Python语言参考手

极客学院出版

前言

本手册翻译自 Python 官方发行的《Python Language Reference》(https://docs.python.org/3/reference/index.html)，适用于 Python 3.4.3 版本。

Python 是一种解释性的、面向对象的、具有动态语义的高级程序设计语言。它内建高级数据结构，配以动态类型和动态捆绑，使其在快速应用开发中非常有利，就像脚本或粘合语言一样将已存在的构件连接在一起。Python 的简单性和句法的易学性使其代码具有优秀的可读牲，因此维护程序的成本得以大大降低。Python 具有模块和包的概念，以支持程序的模块化和代码重用。在主流平台上，Python 的解释器和大量标准库都可以免费地以源代码形式或可执行文件形式获得，并且可以自由发布。

本参考手册描述了该语言的语法和“核心语义”。手册本身是比较简洁的，但尽可能写得准确和完 整。那些非基本的内置对象类型、内置函数和模块的语义在《Python Standard Library》(https://docs.python.org/3/library/index.html#library-index) 中进行描述。对于语言的浅显介绍，可以看看《The Python Tutorial》(https://docs.python.org/3/tutorial/index.html#tutorial-index)。对于 C 和 C++ 程序员, 有两个文档可供参考：

- 《Extending and Embedding the Python Interpreter》(https://docs.python.org/3/extending/index.html#extending-index)，是对 Python 扩展模块设计的总体介绍，
- 《Python/C API Reference Manual》(https://docs.python.org/3/c-api/index.html#c-api-index)，则向 C/C++ 程序员们细致地描述了可以使用的接口。

> Python Language Reference 官网：https://docs.python.org/3/reference/index.html

更新日期	更新内容
2015-04-09	第一版发布 (Python V3.4.3)

目录

前言 1
第 1 章　介绍 3
第 2 章　词法分析 6
第 3 章　数据模型 20
第 4 章　执行模型 53
第 5 章　导入系统 57
第 6 章　表达式 76
第 7 章　简单语句 100
第 8 章　复合语句 115
第 9 章　顶层组件 128
第 10 章　完整的语法规范 131

1

介绍

本参考手册描述了Python编程语言，不能替代入门教程。

虽然我尝试尽可能的精确，除了语法和词法分析之外，仍然选择使用英语而不是形式化的规范的来描述。这应该使文档更具可读性，但可能存在歧义。因此，如果你来自火星，仅从这个文档试图重新实现Python，您可能需要猜一些东西，事实上你可能会最终实施一门不同的语言。另一方面，如果您正在使用Python，想了解某些细节精确的语言规则，你应该可以在这里找到他们。如果你想看到一个更正式的语言定义，也许你自己搞了，或者发明一个克隆机：-）。

添加太多的实现细节到语言的参考文档中是危险的事情 — 实现可能会改变，并且同一语言的不同实现可能以不同方式工作。另一方面，CPython是广泛使用的一个Python实现（虽然替代实现继续得到支持），这里有些细节还是被提及了， 特别是某种实现增加了限制时， 因此你可以在本文档中找到“实现注意”的标记。

每个Python实现都提供了大量的内建和标准模块，他们被记录在Python Standard Libray (https://docs.python.org/3/library/index.html#library-index) 中。一些与语言定义密切相关的内建模块也在这里被介绍了。

替代实现

虽然这是迄今为止最受欢迎的Python实现，不同的Python爱好者对一些替代的实现有特别感兴趣。

已知实现包括：

CPython
这是Python的原始和被维护最多的实现，用c语言编写，新的语言特性通常首先出现在这里。

Jython
Python的Java实现。这个实现可以作为Java应用程序的脚本语言，或可用于创建使用Java类库的应用。他也经常被用于创建Java库的测试。更多的信息可以在Jython的网站 (http://www.jython.org/) 上找到。

Python for .NET
这个实际上使用了CPython的实现，但它是一个托管的.NET应用程序，作为.NET库提供。由Brian Lloyd创建，更多信息见Python for .NET主页 (http://pythonnet.sourceforge.net/) 。

IronPython
另一个.NET实现的Python，与Python.NET不同，这是一个完整的Python实现，生成IL，并直接编译Python代码到.NET程序集。由Jython的原作者Jim Hugunin创建。更多信息见IronPython网站 (http://ironpython.net/)

PyPy
完全用Python实现的Python。他支持在其他实现里没有的几个高级功能，像支持无堆栈和即时编译。该项目的

目标之一是鼓励实验与语言本身，使其更轻松地修改该解释器（因为它是用Python写的），PyPy 项目主页 (http://pypy.org/) 上可用的其他信息。

每种实现在某种程度上的变化，从本手册记录的语言，或者介绍具体信息以外的标准Python文档。请参考详细的实现文档，以确定在使用的实现中还有什么是你想了解的。

记法

在描述词法和句法分析时候， 我们使用不甚严格的BNF， 通常是以下的定义方式:

name ::= lc_ letter (https://docs.python.org/3/reference/introduction.html#grammar-token-lc_letter) (lc_ letter (https://docs.python.org/3/reference/introduction.html#grammar-token-lc_letter) | "_ ")*
lc_letter ::= "a"..."z"

第一行说明 name 为 lc_letter 后跟随零个以上(包括零个) lc_letter 或下划线的序列。 lc_letter 是” a ”至” z ”中任意一个字符.(实际上， 这个” name ”的定义贯穿于本文档的整个词法和语法规则中)

每个规则以一个名字(为所定义的规则的名字)和一个冒号" ::= "为开始. 竖线(|)用于分隔可选项.这是记法中结合性最弱的符号.星号(*)意味着前一项的零次或多次的重复; 同样， 加号(+)意味着一次或多次的重复. 在方括号([])中的内容意味着它可以出现零次或一次(也就是说它是可选的).星号" * "和加号" + "与前面的项尽可能地紧密的结合， 小括号用于分组.字符串的字面值用引号括住.空白字符仅仅在分隔语言符号(token)时有用.通常规则被包含在一行之中， 有很多可选项的规则可能会被格式化成多行的形式， 后续行都以一个竖线开始。

在词法定义中(如上例)， 有两个习惯比较常用:以三个句点分隔的一对串字面值意味着在给定（包括）的ASCII字符范围内任选一个字符。在尖括号(<...>)中的短语给出了非正式的说明， 例如， 这用在了需要说 明”控制字符”记法的时候。

即使在句法和词法定义中使用的记法几乎相同， 但它们之间在含义上还是有着的很大不同: 词法定义是 在输入源的一个个字符上进行操作，而句法定义是在由词法分析所生成的语言符号流上进行操作。在下 节(”词法分析”)中使用的BNF都是词法定义， 以后的章节是句法定义。

2

词法分析

一个 Python 程序是由一个解析器读取的。解析器的输入是一个由词法分析器生成的符号流。本章介绍了词法分析器如何将文件分解为符号。

Python 将程序文本以 Unicode 代码点的方式读入；源文件的编码格式由编码声明给出，其默认值为 UTF-8，详见 PEP 3120 (http://www.python.org/dev/peps/pep-3120) 。源文件无法被解析时，会引发 SyntaxError (https://docs.python.org/3/library/exceptions.html#SyntaxError) 异常。

