ignore them
type_expressions:
    | ','.expression+ ',' '*' expression ',' '**' expression
    | ','.expression+ ',' '*' expression
    | ','.expression+ ',' '**' expression
    | '*' expression ',' '**' expression
    | '*' expression
    | '**' expression
    | ','.expression+

func_type_comment:
    | NEWLINE TYPE_COMMENT &(NEWLINE INDENT)   # Must be followed by indented block
    | TYPE_COMMENT

# ========================= END OF THE GRAMMAR ===========================



# ========================= START OF INVALID RULES ========================
```

APPENDIX A

术语对照表

>>>

interactive shell 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。

...

具有以下含义：

- *interactive* shell 中输入特殊代码时默认的 Python 提示符，特殊代码包括缩进的代码块，左右成对分隔符（圆括号、方括号、花括号或三重引号等）之内，或是在指定一个装饰器之后。
- `Ellipsis` 内置常量。

abstract base class -- 抽象基类

抽象基类简称 ABC，是对*duck-typing* 的补充，它提供了一种定义接口的新方式，相比之下其他技巧例如 `hasattr()` 显得过于笨拙或有微妙错误（例如使用魔术方法）。ABC 引入了虚拟子类，这种类并非继承自其他类，但却仍能被 `isinstance()` 和 `issubclass()` 所认可；详见 `abc` 模块文档。Python 自带许多内置的 ABC 用于实现数据结构（在 `collections.abc` 模块中）、数字（在 `numbers` 模块中）、流（在 `io` 模块中）、导入查找器和加载器（在 `importlib.abc` 模块中）。你可以使用 `abc` 模块来创建自己的 ABC。

annotation -- 标注

关联到某个变量、类属性、函数形参或返回值的标签，被约定作为类型注解 来使用。

局部变量的标注在运行时不可访问，但全局变量、类属性和函数的标注会分别存放模块、类和函数的 `__annotations__` 特殊属性中。

参见*variable annotation*, *function annotation*, **PEP 484** 和 **PEP 526**，对此功能均有介绍。另请参见 annotations-howto 了解使用标注的最佳实践。

argument -- 参数

在调用函数时传给*function* （或*method* ）的值。参数分为两种：

- *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。

另参见*parameter* 术语表条目，常见问题中 参数与形参的区别以及 **PEP 362**。

asynchronous context manager -- 异步上下文管理器
此种对象通过定义`__aenter__()` 和`__aexit__()` 方法来对`async with` 语句中的环境进行控制。由 **PEP 492** 引入。

asynchronous generator -- 异步生成器
返回值为*asynchronous generator iterator* 的函数。它与使用`async def` 定义的协程函数很相似，不同之处在于它包含`yield` 表达式以产生一系列可在`async for` 循环中使用的值。

此术语通常是指异步生成器函数，但在某些情况下则可能是指 异步生成器迭代器。如果需要清楚表达具体含义，请使用全称以避免歧义。

一个异步生成器函数可能包含`await` 表达式或者`async for` 以及`async with` 语句。

asynchronous generator iterator -- 异步生成器迭代器
asynchronous generator 函数所创建的对象。

此对象属于*asynchronous iterator*，当使用`__anext__()` 方法调用时会返回一个可等待对象来执行异步生成器函数的函数体直到下一个`yield` 表达式。

每个`yield` 会临时暂停处理，记住当前位置执行状态（包括局部变量和挂起的 try 语句）。当该 异步生成器迭代器通过`__anext__()` 所返回的其他可等待对象有效恢复时，它会从离开位置继续执行。参见 **PEP 492** 和 **PEP 525**。

asynchronous iterable -- 异步可迭代对象
一个可以在`async for` 语句中使用的对象。必须通过它的`__aiter__()` 方法返回一个*asynchronous iterator*。由 **PEP 492** 引入。

asynchronous iterator -- 异步迭代器
一个实现了`__aiter__()` 和`__anext__()` 方法的对象。`__anext__()` 必须返回一个*awaitable* 对象。`async for` 会处理异步迭代器的`__anext__()` 方法所返回的可等待对象直到其引发一个`StopAsyncIteration` 异常。由 **PEP 492** 引入。

attribute -- 属性
关联到一个对象的值，通常使用点号表达式按名称来引用。举例来说，如果对象 *o* 具有属性 *a* 则可以用 *o.a* 来引用它。

如果对象允许，将未被定义为标识符和关键字 的非标识名称用作一个对象的属性也是可以的，例如使用 `setattr()`。这样的属性将无法使用点号表达式来访问，而是必须通过 `getattr()` 来获取。

awaitable -- 可等待对象
一个可在`await` 表达式中使用的对象。可以是*coroutine* 或是具有`__await__()` 方法的对象。参见 **PEP 492**。

BDFL
“终身仁慈独裁者”的英文缩写，即 Guido van Rossum，Python 的创造者。

binary file -- 二进制文件
file object 能够读写字节型对象。二进制文件的例子包括以二进制模式 (`'rb'`, `'wb'` 或 `'rb+'`) 打开的文件、`sys.stdin.buffer`、`sys.stdout.buffer` 以及 `io.BytesIO` 和 `gzip.GzipFile` 的实例。

另请参见*text file* 了解能够读写 `str` 对象的文件对象。

borrowed reference -- 借入引用
在 Python 的 C API 中，借用引用是指一种对象引用，使用该对象的代码并不持有该引用。如果对象被销毁则它就会变成一个悬空指针。例如，垃圾回收器可以移除对象的最后一个*strong reference* 来销毁它。

推荐在*borrowed reference* 上调用 `Py_INCREF()` 以将其原地转换为*strong reference*，除非是当该对象无法在借入引用的最后一次使用之前被销毁。`Py_NewRef()` 函数可以被用来创建一个新的*strong reference*。

bytes-like object -- 字节型对象
支持 bufferobjects 并且能导出 C-*contiguous* 缓冲的对象。这包括所有 `bytes`、`bytearray` 和 `array.array` 对象，以及许多普通 `memoryview` 对象。字节型对象可在多种二进制数据操作中使用；这些操作包括压缩、保存为二进制文件以及通过套接字发送等。

某些操作需要可变的二进制数据。这种对象在文档中常被称为“可读写字节类对象”。可变缓冲对象的例子包括 `bytearray` 以及 `bytearray` 的 `memoryview`。其他操作要求二进制数据存放于不可变对象 (” 只读字节类对象”)；这种对象的例子包括 `bytes` 以及 `bytes` 对象的 `memoryview`。

bytecode -- 字节码
Python 源代码会被编译为字节码，即 CPython 解释器中表示 Python 程序的内部代码。字节码还会缓存在 `.pyc` 文件中，这样第二次执行同一文件时速度更快（可以免去将源码重新编译为字节码）。这种” 中间语言” 运行在根据字节码执行相应机器码的*virtual machine* 之上。请注意不同 Python 虚拟机上的字节码不一定通用，也不一定能在不同 Python 版本上兼容。

字节码指令列表可以在 dis 模块的文档中查看。

callable -- 可调用对象
可调用对象就是可以执行调用运算的对象，并可能附带一组参数 (参见*argument*)，使用以下语法:

```
callable(argument1, argument2, argumentN)
```

function，还可扩展到*method* 等，就属于可调用对象。实现了*__call__()* 方法的类的实例也属于可调用对象。

callback -- 回调
一个作为参数被传入以用以在未来的某个时刻被调用的子例程函数。

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。

class variable -- 类变量
在类中定义的变量，并且仅限在类的层级上修改 (而不是在类的实例中修改)。

closure variable -- 闭包变量
A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the *nonlocal* keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are *free variables*, but only `x` is a *closure variable*:

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

Due to the *codeobject.co_freevars* attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

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
This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a *context manager* via a `with` statement.
- The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.
- A `contextvars.Context` object. Also see *current context*.

context management protocol
The `__enter__()` and `__exit__()` methods called by the `with` statement. See **PEP 343**.

context manager -- 上下文管理器
An object which implements the *context management protocol* and controls the environment seen in a `with` statement. See **PEP 343**.

context variable -- 上下文变量
A variable whose value depends on which context is the *current context*. Values are accessed via `contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

contiguous -- 连续
一个缓冲如果是 *C 连续*或 *Fortran 连续*就会被认为是连续的。零维缓冲是 C 和 Fortran 连续的。在一维数组中，所有条目必须在内存中彼此相邻地排列，采用从零开始的递增索引顺序。在多维 C-连续数组中，当按内存地址排列时用最后一个索引访问条目时速度最快。但是在 Fortran 连续数组中则是用第一个索引最快。

coroutine -- 协程
协程是子例程的更一般形式。子例程可以在某一点进入并在另一点退出。协程则可以在许多不同的点上进入、退出和恢复。它们可通过`async def` 语句来实现。参见 **PEP 492**。

coroutine function -- 协程函数
返回一个*coroutine* 对象的函数。协程函数可通过`async def` 语句来定义，并可能包含`await`、`async for` 和`async with` 关键字。这些特性是由 **PEP 492** 引入的。

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。

current context
The *context* (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

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(arg):
    ...
f = staticmethod(f)

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

同样的概念也适用于类，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
任何定义了`__get__()`, `__set__()` 或`__delete__()` 方法的对象。当一个类属性为描述器时，它的特殊绑定行为就会在属性查找时被触发。通常情况下，使用 *a.b* 来获取、设置或删除一个属性时

会在 *a* 类的字典中查找名称为 *b* 的对象，但如果 *b* 是一个描述器，则会调用对应的描述器方法。理解描述器的概念是更深层次理解 Python 的关键，因为这是许多重要特性的基础，包括函数、方法、特征属性、类方法、静态方法以及对超类的引用等等。

有关描述器的方法的更多信息，请参阅实现描述器 或 描述器使用指南。

dictionary -- 字典
一个关联数组，其中的任意键都映射到相应的值。键可以是任何具有*__hash__()* 和*__eq__()* 方法的对象。在 Perl 中称为 hash。

dictionary comprehension -- 字典推导式
处理一个可迭代对象中的所有或部分元素并返回结果字典的一种紧凑写法。`results = {n: n ** 2 for n in range(10)}` 将生成一个由键 `n` 到值 `n ** 2` 的映射构成的字典。参见列表、集合与字典的显示。

dictionary view -- 字典视图
从 `dict.keys()`, `dict.values()` 和 `dict.items()` 返回的对象被称为字典视图。它们提供了字典条目的一个动态视图，这意味着当字典改变时，视图也会相应改变。要将字典视图强制转换为真正的列表，可使用 `list(dictview)`。参见 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。它在代码块被执行时将被忽略，但会被编译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，因此是存放对象的文档的规范位置。