行结构

一个 Python 程序会被分成多个逻辑行。

逻辑行

逻辑行使用一个 NEWLINE 标记结尾。除非在语法中允许使用 NEWLINE (例如，复合语句中的声明语句之间)，否则声明语句不能跨多个逻辑行。逻辑行是由一个物理行，或多个通过显示或隐式行连接规则连接在一起的物理行组成。

物理行

物理行是一个以行尾序列结尾的字符序列。在源文件中，任何平台下标准的行终止序列都可以使用-在 Unix 平台上使用 ASCII LF(换行)符，在 Windows 平台上使用 ASCII 字符序列CR LF(回车加换行)，在老版本的 Macintosh 平台上使用 ASCII CR(回车)符。无论使用什么平台，所有这些形式都可以使用。

使用嵌入式Python开发时，需要使用标准 C 语言约定的换行符将源代码传递给 Python API（字符 \n 表示 ASCII LF，即行终止符)。

注释

注释以(`#`)字符开始(且此时该符号不是当前字符串的一部分)，以该物理行的结尾为结束。如果没有调用隐式行连接，那么注释即意味着逻辑行的结尾。注释会被语法分析所忽视，它们不会被作为程序代码处理。

编码声明

如果第一或第二行的 Python 脚本注释能够匹配正则表达式 `coding[=:]\s*([-\w.]+)` ,该注释会被当做编码声明处理；该表达式的第一部分定义了源代码文件的编码格式。此表达式的推荐形式是

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

该表达式也可以被 GNU Emacs 所识别，Bram Moolenaar’s VIM 识别的表达式形式为

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

如果未发现编码声明，则默认编码格式为 UTF-8。另外，如果此文件的第一个字节是 UTF-8 字节序标记（ `b'\xef\xbb\xbf'` ），则声明的编码为 UTF-8（微软的 notepad 支持这种编码格式）。

如果声明了一种编码格式，则编码格式名称必须可以被 Python 识别。所有的词法分析，包括字符串、注释以及标识符都需要使用这种编码格式。编码声明必须出现在其自己的行中。

显式行连接

两个或者多个物理行可以使用反斜线(`\`)合并为一个逻辑行，例如：当一个物理行以反斜线结束，且该反斜线不是字符串值或注释的一部分时，它就与下面的物理行合并构成一个逻辑行，并将反斜线及行结束符删除。例如：

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

以反斜线结尾的行不能在其后加注释，反斜线不能延续注释行。除了字符串文本，反斜线也不能延续语言符号(也就是说，其他不是字符串文本的语言符号不可以使用反斜线横跨多个物理行)。在字符串文本外的其他地方出现反斜线都是非法的。

隐式行连接

圆括号、 方括号或大括号中的表达式可以跨多个物理行，而无需使用反斜杠。例如：

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

可以在隐式行连接的末尾添加注释。接续行的缩进可以不考虑。并且允许出现空的接续行。在隐式的接续行中是不存在 NEWLINE 符号的。隐式的行连接在三重引用串（后述）中也是合法的，在那种情况下不能添加注释。

空行

一个只包含空格符、制表符、换页符和可能是注释的逻辑行可以忽略（也就是说没有 NEWLINE 符号产生）。在交互式输入语句时，空行的处理可能不同，其依赖于输入-计算-输出循环的实现方式。在标准的交互实现中，一个纯粹的空逻辑行（即不包含任何东西，甚至是空白和注释）可以结束多行语句。

缩进

位于逻辑行开始前的空白（空格和制表符）用于计算行的缩进层次，该层次可用于语句的分组。

制表符被（从左到右）替换为一至八个空格，这样直到包括替换部分的总字符数是八的倍数（这与 Unix 中使用的规则是相同的）。第一个非空字符前的空格总数决定了行的缩进。不能使用反斜线在多个物理行之间对缩进进行拆分；第一个反斜线之前的空白字符用于检测缩进。

在以下情形下缩进会被认为是不符合逻辑的：一个源文件中既包含制表符又包含空格，此时如果该文件的意义依赖于包含在空格符中的制表符时，就会触发 TabError (https://docs.python.org/3/library/exceptions.html#TabError) 异常。

跨平台兼容性注意：由于非 UNIX 平台上文本编辑器的特性，在单个源文件中混合使用空格符和制表符的缩进是不明智的。还需要注意的是，不同的平台可能会显式的限制最大缩进级别。

虽然在行首可能会出现换页符；但它在以上的缩进计算中会被忽略。出现在其他位置的换页符的作用是不确定的（比如，它可能将空格数重置为0）。

连续行的缩进层次用于生成 INDENT 及 DEDENT 语言符号，在此过程中使用了堆栈，如下所述。

文件的第一行未被读取之前，一个0被压入栈中；它以后也不会被弹出来。被压入栈中的数字都从栈底向栈顶增长，在每个逻辑行的开头处，将行的缩进层次与栈顶元素比较，如果相等则什么都不做。如果大于栈顶元素，则将其压入栈中并生成一个 INDENT 语言符号。如果小于栈顶元素，那么其应该是堆栈中已经存在的数字中的一个，堆栈中所有大于它的数都将被弹出，并且每个弹出的数字都会生成一个 DEDENT 语言符号。在文件的结尾，每个仍留在栈中且大于0的元素都会生成一个 DEDENT 语言符号。

这里有一个正确（尽管有点乱）缩进格式的 Python 代码的例子：

```
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
```

下面的例子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，前三种错误都是由解析器检测出来的，只有最后一个错误是由词法分析器发现的—— `return r` 的缩进层次与堆栈中弹出的数字不匹配。）

符号间的空白

除非位于逻辑行的行首或者字符串当中，空格符、制表符以及换页符都可以用于分割语言符号。只有当两个符号串接在一起可能会被解释为不同的符号时，才会在两个符号之间增加空格（例如： ab 是一个符号，但 a b 却是两个符号）。

其他语言符号

除了 NEWLINE， INDENT 和 DEDENT，还有以下几类语言符号：标识符、关键字、文本、运算符及分隔符。空格符不是语言符号（除了断行符，之前讨论过），但是可以用来分隔语言符号。当解释某个语言符号存在歧义时，该语言符号被看作是由一个尽可能长的字符串组成的合法符号（从左至右）。

标识符及关键字

标识符（也被称为名字）是由以下词法定义描述的。

Python 语言中的标识符语法基于 Unicode 标准附件 UAX-31，阐述变化定义如下；也可以点击 PEP 3131 (http://www.python.org/dev/peps/pep-3131) 获得更详细的信息。

在 ASCII 范围内（ U+0001..U+007F ），合法的标识符字符与 Python 2.X版本中是一致的：大写及小写字母的 `A` 到 `Z` ，下划线 `_` 以及数字从 `0` 到 `9` （第一个字符除外）。

Python 3.0版本引入了 ASCII 范围以外其他的字符（详见 PEP 3131 (http://www.python.org/dev/peps/pep-3131) ）。这些字符使用包含在 unicodedata (https://docs.python.org/3/library/unicodedata.html#module-unicodedata) 模块中的 Unicode 字符数据库版本进行分类。

标识符的长度是没有限制的。

identifier ::= xid_start (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-xid_start) xid_continue (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-xid_continue) *
id_start ::=
id_continue ::=
xid_start ::=
xid_continue ::=

上面提到的 Unicode 分类代码分别代表：

- *Lu* – 大写字母
- *Ll* – 小写字母
- *Lt* – 标题字母
- *Lm* – 修饰字符
- *Lo* – 其他字符
- *Nl* – 字母数字
- *Mn* – 无空格标记
- *Mc* – 空格组合标记
- *Nd* – 十进制数字
- *Pc* – 连接符
- *Other_ID_Start* – PropList.txt (http://www.unicode.org/Public/6.3.0/ucd/PropList.txt) 中的显示字符列表，支持后向兼容
- *Other_ID_Continue* – 同上

解析器会将所有标识符转换为标准形式的 NFKC，标识符的比较是基于 NFKC 的。

非标准的 HTML 文件列出了Unicode 4.1 中用到的所有合法标识符，可以在http://www.dcl.hpi.uni-potsdam.de/home/loewis/table-3131.html中找到。

关键字

下面列出的标识符被用作保留字，或者叫做该语言的关键字，这些保留字不能作为普通标识符使用。这些关键字必须严格像下面一样拼写：

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

保留的标识符类型

不同类型的标识符（关键字除外）都有其特殊含义。这些类型由前导和尾随的下划线字符模式确定：

`_*`
不能由 `from module import *` 导入。特殊字符 `_`，在交互式解释器中用于存储上次估值的结果；它存储在 builtins (https://docs.python.org/3/library/builtins.html#module-builtins) 模块中。在非交互式模式下，字符 `_` 没有特殊含义，且是未定义的。详见 *The import statement* (https://docs.python.org/3/reference/simple_stmts.html#import) 。

> 注意: `_` 常用于结合国际化；您也可以通过 gettext (https://docs.python.org/3/library/gettext.html#module-gettext) 获取更多关于此问题的信息。

`__*__`
系统定义的名称。这些名称由解释器及其实现定义(包括标准库)。当前系统名称在 Special method names (https://docs.python.org/3/reference/datamodel.html#specialnames) 部分讨论。更多的系统名称可能会在后续的 Python 版本中定义。在任意上下文中不遵循明确记录的使用 `__*__` 名字，会更容易产生错误且不会提示任何告警。

`__*` 类私有名称。在一个类定义的上下文中使用此类别中的名字时，为避免基类及派生类中“私有”属性之间的命名冲突，会被重写为一种乱序形式。

文本

文本是某些内建类型常量的表示方法。

字符串文本及字节文本

字符串文本由以下词法定义描述：

stringliteral ::= [stringprefix (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-stringprefix)] (shortstring (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-shortstring) | longstring (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-longstring))
stringprefix::= "r" | "u" | "R" | "U"
shortstring ::= "'" shortstringitem (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-shortstringitem) * "'" | '"' shortstringitem (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-shortstringitem) * '"'
longstring ::= "'''" longstringitem (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-longstringitem) * "'''" | '"""' longstringitem (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-longstringitem) * '"""'
shortstringitem ::= shortstringchar (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-shortstringchar) | stringescapeseq (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-stringescapeseq)
longstringitem ::= longstringchar (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-longstringchar) | stringescapeseq (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-stringescapeseq)
shortstringchar ::=
longstringchar ::=
stringescapeseq ::= "\"

bytesliteral ::= bytesprefix (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-bytesprefix) (shortbytes (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-shortbytes) | longbytes (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-longbytes))
bytesprefix::= "b" | "B" | "br" | "Br" | "bR" | "BR" | "rb" | "rB" | "Rb" | "RB"
shortbytes ::= "'" shortbytesitem (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-shortbytesitem) * "'" | '"' shortbytesitem (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-shortbytesitem) * '"'
longbytes ::= "'''" longbytesitem (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-longbytesitem) * "'''" | '"""' longbytesitem (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-longbytesitem) * '"""'

shortbytesitem ::= shortbyteschar (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-shortbyteschar) | bytesescapeseq (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-bytesescapeseq)
longbytesitem ::= longbyteschar (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-longbyteschar) | bytesescapeseq (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-bytesescapeseq)
shortbyteschar ::=
longbyteschar ::=
bytesescapeseq ::= "\"

上面没有表示出来的一个语法限制是在 stringprefix (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-stringprefix) 或 bytesprefix (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-bytesprefix) 和字符串文本之间不允许出现空格。编码声明定义了源字符集；如果在源文件中未明确声明编码方式，则默认使用 UTF-8 编码；详见 Encoding declarations (https://docs.python.org/3/reference/lexical_analysis.html#encodings) 部分。

简单来说：两种类型的字符串文本都可以用成对的单引号(`'`)或双引号（`"`）括起来。它们也可以用成对的三个单引号或双引号括起来（这些通常统称为三重引号的字符串）。反斜线字符（`\`）称为转义字符，如果不转义则其可能会产生歧义，例如换行符、反斜线本身或者引号。

字节文本总是以 `‘b’` 或 `‘B’` 开头；它们会生成 bytes (https://docs.python.org/3/library/functions.html#bytes) 类型实例而不是 str (https://docs.python.org/3/library/stdtypes.html#str) 类型。它们可能只包含 ASCII 字符；大于或等于 128 的数字必须使用转义字符表示。

Python 3.3 中可能会再次在字符串文本前增加 `u` 字符前缀，用以简化 2.X 及 3.X 版本代码的维护工作。

字符串文本及字节文本都可以（可选）加上字符前缀 `r` 或 `R`；这样的字符串被称为原始字符串，将反斜线视为原义字符。 这样会导致在字符串文本中，原始串中的转义字符 `‘\U’` 及 `‘\u’` 不会被特殊处理。考虑到 Python 2.X 版本中的原始 unicode 类型的特性与 3.X 版本差距较大，因此在 Python 3.X 版本中不再支持 `‘ur’` 语法。

3.3 版本新特性：原始字节文本的前缀 `‘rb’` 与 `‘br’` 意义相同。

3.3 版本新特性：旧版本中的（ `u‘value’` ）被再次引入，用以简化 Python 2.X 及 3.X 版本的维护工作。更多的信息，请参阅 PEP 414 (http://www.python.org/dev/peps/pep-0414) 。

在三重引用串中，允许出现未转义的新行和引用字符（并被保留），除非三个连续的引用字符串中断了该串。（引用字符是用于引用字符串的字符，如，`’` 或 `”`。）

如果给出一个 `‘r’` 或 `‘R’`，那么其含义就像标准C中的规则类似的解释，转义序列如下：

转义序列	意义	注意事项
\newline	反斜线且忽略换行	
\\	反斜线（\）	
\’	单引号（’）	
\"	双引号（”）	
\a	ASCII Bell（BEL）	
\b	ASCII 退格（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	八进制值为 *ooo* 的字符	（1,3）
\xhh	十六进制值为 *hh* 的字符	（2,3）