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
可以求出某个值的语法单元。换句话说，一个表达式就是表达元素例如字面值、名称、属性访问、运算符或函数调用的汇总，它们最终都会返回一个值。与许多其他语言不同，并非所有语言构件都是表达式。还存在不能被用作表达式的*statement*，例如*while*。赋值也是属于语句而非表达式。

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。

f-string -- f-字符串
带有 `'f'` 或 `'F'` 前缀的字符串字面值通常被称为“f-字符串”即格式化字符串字面值 的简写。参见 **PEP 498**。

file object -- 文件对象
对外公开面向文件的 API（带有 `read()` 或 `write()` 等方法）以使用下层资源的对象。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、其他类型的存储或通信设备的访问（例如标准输入/输出、内存缓冲区、套接字、管道等）。文件对象也被称为 *文件型对象* 或 *流*。

实际上共有三种类别的文件对象: 原始二进制文件, 缓冲二进制文件 以及文本文件。它们的接口定义均在 `io` 模块中。创建文件对象的规范方式是使用 `open()` 函数。

file-like object -- 文件型对象
file object 的同义词。

filesystem encoding and error handler -- 文件系统编码格式与错误处理器
Python 用来从操作系统解码字节串和向操作系统编码 Unicode 的编码格式与错误处理器。

文件系统编码格式必须保证能成功解码长度在 128 以下的所有字节串。如果文件系统编码格式无法提供此保证，则 API 函数可能会引发 `UnicodeError`。

sys.getfilesystemencoding() 和 sys.getfilesystemencodeerrors() 函数可被用来获取文件系统编码格式与错误处理器。

filesystem encoding and error handler 是在 Python 启动时通过 PyConfig_Read() 函数来配置的：请参阅 PyConfig 的 filesystem_encoding 和 filesystem_errors 等成员。

另请参见*locale encoding*。

finder -- 查找器
一种会尝试查找被导入模块的*loader* 的对象。

存在两种类型的查找器: 元路径查找器 配合 sys.meta_path 使用，以及路径条目查找器 配合 sys.path_hooks 使用。

See 查找器和加载器 and importlib for much more detail.

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** 。

free threading -- 自由线程
一种线程模型，在同一个解释器内部的多个线程可以同时运行 Python 字节码。与此相对的是*global interpreter lock*，在同一时刻只允许一个线程执行 Python 字节码。参见 **PEP 703**。

free variable -- 自由变量
Formally, as defined in the *language execution model*, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the *codeobject.co_freevars* attribute, the term is also sometimes used as a synonym for *closure variable*.

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。

function annotation -- 函数标注
即针对函数形参或返回值的*annotation* 。

函数标注通常用于类型提示：例如以下函数预期接受两个 int 参数并预期返回一个 int 值:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

函数标注语法的详解见函数定义 一节。

参见*variable annotation* 和 **PEP 484**，其中描述了此功能。另请参阅 annotations-howto 以了解使用标的最佳实践。

__future__
future 语句, from __future__ import <feature> 指示编译器使用将在未来的 Python 发布版中成为标准的语法和语义来编译当前模块。__future__ 模块文档记录了可能的 *feature* 取值。通过导入此模块并对其变量求值，你可以看到每项新特性在何时被首次加入到该语言中以及它将（或已）在何时成为默认:

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

garbage collection -- 垃圾回收
释放不再被使用的内存空间的过程。Python 是通过引用计数和一个能够检测和打破循环引用的循环垃圾回收器来执行垃圾回收的。可以使用 gc 模块来控制垃圾回收器。

generator -- 生成器
返回一个*generator iterator* 的函数。它看起来很像普通函数，不同点在于其包含*yield* 表达式以便产生一系列值供给 for-循环使用或是通过 next() 函数逐一获取。

通常是指生成器函数，但在某些情况下也可能是指 生成器迭代器。如果需要清楚表达具体含义，请使用全称以避免歧义。

generator iterator -- 生成器迭代器
generator 函数所创建的对象。

每个`yield`会临时暂停处理，记住当前位置执行状态（包括局部变量和挂起的 try 语句）。当该 生成器迭代器恢复时，它会从离开位置继续执行（这与每次调用都从新开始的普通函数差别很大）。

generator expression -- 生成器表达式
返回一个*iterator* 的*expression*。它看起来很像普通表达式后带有定义了一个循环变量、范围的 `for` 子句，以及一个可选的 `if` 子句。以下复合表达式会为外层函数生成一系列值:

```
>>> sum(i*i for i in range(10))         # 平方值 0, 1, 4, ... 81 之和
285
```

generic function -- 泛型函数
为不同的类型实现相同操作的多个函数所组成的函数。在调用时会由调度算法来确定应该使用哪个实现。

另请参见*single dispatch* 术语表条目、`functools.singledispatch()` 装饰器以及 **PEP 443**。

generic type -- 泛型
可参数化的*type*；通常为 `list` 或 `dict` 这样的容器类。用于类型提示 和注解。

更多细节参见 泛型别名类型、**PEP 483**、**PEP 484**、**PEP 585** 和 `typing` 模块。

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。

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703**.

hash-based pyc -- 基于哈希的 pyc
使用对应源文件的哈希值而非最后修改时间来确定其有效性的字节码缓存文件。参见已缓存字节码的失效。

hashable -- 可哈希
一个对象如果具有在其生命期内绝不改变的哈希值（它需要有`__hash__()`方法），并可以同其他对象进行比较（它需要有`__eq__()`方法）就被称为 可哈希对象。可哈希对象必须具有相同的哈希值比较结果才会相等。

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。

大多数 Python 中的不可变内置对象都是可哈希的；可变容器（例如列表或字典）都不可哈希；不可变容器（例如元组和 frozenset）仅当它们的元素均为可哈希时才是可哈希的。用户定义类的实例对象默认是可哈希的。它们在比较时一定不相同（除非是与自己比较），它们的哈希值的生成是基于它们的 `id()`。

IDLE
Python 的集成开发与学习环境。idle 是 Python 标准发行版附带的基本编辑器和解释器环境。

immortal -- 永生对象
永生对象是在 **PEP 683** 中引入的 CPython 实现细节。

如果对象属于永生对象，则它的*reference count* 永远不会被修改，因而它在解释器运行期间永远不会被取消分配。例如，`True` 和 `None` 在 CPython 中都属于永生对象。

immutable -- 不可变对象
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。

import path -- 导入路径
由多个位置（或路径条目）组成的列表，会被模块的*path based finder* 用来查找导入目标。在导入时，此位置列表通常来自 `sys.path`，但对次级包来说也可能来自上级包的 `__path__` 属性。

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，这意味着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主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这会非常方便（记住 `help(x)` 函数）。有关交互模式的详情，参见 tut-interac。

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*。

interpreter shutdown -- 解释器关闭
当被要求关闭时，Python 解释器将进入一个特殊运行阶段并逐步释放所有已分配资源，例如模块和各种关键内部结构等。它还会多次调用垃圾回收器。这会触发用户定义析构器或弱引用回调中的代码执行。在关闭阶段执行的代码可能会遇到各种异常，因为其所依赖的资源已不再有效（常见的例子有库模块或警告机制等）。

解释器需要关闭的主要原因有 `__main__` 模块或所运行的脚本已完成执行。

iterable -- 可迭代对象
一种能够逐个返回其成员项的对象。可迭代对象的例子包括所有序列类型（如 `list`, `str` 和 `tuple` 等）以及某些非序列类型如 `dict`, 文件对象 以及任何定义了 `__iter__()` 方法或实现了*sequence* 语义的*__getitem__()* 方法的自定义类的对象。

可迭代对象可被用于*for* 循环以及许多其他需要一个序列的地方 (`zip()`, `map()`, ...)。当一个可迭代对象作为参数被传给内置函数 `iter()` 时，它会返回该对象的迭代器。这种迭代器适用于对值集合的一次性遍历。在使用可迭代对象时，你通常不需要调用 `iter()` 或者自己处理迭代器对象。*for* 语句会自动为你处理那些操作，创建一个临时的未命名变量用来在循环期间保存迭代器。参见*iterator*, *sequence* 和*generator*。

iterator -- 迭代器
用来表示一连串数据流的对象。重复调用迭代器的 `__next__()` 方法 (或将其传给内置函数 `next()`) 将逐个返回流中的项。当没有数据可用时则将引发 `StopIteration` 异常。到这时迭代器对象中的数据项已耗尽，继续调用其 `__next__()` 方法只会再次引发 `StopIteration`。迭代器必须具有 `__iter__()` 方法用来返回该迭代器对象自身，因此迭代器必定也是可迭代对象，可被用于其他可迭代对象适用的大部分场合。一个显著的例外是那些会多次重复访问迭代项的代码。容器对象 (例如 `list`) 在你每次将其传入 `iter()` 函数或是在*for* 循环中使用时都会产生一个新的迭代器。如果在此情况下你尝试用迭代器则会返回在之前迭代过程中被耗尽的同一迭代器对象，使其看起来就像是一个空容器。

更多信息可查看 typeiter。

CPython 实现细节： CPython 没有强制推行迭代器定义 `__iter__()` 的要求。还要注意的是自由线程 CPython 并不保证迭代器操作的线程安全性。

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。

Python 中有许多工具都允许用键函数来控制元素的排位或分组方式。其中包括 `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` 以及 `itertools.groupby()`。

有多种方式可以创建一个键函数。例如，`str.lower()` 方法可以用作忽略大小写排序的键函数。或者，键函数也可通过 *lambda* 表达式来创建例如 `lambda r: (r[0], r[2])`。此外，`operator.attrgetter()`, `operator.itemgetter()` 和 `operator.methodcaller()` 是键函数的三个构造器。请查看 排序指引来获取创建和使用键函数的示例。

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

list comprehension -- 列表推导式
处理一个序列中的所有或部分元素并返回结果列表的一种紧凑写法。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 将生成一个 0 到 255 范围内的十六进制偶数对应字符串（0x..）的列表。其中 *if* 子句是可选的，如果省略则 `range(256)` 中的所有元素都会被处理。

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 also:

- 查找器和加载器
- `importlib.abc.Loader`
- **PEP 302**

locale encoding -- 语言区域编码格式
在 Unix 上，它是 LC_CTYPE 语言区域的编码格式。它可以通过 `locale.setlocale(locale.LC_CTYPE, new_locale)` 来设置。

在 Windows 上，它是 ANSI 代码页 (如: `"cp1252"`)。

在 Android 和 VxWorks 上，Python 使用 `"utf-8"` 作为语言区域编码格式。

`locale.getencoding()` 可被用来获取语言区域编码格式。

另请参阅*filesystem encoding and error handler*。

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

mapping -- 映射
一种支持任意键查找并实现了 `collections.abc.Mapping` 或 `collections.abc.MutableMapping` 抽象基类所规定方法的容器对象。此类对象的例子包括 `dict`, `collections.defaultdict`, `collections.OrderedDict` 以及 `collections.Counter`。

meta path finder -- 元路径查找器
`sys.meta_path` 的搜索所返回的*finder*。元路径查找器与*path entry finders* 存在关联但并不相同。

请查看 `importlib.abc.MetaPathFinder` 了解元路径查找器所实现的方法。

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。

更多详情参见元类。

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基类的顺序。请参阅 python_2.3_mro 了解自 2.3 发布版起 Python 解释器所使用算法的详情。

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*。

module spec -- 模块规格
一个命名空间，其中包含用于加载模块的相关导入信息。是 `importlib.machinery.ModuleSpec` 的实例。

See also 模块规格说明.

MRO
参见*method resolution order*。

mutable -- 可变对象
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
术语“具名元组”可用于任何继承自元组，并且其中的可索引元素还能使用名称属性来访问的类型或类。这样的类型或类还可能拥有其他特性。

有些内置类型属于具名元组，包括 `time.localtime()` 和 `os.stat()` 的返回值。另一个例子是 `sys.float_info`:

```
>>> sys.float_info[1]                   # 索引访问
1024
>>> sys.float_info.max_exp              # 命名字段访问
1024
>>> isinstance(sys.float_info, tuple)   # 属于元组
True
```

有些具名元组是内置类型（比如上面的例子）。此外，具名元组还可通过常规类定义从 `tuple` 继承并定义指定名称的字段的方式来创建。这样的类可以手工编号，或者也可以通过继承 `typing.NamedTuple`，或者使用工厂函数 `collections.namedtuple()` 来创建。后一种方式还会添加一些手工编写或内置的具名元组所没有的额外方法。

namespace -- 命名空间
命名空间是存放变量的场所。命名空间有局部、全局和内置的，还有对象中的嵌套命名空间（在方法之内）。命名空间通过防止命名冲突来支持模块化。例如，函数 `builtins.open` 与 `os.open()` 可通过各自的命名空间来区分。命名空间还通过明确哪个模块实现那个函数来帮助提高可读性和可维护性。例如，`random.seed()` 或 `itertools.islice()` 这种写法明确了这些函数是由 `random` 与 `itertools` 模块分别实现的。

namespace package -- 命名空间包
PEP 420 所引入的一种仅被用作子包的容器的*package*，命名空间包可以没有实体表示物，其描述方式与*regular package* 不同，因为它们没有 `__init__.py` 文件。

另可参见*module*。

nested scope -- 嵌套作用域
在一个定义范围内引用变量的能力。例如，在另一函数之内定义的函数可以引用前者的变量。请注意嵌套作用域默认只对引用有效而对赋值无效。局部变量的读写都受限于最内层作用域。类似的，全局变量的读写则作用于全局命名空间。通过*nonlocal* 关键字可允许写入外层作用域。

new-style class -- 新式类
对目前已被应用于所有类对象的类形式的旧称谓。在较早的 Python 版本中，只有新式类能够使用 Python 新增的更灵活我，如*__slots__*、描述器、特征属性、*__getattribute__()*、类方法和静态方法等。

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。

optimized scope -- 已优化的作用域
当代码被编译时编译器已充分知晓目标局部变量名称的作用域，这允许对这些名称的读写进行优化。针对函数、生成器、协程、推导式和生成器表达式的局部命名空间都是这样的已优化作用域。注意：大部分解释器优化将应用于所有作用域，只有那些依赖于已知的局部和非局部变量名称集合的优化会仅限于已优化的作用域。

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，包是具有 `__path__` 属性的 Python 模块。

另参见*regular package* 和*namespace package*。

parameter -- 形参
function（或方法）定义中的命名实体，它指定函数可以接受的一个*argument*（或在某些情况下，多个实参）。有五种形参：

- *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，指定一个只能通过位置传入的参数。仅限位置形参可通过在函数定义的形参列表中它们之后包含一个 `/` 字符来定义，例如下面的 *posonly1* 和 *posonly2*:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*：仅限关键字，指定一个只能通过关键字传入的参数。仅限关键字形参可通过在函数定义的形参列表中包含单个可变位置形参或者在多个可变位置形参之前放一个 `*` 来定义，例如下面的 *kw_only1* 和 *kw_only2*:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *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。

另参见*argument* 术语表条目、参数与形参的区别中的常见问题、`inspect.Parameter` 类、*函数定义* 一节以及 **PEP 362**。

path entry -- 路径入口
import path 中的一个单独位置，会被*path based finder* 用来查找要导入的模块。

path entry finder -- 路径入口查找器
任一可调用对象使用 `sys.path_hooks` (即*path entry hook*) 返回的*finder*，此种对象能通过*path entry* 来定位模块。

请参看 `importlib.abc.PathEntryFinder` 以了解路径入口查找器所实现的各个方法。

path entry hook -- 路径入口钩子
一种可调用对象，它在知道如何查找特定*path entry* 中的模块的情况下能够使用 `sys.path_hooks` 列表返回一个*path entry finder*。

path based finder -- 基于路径的查找器
默认的一种*元路径查找器*，可在一个*import path* 中查找模块。

path-like object -- 路径类对象
代表一个文件系统路径的对象。路径类对象可以是一个表示路径的 `str` 或者 `bytes` 对象，还可以是一个实现了 `os.PathLike` 协议的对象。一个支持 `os.PathLike` 协议的对象可通过调用 `os.fspath()` 函数转换为 `str` 或者 `bytes` 类型的文件系统路径；`os.fsdecode()` 和 `os.fsencode()` 可被分别用来确保获得 `str` 或 `bytes` 类型的结果。此对象是由 **PEP 519** 引入的。

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**。

portion -- 部分
构成一个命名空间包的单个目录内文件集合（也可能存放于一个 zip 文件内），具体定义见 **PEP 420**。

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

provisional API -- 暂定 API
暂定 API 是指被有意排除在标准库的向后兼容性保证之外的应用编程接口。虽然此类接口通常不会再有重大改变，但只要其被标记为暂定，就可能在核心开发者确定有必要的情况下进行向后不兼容的更改（甚至包括移除该接口）。此种更改并不会随意进行 -- 仅在 API 被加入之前未考虑到的严重基础性缺陷被发现时才可能会这样做。

即便是对暂定 API 来说，向后不兼容的更改也会被视为“最后的解决方案”——任何问题被确认时都会尽可能先尝试找到一种向后兼容的解决方案。

这种处理过程允许标准库持续不断地演进，不至于被有问题的长期性设计缺陷所困。详情见 **PEP 411**。

provisional package -- 暂定包
参见*provisional API*。

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)
```