字符串文本中的转义序列规则如下：

转义序列	意　义	注意事项
\N{name}	Unicode 数据库中以 *name* 命名的字符	（4）
\uxxxx	16位16进制字符值：*xxxx*	（5）
\Uxxxxxxxx	32位16进制字符值：*xxxxxxxx*	（6）

注意：

1. 与标准 C 相同的是，最多只可以接受三位八进制数字。
2. 与标准 C 不同的是，只能接收两个十六进制数字。
3. 在字节文本中，十六进制和八进制转义字符表示给定值的字节数。在字符串文本中，这些转义字符表示给定值的 Unicode 字符。
4. 与 3.3 版本不同之处：增加了对别名[1 (https://docs.python.org/3/reference/lexical_analysis.html#id13)]的支持。
5. 可以使用该转义序列为那些构成代理对的单个代码单元编码。只能使用四个十六进制数表示。
6. 任何 Unicode 字符都可以采用这样的编码方式。需要注意的是只能使用八个十六进制数表示。

不像标准 C，所有不能被识别的转义序列都保留在串中且不做改变，例如，反斜线会保留在结果中。（这个行为在调试过程中非常有用：如果输入了一个错误的转义序列，在输出结果中更容易识别出错误。）此外，至关重要的是要注意转义字符只能在字符串文本中起作用，在字节文本 类别中无法被识别。

即使在原始文本中，可以使用反斜线将引号转义，但是反斜线本身会在结果中保留；比如 `r"\""` 是一个由两个字符组成的合法字符串：一个反斜线与一个双引号；但 `r"\"` 却是一个非法字符串（即原始的字符串也不能以奇数个反斜杠结尾）。具体而言，一个原始的文本不能以单个反斜杠结尾（由于反斜线会将跟在其后的引号转义）。另外需要注意的是，如果一个反斜线跟在换行符后，反斜线与换行符会被当做文本的两个字符，而不是一个连续行。

字符串的连接

多个相邻的字符串文本或字节文本（由空白分隔），允许使用不同的引用习惯，并且其含义与连接在一起时是一样的。因此，`"hello" ‘world’` 与 `"helloworld"` 是等价的。这个特性可以用来减少反斜线的使用数量，可以很方便的将一个长字符串分隔在多行上，甚至可以在字符串的某一部分添加注释，例如：

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

需要注意的是，这个特性是定义在句法层次上的，但是是在编译时实现的。在运行时连接串必须使用 ‘+’ 运算符。并且不同的引用字符可以混用，甚至可以将原始串与三重引用串混合使用。

数字文本

总共有三种类型数字文本：整型、浮点型以及虚数型。不存在复数文本（复数可以由一个实数加一个虚数的形式给出）。

注意，数字文本不包含符号（正负号）；像 `-1` 实际上是一个组合了一元运算符 `'-'` 和数字 `1` 的表达式。

整型文本

整型文本描述的词法定义如下：

integer::= decimalinteger (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-decimalinteger) | octinteger (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-octinteger) | hexinteger (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-hexinteger) | bininteger (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-bininteger)
decimalinteger ::= nonzerodigit (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-nonzerodigit) digit (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-digit) * | "0"+

nonzerodigit ::= "1"..."9"
digit ::= "0"..."9"
octinteger ::= "0" ("o" | "O") octdigit (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-octdigit) +
hexinteger ::= "0" ("x" | "X") hexdigit (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-hexdigit) +
bininteger ::= "0" ("b" | "B") bindigit (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-bindigit) +
octdigit ::= "0"..."7"
hexdigit ::= digit (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-digit) | "a"..."f" | "A"..."F"
bindigit ::= "0" | "1"

除了可以存储在可用内存中外，对整型文本的长度是没有限制的。

注意，不允许在非零的小数前加零。这是为了消除与 Python 3.0 之前版本中使用的 C 风格八进制文本的歧义。

整型文本的例子如下：

```
7   2147483647                    0o177          0b100110111
3   79228162514264337593543950336   0o377          0x100000000
    79228162514264337593543950336                  0xdeadbeef
```

浮点型文本

浮点型文本描述的词法定义如下：

floatnumber ::= pointfloat (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-pointfloat) | exponentfloat (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-exponentfloat)
pointfloat ::= [intpart (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-intpart)] fraction (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-fraction) | intpart (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-intpart) "."
exponentfloat ::= (intpart (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-intpart) | pointfloat (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-pointfloat)) exponent (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-exponent)
intpart ::= digit (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-digit) +

fraction ::= "." digit (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-digit) +
exponent ::= ("e" | "E") ["+" | "-"] digit (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-digit) +

注意，浮点型文本的整数部分及指数部分都使用十进制表示。比如，`077e010` 是合法字符，并且与 `77e10` 等价。浮点型文本允许的范围是依赖于实现的。一些浮点型文本的例子如下：

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

注意，数字文本不包含符号（正负号）；像 `-1` 实际上是由一元运算符 `-` 和文本 `1` 组成。

虚数文本

虚数文本描述的词法定义如下：

imagnumber ::= (floatnumber (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-floatnumber) | intpart (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-intpart)) ("j" | "J")

虚数是一个实部为零的复数，复数代表一对有着相同取值范围限制的浮点数对。为了创建一个实部非零的复数，可以对它增加一个浮点数，例如（ `3+4j` ）。下面是一些关于虚数文本的例子：

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

运算符

下面列出的符号为操作符：

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

分隔符

以下符号用作语法上的分隔符：

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

句号也可以在浮点型及虚数文本中出现，一个连续三个句号的序列具有特殊含义，代表了一个片段中的省略部分。该列表的后半部分，即参数化赋值运算符，在词法上是作为分隔符处理，但也执行运算。

下面列出的 ASCII 字符作为其他符号的一部分具有特殊含义，或者对于词法分析器具有重要意义。

```
'   "   #   \
```

下面列出的 ASCII 字符没有在 Python 中使用。当它们出现在字符串文本及注释之外时就认为是非法的。

```
$   ?   `
```

3

数据模型

对象，值和类型

对象是 Python 的抽象数据。在一个Python程序中，所有数据都通过对象或对象之间的关系表示。（在某种意义上讲，和冯·诺依曼的“存储程序计算机”的模型一致，代码也是由对象所表示的。）

每一个对象都具有一个标识，一个类型和一个值。对象一旦建立，它的表示就永远不能改变了；你可以认为它是在内存中的地址。‘is (https://docs.python.org/3/reference/expressions.html#is)’ 运算符用来可以比较两个对象的身份；‘id (https://docs.python.org/3/library/functions.html#id)’ 可以获得一个整数形式的对象标示。

CPython 的实现细节：对于CPython，`id(x)` 是 `x` 存储的内存地址。

一个对象的类型决定了这个对象所支持的操作（例如，“有长度吗？”），并且还定义了针对该类型对象的可能值。type() (https://docs.python.org/3/library/functions.html#type) 函数返回一个对象的类型（这是一个对象本身）。正如它的身份一样，一个对象的*类型*也是不可以改变的。[1] (https://docs.python.org/3/reference/datamodel.html#id4)