qualified name -- 限定名称
一个以点号分隔的名称，显示从模块的全局作用域到该模块中定义的某个类、函数或方法的“路径”，相关定义见 **PEP 3155**。对于最高层级的函数和类，限定名称与对象名称一致:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

当被用于引用模块时，完整限定名称意为标示该模块的以点号分隔的整个路径，其中包含其所有的父包，例如 `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count -- 引用计数
指向某个对象的引用的数量。当一个对象的引用计数降为零时，它就会被释放。特殊的*immortal* 对象具有永远不会被修改的引用计数，因此这种对象永远不会被释放。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。程序员可以调用 `sys.getrefcount()` 函数来返回特定对象的引用计数。

regular package -- 常规包
传统型的*package*，例如包含有一个 `__init__.py` 文件的目录。

另参见*namespace package*。

REPL
“读取-求值-打印循环” read-eval-print loop 的缩写，*interactive* 解释器 shell 的另一个名字。

__slots__
一种写在类内部的声明，通过预先声明实例属性等对象并移除实例字典来节省内存。虽然这种技巧很流行，但想要用好却并不容易，最好是只保留在少数情况下采用，例如极耗内存的应用程序，并且其中包含大量实例。

sequence -- 序列
一种*iterable*，它支持通过*__getitem__()* 特殊方法来使用整数索引进行高效的元素访问，并定义了一个返回序列长度的*__len__()* 方法。内置序列类型有 `list`, `str`, `tuple` 和 `bytes` 等。请注意虽然 `dict` 也支持*__getitem__()* 和 `__len__()`，但它被归类为映射而非序列，因为它使用任意的*hashable* 键而不是整数来查找元素。