某些对象的*值*是可以改变的。那些值可以改变的对象被认为是*可变的*；那些值不可以改变的对象一旦创建就被称为不可变的。（属于不可变对象的容器在包括有可变对象的引用时，当可变对象改变了时，其实是可变的，但仍被看作是可变对象，因为它所包含的对相机集合不可变的，所以不可变对象和值不可变是不完全一样的，它更加微妙。）一个对象的可变性是由它的类型来确定的；例如，数值，字符串和元组是不可变的，而字典和列表是可变的。

对象不用显式地释放他们；然而，当这些对象变的不能访问时，它们可能当做垃圾被收集。一种处理方式是延迟垃圾收集或者完全忽略 —— 这是回收机制如何实现的质量问题，只要求尚能访问的对象不被回收。

CPython 的实现细节：CPython 当前实现使用一个引用计数机制和一个可选的循环垃圾延时检测机制，只要对象不可用了，就会回收大部分这样的对象，但不能保证回收中包含有循环引用。对于循环回收的详细控制情况，见 gc (https://docs.python.org/3/library/gc.html#module-gc) 模块参考。其他不同的实现方式在 CPython 中可能会改变。 当对象变得不能实现时，不要依赖对象可以立马终结（所以你需要经常明确地关闭文件）。

注意使用实现的跟踪和调试工具时可能会保留本该回收的对象，也要注意使用语句 ‘try... (https://docs.python.org/3/reference/compound_stmts.html#try) except’ (https://docs.python.org/3/reference/datamodel.html) 也可能保留本该回收的对象。

有些对象包含有“外部”资源，像文件或窗口。可以相信在垃圾回收时这些资源也被释放，但因为垃圾回收不保证一定发生，这样的对象提供了显式的方法释放这些资源，通常是用 close() 方法。特别推荐使用这种方法释放包含外部资源的对象。 ‘try... (https://docs.python.org/3/reference/compound_stmts.html#try) finally’

(https://docs.python.org/3/reference/compound_stmts.html#finally) 和 ‘with (https://docs.python.org/3/reference/compound_stmts.html#with) ’ 提供了这样的一个便利途径。

有些对象包含了其它对象的引用；它们叫做容器。容器的例子有元组，列表和字典。引用作为容器值的一部分。大多数情况下，当我们谈及一个容器的值时，我们只是涉及了这个值，而不是所包含的对象；但是，当我们谈及容器对象的可变性的时候，就只是涉及被直接包含的对象的标识了。因此，如果一个不可变对象（如元组）包含了一个可变对象，那么只要这个可变对象的值变了则它的值也就改变了。

类型对对象的大多数行为都有影响。甚至在某种程度上对对象的标识也有重要的影响：对于不可变对象，计算新值的运算符可能实际上返回的是一个已经存在的具有相同类型和值的对象的引用，对于可变对象，只是不允许的。例如，在 `a=1; b=1` 之后， `a` 和 `b` 有可能指向一个具有值为1的对象，这依赖于实现，但 `c=[];d=[]` 之后， `c` 和 `d` 可以保证是保存了两个不同的，独立的，新创建的空列表。（注意 `c=d=[]` 是把相同的对象赋予了 `c` 和 `d` 。）

标准类型层次

以下是 Python 内建类型的一个列表，扩展模块（ 用C 语言，Java，或者其他语言写成，取决于实现方式）可以定义其他类型。以后版本的 Python 可能会在这个类型层次中增加其他类型（例如：有理数，高效率存储的整数数组，等等），即使通过标准库将会提供这些类型。

有些类型描述中包括了一个列出“特殊属性”的段落。他们提供了一些访问实现的方法而不是作为一般目的使用。这些定义可能会在未来改变。

无 None
这个类型具有单一值，也只有一个对象有这个值，这个对象可以通过内建名字 `None` 访问它在许多场合不是无值，例如它在那些没有返回值的函数中返回，其值为真值。

未实现 NotImplemented
这个类型具有单一值，也只有一个对象有这个值，这个对象可以通过内建名字 `NotImplemented` 访问。如果数值方法和许多比较方法的操作数未提供其实现，他们就可能返回这个值。（解释器会试图扩展成其它操作，或者其它退化的擦做，它依赖于运算符）他的真值为真。

更多细节见 Implementing the arithmetic operations (https://docs.python.org/3/library/numbers.html#implementing-the-arithmetic-operations) 。

省略 Ellipsis
这个类型具有单一只，也只有一个对象有这个值。这个对象可以通过文字 `...` 或者内建名字 `Ellipsis` 访问。其真值为真。

数值型 nubmbers.Numbers (https://docs.python.org/3/library/numbers.html#numbers.Number)
它们由数值型字面值生成，或由算术运算和内置算术函数作为值返回。数值型对象是不可变的；一旦创建其值就不可以改变。Python 的数值型和数学上的关系非常密切，但要受到计算机的数值表达能力的限制。

Python 对整数，浮点数和复数区别对待：

numbers.Integral (https://docs.python.org/3/library/numbers.html#numbers.Integral)
描述了数学上的整数集（正数和负数）。

有2种整数类型：

数值型整数 Integers (int (https://docs.python.org/3/library/functions.html#int))
数值型整数的表示范围在语言上是无限制的，只受限于你的内存（虚拟内存）。对于以移位和屏蔽为目的的运算，数值型正数被认为是二进制的形式，如果是负数，那么就被转换成二进制补码形式，符号位向左扩展。

布尔型整数 Booleans (bool (https://docs.python.org/3/library/functions.html#bool))
布尔型整数的真值是用假和真来表示的。对于布尔型对象的值只能通过 `假` 和 `真` 表示。布尔类型是整数类型的子类型，分别用数值 0 和 1 来表示，但是当返回值的类型是字符时，其分别用 `"False"` 或者 `"True"` 表示。

整数表示法的这个规则是为了使对负数操作时尽量有实际意义。

浮点数 numbers.Real (float (https://docs.python.org/3/library/functions.html#float))
本类型表示了机器级的双精度浮点数。是机器的体系结构和C实现的浮点数范围和控制溢出方法屏蔽你不需知道的细节。Python 不支持单精度浮点数，使用它的原因通常是减少CPU负荷和节省内存，但是这一节省被对象在Python中的使用开销所抵消，因此没有必要支持两种浮点数使语言变的复杂。

复数 numbers.Complex (complex (https://docs.python.org/3/library/functions.html#complex))
本类型描述了一对机器级的双精度浮点数，在浮点数中的警告内容也适用于复数类型。复数 `z` 的实部和虚部可以通过属性 `z.real` 和 `z.imag` 获得。

有序类型 Sequences
本类型描述了用非负数索引的有限的有序集合。内建函数 len() (https://docs.python.org/3/library/functions.html#len) 返回有序类型数据中的项数当有序类型数据的长为 *n* 时，索引号为 0,1，...,*n*−1。有序类型数据 *a* 中的项 *i* ,可以以 `a[i]` 表示。

有序类型也支持片断： `a[i:j]` 可以表示满足 *i* `<=` *k* `<` *j* 的所有项 `a[k]` 。在使用这个表达式时，它具有相同的有序类型，这也隐含着索引重新从0开始计。

有些有序类型用第三个“步骤”参数来支持“扩展片断”： `a[i:j:k]` 选择所有项的索引 *x*,其中 `x = i + n * k` ，*n* `>=` 0 并且 *i* `<=` *x* `<` *j*。