`collections.abc.Sequence` 抽象基类定义了一个更丰富的接口，它在*__getitem__()* 和*__len__()* 之外，还添加了 `count()`, `index()`, *__contains__()* 和*__reversed__()*。实现此扩展接口的类型可以使用 `register()` 来显式地注册。要获取有关通用序列方法的更多文档，请参阅通用序列操作。

set comprehension -- 集合推导式
处理一个可迭代对象中的所有或部分元素并返回结果集合的一种紧凑写法。`results = {c for c in 'abracadabra' if c not in 'abc'}` 将生成字符串集合 `{'r', 'd'}`。参见列表、集合与字典的显示。

single dispatch -- 单分派
一种*generic function* 分派形式，其实现是基于单个参数的类型来选择的。

slice -- 切片
通常只包含了特定*sequence* 的一部分的对象。切片是通过使用下标标记来创建的，在 `[]` 中给出几

个以冒号分隔的数字，例如 `variable_name[1:3:5]`。方括号（下标）标记在内部使用 `slice` 对象。

软弃用
A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

参见 PEP 387: Soft Deprecation。

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*。

static type checker -- 静态类型检查器
读取 Python 代码并进行分析，以查找问题例如拼写错误的外部工具。另请参阅类型提示 以及 `typing` 模块。

strong reference -- 强引用
在 Python 的 C API 中，强引用是指为持有引用的代码所拥有的对象的引用。在创建引用时可通过调用 `Py_INCREF()` 来获取强引用而在删除引用时可通过 `Py_DECREF()` 来释放它。

`Py_NewRef()` 函数可被用于创建一个对象的强引用。通常，必须在退出某个强引用的作用域时在该强引用上调用 `Py_DECREF()` 函数，以避免引用的泄漏。

另请参阅*borrowed reference*。

text encoding -- 文本编码格式
在 Python 中，一个字符串是一串 Unicode 代码点（范围为 `U+0000--U+10FFFF`）。为了存储或传输一个字符串，它需要被序列化为一串字节。

将一个字符串序列化为一个字节序列被称为“编码”，而从字节序列中重新创建字符串被称为“解码”。

有各种不同的文本序列化 编码器，它们被统称为” 文本编码格式”。

text file -- 文本文件
一种能够读写 `str` 对象的*file object*。通常一个文本文件实际是访问一个面向字节的数据流并自动处理*text encoding*。文本文件的例子包括以文本模式（`'r'` 或 `'w'`）打开的文件、`sys.stdin`、`sys.stdout` 以及 `io.StringIO` 的实例。

另请参看*binary file* 了解能够读写字节型对象 的文件对象。

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
Python 对象的类型决定它属于什么种类；每个对象都具有特定的类型。对象的类型可通过其*__class__* 属性来访问或是用 `type(obj)` 来获取。

type alias -- 类型别名
一个类型的同义词，创建方式是把类型赋值给特定的标识符。

类型别名的作用是简化类型注解。例如:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

可以这样提高可读性:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

参见 typing 和 **PEP 484**，其中有对此功能的详细描述。

type hint -- 类型注解
annotation 为变量、类属性、函数的形参或返回值指定预期的类型。

类型提示是可选的而不是 Python 的强制要求，但它们对静态类型检查器 很有用处。它们还能协助 IDE 实现代码补全与重构。

全局变量、类属性和函数的类型注解可以使用 typing.get_type_hints() 来访问，但局部变量则不可以。

参见 typing 和 **PEP 484**，其中有对此功能的详细描述。

universal newlines -- 通用换行
一种解读文本流的方式，将以下所有符号都识别为行结束标志：Unix 的行结束约定 '\n'、Windows 的约定 '\r\n' 以及旧版 Macintosh 的约定 '\r'。参见 **PEP 278** 和 **PEP 3116** 和 bytes.splitlines() 了解更多用法说明。

variable annotation -- 变量标注
对变量或类属性的*annotation*。

在标注变量或类属性时，还可选择为其赋值:

```
class C:
    field: 'annotation'
```

变量标注通常被用作类型提示：例如以下变量预期接受 int 类型的值:

```
count: int = 0
```

变量标注语法的详细解释见带标注的赋值语句 一节。

参见*function annotation*, **PEP 484** 和 **PEP 526**，其中描述了此功能。另请参阅 annotations-howto 以了解使用标注的最佳实践。

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。

另参见 venv。

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

这些文档是用 Sphinx 从 reStructuredText 源生成的，*Sphinx* 是一个专为处理 Python 文档而编写的文档生成器。

本文档及其工具链之开发，皆在于志愿者之努力，亦恰如 Python 本身。如果您想为此作出贡献，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.，原始 Python 文档工具集之创造者，众多文档之作者；
- 用于创建 reStructuredText 和 Docutils 套件的 Docutils 项目；
- Fredrik Lundh 的 Alternative Python Reference 项目，为 Sphinx 提供许多好的点子。