有序类型按照可变性分为两类：

不可变类型 Immutable sequences

一个不可变对象一旦建立其值不可能再改变。（如果这个对象包括其它对象的引用，这个被引用的对象可以是可变对象并且其值可以变化；但为该不可变对象所直接引用的对象集合是不能变的。）

以下类型是不可变类型：

字符串 Strings

字符串的序列值体现了 Unicode 的代码点，字符串中所有代码点的范围是 `U+0000 - U+10FFF` 。Python 不支持 char 类型，相反，每个代码的字符串用长度 `1` 表示一个字符串对象。内建函数 ord() (https://docs.python.org/3/library/functions.html#ord) 从字符串转换为代码点形成一个整数的范围 `0 - 10FFFF` ，chr() (https://docs.python.org/3/library/functions.html#chr) 将一个范围在 `0 - 10FFFF` 的整数对应长度 `1` 的字符串对象。str.encode() (https://docs.python.org/3/library/stdtypes.html#str.encode) 可以使用给定的明文编码将 str (https://docs.python.org/3/library/stdtypes.html#str) 转换成 bytes (https://docs.python.org/3/library/functions.html#bytes)， 并且 bytes.decode() (https://docs.python.org/3/library/stdtypes.html#bytes.decode) 可以实现相反转换。

元组 Tuples

元组对象的项可以是做任意的 Python 对象。具有两个或多个项的元组是用逗号分隔开表达式列表，只有一项的列表（独元）其项可以后缀一个逗号使其成为满足要求元组要求的表达式（一个表达式本身不能创建一个元组,因为圆括号本身用于表达式的分组）。一个空元组可以以一对空圆括号表示。

字节 Bytes

字节对象是不可变的数组，每个项由8位字节组成，由整数范围为 0 <= x < 256 表示。字节文字（如 `b'abc'`)和内建函数 bytes() (https://docs.python.org/3/library/functions.html#bytes) 可以被用来构建字节对象。同时，字节对象可以经由 decode() (https://docs.python.org/3/library/stdtypes.html#bytes.decode) 方法被解码为字符串。

可变对象 Mutable sequences

可变对象可以在其创建后改变，其下标表示和片断表示可以作为赋值语句和 del (https://docs.python.org/3/reference/simple_stmts.html#del)（删除）语句的对象。

目前只有一种可变类型。

列表 Lists

列表对象的项可以是任意类型的 Python 对象。列表对象是在方括号之间的用逗号分开的表达式表。（注意，形成长度为0或者1的链表不要求特别的写法）。

字节数组 Byte Arrays

字节数组对象是一个可变的数组，它们是由内建函数 bytearray() (https://docs.python.org/3/library/functions.html#bytearray) 构造创建。除了是可变的（和无序的），字节数组以其他方式提供相同的界面和功能不变的字节对象。

扩展模块 array (https://docs.python.org/3/library/array.html#module-array) 提供一个额外的可变序列类型模板， collections (https://docs.python.org/3/library/collections.html#module-collections) 模块。

集类型 Set types

集类型对象具有无序性，有限集的独一无二性，和不变性。因此，它们不能被任何下标所搜引，然而，它们可以进行遍历，并有内建函数 len() (https://docs.python.org/3/library/functions.html#len) 返回遍历数。集的常见应用包括快速加入测试，删除重复虚了，和计算数学操作，如十字路口，并，差异和对称查分。

对于集元素，相同的不变性属性作为申请字典关键字。注意，数字类型遵从数字比较的正常规则：如果两个数字比较后相等（比如 `1` 和 `1.0` ），那么会只有一个数字包含在集合里。

目前有两种原始集类型：

集 Sets

这种代表可变的集合，它们是由内建函数 set() (https://docs.python.org/3/library/stdtypes.html#set) 构建，并且之后可变用多发方法去修改的，比如 add() (https://docs.python.org/3/library/stdtypes.html#set.add) 。

冻结集 Frozen sets

这种代表不可变的集合，它们是由内建函数 frozenset() (https://docs.python.org/3/library/stdtypes.html#frozenset) 构建，由于冻结集的不可变性以及无序性 *hashable* (https://docs.python.org/3/glossary.html#term-hashable) ，它也可以用来作另外一个集合的元素或者字典的关键字。

映射类型 Mappings

本类型描述了可以是由任意类型作索引的有限对象集合。下标表示法 `a[k]` 从映射类型对象中选择一个由 `k` 索引项，它们可以用作赋值语句和 del (https://docs.python.org/3/reference/simple_stmts.html#del) 语句的目标。内建函数 len() (https://docs.python.org/3/library/functions.html#len) 返回映射对象的项的个数。

当前只有一个内置的映射类型：

字典 Dictionaries

本类型表达了几乎是任意类型对象都可作索引的有限对象集合。不可接受的键值仅仅是那些包括有列表和字典的值，或那些是通过值比较而不是通过对象标识符比较的类型的值。其原因是字典的高效实现要求键的哈希值保持不变。用于键值的数值型在比较时使用通常的规则：如果两值相等（如： `1` 和 `0` ），那么它们可以互换地索引相同的字典项。

字典是可变的，它们由 ... 语法创建（详见 *Dictionary displays* (https://docs.python.org/3/reference/expressions.html#dict) ）。

扩展模块 dbm.ndbm 和 dbm.gnu 提供了另外的映射类型的例子， 同 collections (https://docs.python.org/3/library/collections.html#module-collections) 模块。

可调用类型 Callable types

这是一个可用的函数调用操作类型（详见 *Calls* (https://docs.python.org/3/reference/expressions.html#calls) ），应用于：

用户自定义函数 User-defined functions

一个自定义函数对象由函数定义（详见 *Function definitions* (https://docs.python.org/3/reference/compound_stmts.html#function) ）。创建在函数调用时应该与定义时的形式参数相同数目参数。

特殊属性：

属性	意义	–
`__doc__`	是函数的文档串，如果无效就是 `None` ，不可以被子类继承	可写
`__name__`	是函数名称	可写
`__qualname__`	是函数的限定名称，在版本 *3.3.* 中的新属性	可写
`__module__`	定义在函数模块里的定义，如果无效就是 `None`	可写
`__defaults__`	一个元组包含默认参数值，这些参数是默认的，如果没有默认参数值就是 `None` ,	可写
`__code__`	一个编译后的代码对象，	可写
`__globals__`	是一个引用，指向保存了函数的全局变量的字典——如果在函数所在模块中定义了全局变量的话，	只读
`__dict__`	包含了支持任意函数属性的命名空间，	可写
`__closure__`	要么是 `None` ，要么是包括有函数自由变量绑定的单元的元组，	只读
`__annotations__`	是一个包含注释的字典参数，字典的关键字是参数名字，并且如果有提供的话，返回注释 `return` ,	可写
`__kewdefaults__`	是一个只包含默认关键字的字典，	可写

大部分的“可写”属性需要检查分配的值的类型。

函数对象也支持获取和设置任意属性，它可以被使用，例如，元数据功能。常规属性一样用于获取和设置这些属性。*注意，当前的实现只支持函数属性对用户定义的函数，功能属性内置函数可能在未来才会支持。*

关于函数的定义可以由它的代码对象得到；见下面关于内部对象的描述。

实例方法 Instance methods

实例方法合并了一个类，一个类实例和任何可调用对象（一般是用户自定义函数）。

特殊的只读属性： `__self__` 是类实例对象， `__func__` 是函数对象； `__doc__` 是其方法的文档（同 `__func__.__doc__` ）； `__name__` (https://docs.python.org/3/reference/import.html#__name__) 是方法名（同 `__func__.__name__` ）； `__modele__` 是定义方法的模块名，如果无效则为无 None。

方法也支持访问（不是设置）的其实际所调用的函数的任意属性。

如果类的属性是一个用户定义函数对象或类方法对象，那么获得类属性的用户可以创建自定义方法对象（也可能通过这个类的一个属性）。

当用户自定义方法对象采用一个用户自定义函数对象来创建时，这个对象来自于它的一个实例类，它的 `__self__` 属性是个实例，并且该方法称为是捆绑的。新方法的 `__func__` 属性是原始函数对象。

当用户自定义方法对象采用另外一个类或实例的方法对象创建时，其特性与函数对象创建时相同的，不同之处是这个 `__func__` 新类的属性不是原始函数对象，而是它的 `__func__` 属性。

当调用实例方法对象时，实际调用的是函数（ `__func__` ），同时在参数列表前插入类实例（ `__self__` ）。比如：当类 C 包含了一个函数定义 f()，并且 `x` 是 C 的一个实例，调用 `x.f(1)` 相当于调用 `C.f(x,1)` 。

当一个实例方法对象时由一个类方法对象衍生而来时，存储在 `__self__` 的“类实例”实质上将会是类本身，所以不论调用 `x.f(1)` 或者 `C.f(1)` 相当于调用 `f(C,1)` , 其中 `f` 是实际函数。

注意，由函数对象转变到实例方法对象，每次都会检索实例中的属性。在某些情况下，卓有优化是成效的优化是将属性分配给一个局部变量并调用它。同时应注意，这种转变只会发生在在用户自定义函数上，检索其他的可调用对象（以及不可调用对象）不需要这种转变。同样重要的是要注意，具有类实例属性的用户自定义函数不能转换成绑定方法；*当且仅当*发生在函数是一个类的属性时。

生成器函数 Generator functions

一个使用 yield (https://docs.python.org/3/reference/simple_stmts.html#yield) 的语句（见 yield (https://docs.python.org/3/reference/simple_stmts.html#yield) 语句 ）的方法或函数，它叫做生成器函数。这样的函数，在返回后，通常返回一个迭代子对象，一个可以用于执行函数的对象。调用对象的 iterator.next() (https://docs.python.org/3/library/stdtypes.html#iterator.__next__) 方法会引起函数的执行，直到其使用 yield (https://docs.python.org/3/reference/simple_stmts.html#yield) 语句返回一个值。当函数在执行到 return (http

s://docs.python.org/3/reference/simple_stmts.html#return) 语句时，或在末尾结束时，会抛出异常 StopIteration (https://docs.python.org/3/library/exceptions.html#StopIteration)，此时迭代器也到达了返回值集合的结尾。

内建函数 Built-in functions

一个内建函数就是一个 C 函数的包装，例如内建函数 len() 和 math.sin()（math 是标准的内建模块）。参数的类型和数目都由C函数检测，特殊的只读属性：`__doc__` 是函数的文档串，或者无效为无 `None`；`__name__` (https://docs.python.org/3/reference/import.html#__name__) 是函数名；`__self__` 设为 `None`（见下述）；`__module__` 是模块名，或者无效为无 `None`。

内建方法 Built-in methods

这实际上是内建函数的一个不同的包装而已，此时传递给C函数一个对象作为隐含的参数。例如，内建方法 `alist.append()` 假定 *alist* 是一个列表对象，此时特殊只读属性 `__self__` 指定为对象 *alist*。

类 Classes

类是可调用的。这些对象通常是自己的新实例的工厂，但是变化可能是重写 `__new__` () (https://docs.python.org/3/reference/datamodel.html#object.__new__) 类对象。传送调用的参数到 `__new__` () (https://docs.python.org/3/reference/datamodel.html#object.__new__)，而且典型情况是传送到 `__init__` () (https://docs.python.org/3/reference/datamodel.html#object.__init__) 来初始化成新实例。

类实例 Class Instances

任意类实例可以通过定义一个 `__call__` () (https://docs.python.org/3/reference/datamodel.html#object.__call__) 方法类去调用。

模块 Modules

模块是 Python 代码的基本组成单元，并由导入系统 *import system* (https://docs.python.org/3/reference/import.html#importsystem) 创建，可以通过 import (https://docs.python.org/3/reference/simple_stmts.html#import) 语句或者诸如函数 importlib.import_module() (https://docs.python.org/3/library/importlib.html#importlib.import_module) 和内建函数 `__import__` () (https://docs.python.org/3/library/functions.html#__import__)。一个模块有一个名字空间，用字典对象实现（在模块定义中的函数可以通过 `__globals__` 属性来访问这个字典）。把对属性的访问翻译成查找这个字典。例如，`m.x` 等价于 `m.__dict__["x"]`。模块对象并不包含用来初始化该模块的代码对象（因为一旦初始化完成它就不再需要了）。

属性的赋值更新模块的空间名字，例如，`m.x = 1` 等价于 `m.__dict__["x"]`。

特殊只读属性：`__dict__` 是字典形式的模块名字空间。

Python 实现细节：由于 CPython 清理模块字典的方式，当模块超出范围时，即使字典还在引用，字典模块也将被消除。为避免这种情况发生，在直接使用字典时应该备份字典或将其保留在模块附近。

预定义的可写属性： `__name__` (https://docs.python.org/3/reference/import.html#__name__) 是模块名； `__doc__` 是模块的文档串，如果无效即为 `None` ； `__file__` 是被装载模块的文件路径名， 因为C模块不提供些属性，该模块已连接至解释器中，对于那些从共享库装载的模块，这个属性是那些共享库的路径。

自定义类 Custom classes

自定义类类型由类定义创建（见类定义 *Class definitions* (https://docs.python.org/3/reference/compound_stmts.html#class) ）。一个类有一个用字典对象实现的名字空间，类属性的访问可以转换成对该字典的查找，例如， `C.x` 被解释成 `C.__dict__` ["x"]`（虽然有许多允许定位属性的挂钩）。当属性名未找到时，查找是深度优先的。这种基本类搜索使用 C3 方法解析正确行为，即使在‘钻石’继承结构那里有多重继承回到共同祖先的路径。更多细节在 Python 中使用的 C3 MRO，参考随着 2.3发布的文档里 https://www.python.org/download/releases/2.3/mro/。

当一个类属性引用（对于类 C 说）应该让步于一个类方法对象时，它会变成一个实例方法对象，其属性 `__self__` 是 C。当它应该让步于一个静态方法对象时，它会变成静态方法对象的封装对象。见另一种方式的实现描述 *Implementing Descriptors* (https://docs.python.org/3/reference/datamodel.html#descriptors) ，从一个类检索属性可能不同于那些实际包含在其字典 `__dict__` 。