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 源代码分发的 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 公司；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 公司现在是 Python 软件基金会的赞助成员。

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

备注

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

Python 软件和文档的使用许可均基于*PSF 许可协议*。

从 Python 3.8.6 开始，文档中的示例、操作指导和其他代码采用的是 PSF 许可协议和零条款 *BSD* 许可 的双重使用许可。

某些包含在 Python 中的软件基于不同的许可。这些许可会与相应许可之下的代码一同列出。有关这些许可的不完整列表请参阅收录软件的许可与鸣谢。

C.2.1 用于 PYTHON 3.13.0 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
   3.13.0 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 3.13.0 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights
   Reserved" are retained in Python 3.13.0 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 3.13.0 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
   3.13.0.

4. PSF is making Python 3.13.0 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 3.13.0 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.13.0
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.13.0, 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 3.13.0, 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
   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
```

(续下页)

(接上页)

```
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

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

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
```

(续下页)

(接上页)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
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 收录软件的许可与鸣谢

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

作为 `random` 模块下层的 `_random` C 扩展包括基于从 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:

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

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

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

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

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

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

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

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
```

(续下页)

(接上页)

```
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` 使用了 `getaddrinfo()` 和 `getnameinfo()` WIDE 项目的不同源文件中: https://www.wide.ad.jp/

```
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
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 异步套接字服务

`test.support.asynchat` 和 `test.support.asyncore` 模块包含以下说明。:

```
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
```

(续下页)

(接上页)

```
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.3.4 Cookie 管理

`http.cookies` 模块包含以下声明:

```
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.3.5 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,
```

(续下页)

(接上页)

```
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode 与 UUdecode 函数

uu 编解码器包含以下声明:

```
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
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML 远程过程调用

xmlrpc.client 模块包含以下声明:

```
    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
```

(续下页)

(接上页)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

`test.test_epoll` 模块包含以下说明:

```
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
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.3.9 Select kqueue

`select` 模块关于 kqueue 的接口包含以下声明:

```
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.3.10 SipHash24

Python/pyhash.c 文件包含 Marek Majkowski' 对 Dan Bernstein 的 SipHash24 算法的实现。它包含以下声明:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

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.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 和 dtoa

Python/dtoa.c 文件提供了 C 函数 dtoa 和 strtod，用于 C 双精度数值和字符串之间的转换，它派生自由 David M. Gay 编写的同名文件。目前该文件可在 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c 访问。在 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
 * 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.3.12 OpenSSL

如果操作系统有支持则 hashlib, posix 和 ssl 会使用 OpenSSL 库来提升性能。此外，Python 的 Windows 和 macOS 安装程序可能会包括 OpenSSL 库的副本，所以我们也在此包括一份 OpenSSL 许可证的副本。对于 OpenSSL 3.0 发布版，及其后续衍生版本，均使用 Apache License v2:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/
```

(续下页)

(接上页)

```
TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.
```

(续下页)

(接上页)

```
2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
```

(续下页)

(接上页)

```
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

`pyexpat` 扩展是使用所包括的 expat 源副本来构建的，除非配置了 `--with-system-expat`:

```
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
```

(续下页)

(接上页)

```
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.3.14 libffi

作为 `ctypes` 模块下层的 `_ctypes` C 扩展是使用包括了 libffi 源的副本构建的，除非构建时配置了 `--with-system-libffi`:

```
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.3.15 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) 1995-2011 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
```

(续下页)

(接上页)

```
   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
```

C.3.16 cfuhash

`tracemalloc` 使用的哈希表的实现基于 cfuhash 项目:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

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

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

  * 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.

  * Neither the name of the author 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 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.
```

C.3.17 libmpdec

作为 `decimal` 模块下层的 `_decimal` C 扩展是使用包括了 libmpdec 库的副本构建的，除非构建时配置了 `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

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

(续下页)

(接上页)

```
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.3.18 W3C C14N 测试套件

`test` 包中的 C14N 2.0 测试集 (`Lib/test/xmltestdata/c14n-20/`) 提取自 W3C 网站 https://www.w3.org/TR/xml-c14n2-testcases/ 并根据 3 条款版 BSD 许可证发行:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

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

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work 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.
```

C.3.19 mimalloc

MIT 许可证:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy
```

(续下页)

(接上页)

```
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.3.20 asyncio

`asyncio` 模块的某些部分来自 uvloop 0.16，它是基于 MIT 许可证发行的:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

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.3.21 Global Unbounded Sequences (GUS)

文件 `Python/qsbr.c` 改编自 subr_smr.c 中 FreeBSD 的"Global Unbounded Sequences" 安全内存回收方案。该文件是基于 2 条款 BSD 许可证分发的:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

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 unmodified, 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 ``AS IS'' AND ANY EXPRESS OR
```

(续下页)

(接上页)

```
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 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.
```

APPENDIX D

版权所有

Python 与这份文档：

有关完整的许可证和许可信息，请参见历史和许可证。

索引

非字母

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z