类属性的赋值会更新类的字典，而不是基类的字典。

一个类对象可以创建(见上)，这样会产生一个类实例(下述)。

特殊属性： `__name__` (https://docs.python.org/3/reference/import.html#__name__) 是类名； `__module__` 是类所定义的模块名； `__dict__` 是包括类名字空间的字典； `__bases__` (https://docs.python.org/3/library/stdtypes.html#class.__bases__) 是一个元组（可能是空或独元），包括其基类，以基类列表中他们的排列次序出现； `__doc__` 是类的文档串，如果无效，它就是无 None。

类实例 Class instances

类实例可以通过调用一个类对象来创建，类实例可以有一个用字典实现的名字空间，它只由搜索属性范围的第一个结果构成。如果属性没在那找到，并且实例的类有那个名字的属性，就继续在类属性中寻找。如果找到的是一个用户自定义函数对象(而不是其它情况)，它被转换成一个实例方法对象，它的 `__self__` 属性是个实例。 静态方法和类方法对象也可以转换，见前文在 “Classes” 中所述。另一种方式，通过实例检索类的属性可能不同于对象实际存储在类 `__dict__` 中的，见 *Implementing Descriptors* (https://docs.python.org/3/reference/datamodel.html#descriptors) 部分。如果没有类属性找到，并且对象的类有方法 `__getattr__` (https://docs.python.org/3/reference/datamodel.html#object.__getattr__) ()，那就调用它来满足查找请求。

属性赋值和删除会更新实例目录，而不是类的字典，如果类具有方法 `__setattr__` () (https://docs.python.org/3/reference/datamodel.html#object.__setattr__) 和 `__delattr__` () (https://docs.python.org/3/reference/datamodel.html#object.__delattr__) ，则它们会在更新类实例属性时调用，而不是实例字典直接更新。

类实例可以伪装成数字，序列，或者具有方法映射成某些特别的名字。 见 *Special method names* (https://docs.python.org/3/reference/datamodel.html#specialnames) 部分。

特殊属性： `__dict__` (https://docs.python.org/3/library/stdtypes.html#object.__dict__) 是字典属性； `__class__` (https://docs.python.org/3/library/stdtypes.html#instance.__class__) 是实例属性。

输入/输出对象（或称文件对象） I/O objects
一个文件对象 *file object* (https://docs.python.org/3/glossary.html#term-file-object) 描述了一个开放的文件。 有多种可用的快捷方式来创建文件对象： 内建函数 open() (https://docs.python.org/3/library/functions.html#open) ,以及 os.open() (https://docs.python.org/3/library/os.html#os.fdopen) ， os.fdopen() (https://docs.python.org/3/library/os.html#os.fdopen) ，和套接字对象方法 makefile() (https://docs.python.org/3/library/socket.html#socket.socket.makefile) （或许也可以通过其他扩展模块的函数或方法）。

对象 `sys.stdin` ， `sys.stdout` ，和 `sts.stderr` 被相应的初始化成解释器的标准输入流，标准输出流，和标准错误输出流；它们都是开放文本模式，因此遵循抽象类 io.TextIOBase (https://docs.python.org/3/library/io.html#io.TextIOBase) 定义的接口。

内部类型 Internal types
少部分由解释器内部使用的类型，开放给了用户。它们的定义可能会在未来的解释器版本中改变，但都会在这儿提及。

代码对象 Code objects
代码对象表达了*字节编译过*的可执行 Python 代码，或者叫字节码 (https://docs.python.org/3/glossary.html#term-bytecode) 。代码对象和函数对象的不同在于函数对象包括了一个外在的全局变量引用（其所定义的模块），而代码对象不包括上下文。默认参数值存入函数对象中，而代码对象则不（因为它们的值由运行时计算）。不像函数对象，代码是不可改变的，并且不包括对可变对象的引用。

特殊属性： co _ name 给出了函数名；co _ argument 是位置参数的数目(包括有默认值的参数)；co _ nlocals 是函数中局部变量的数目。co _ varnames 是一个包括局部变量名的元组（从参数名开始）；co _ callvars 是一个元组，包括由嵌套函数所引用局部变量名；co _ freevals 包括了既不是局部变量也不是全局变量的；co _ code 包括了编译过的字节码指令序列的字符串；co _ consts 包括字节码中使用的字面值的元组；co _ names 包括字节码中使用的名字的元组；co _ filename 包括着编译过的字节码文件名；co _ firstlineno 是函数首行行号；co _ lnotab 是一个字符串，是字节码偏移到行号的映射（详见解释器代码）；co _ stacksize 是所需要的堆栈尺寸（包括局部变量）；co _ flags 是一个整数，其解释成为许多解释器的标志。

以下标志位由 co _ flags 定义：如果函数使用 `*argument` 语法来接受任意数目的位置参数，就置位 `0x04` ；如果函数使用 `*keywords` 语法来接受任意数量的关键字参数，就置位 `0x08` ；如果函数是个生成器，就置位 `0x20` 。

未来功能还声明（ from __future__ import division ）在 co _ flages 中使用位表明编译代码对象是否启用了一个特定的嵌入式功能：如果编译函数启用了嵌入式功能，则置位 0x2000 ；而在老版的 Python 中使用位 0x10 和 0x1000 。

在 co _ flages 中的其他为预留为内部使用。

如果一个代码对象描述一个函数，则 co _ consts 的第一项是该函数的文档串，如果未定义，它就是无 None 。

堆栈结构对象 Frame objects
堆栈结构对象描述了可执行结构，它们会在跟踪回溯对象中出现（下述）。

特殊只读属性：f _ back 指出前一个堆栈结构（向着调用者的方向），如果位于堆栈的底部它就是 None ；f _ code 指出本结构中能执行的代码对象。f _ locals 是用于查找局部变量的字典；f _ globals 用于全局变量；f _ builtin 用于内建名字；f _ lasti 给出精确的指令（是一个代码对象的字符串的索引）。

特殊可写属性：f _ trace 如果非空，就是从每个源代码行的开始处调用的函数（用于调试器）;f _ lineno 给出行号——写这个函数从内部跟踪的行（只限于最底层堆栈）。调试器可以通过编写 f _ lineno 实现跳转命令(即设置下一条语句)。

堆栈结构对象支持一种方法：

堆栈结构清除 frame.clear()
这种方法清除所有局部变量的引用。同时，这个结构堆栈属于一个发生器，那么会终结这个发生器。这有助有终止堆栈结构对象循环（比如，供以后使用异常捕获和回溯存储）。

如果当前结构正在执行，上报 RuntimeError (https://docs.python.org/3/library/exceptions.html#RuntimeError) 。

新版本3.4。

跟踪回溯对象 Traceback objects
跟踪回溯对象描述一个异常的栈回溯，跟踪回溯对象在异常发生时创建，在展开可执行堆栈搜索异常处理器时，每个展开层的跟踪回溯对象插进当前跟踪回溯对象的前面。在遇到异常处理器时，跟踪回溯对象也对程序有效了。（见 *try* (https://docs.python.org/3/reference/compound_stmts.html#try) 语句）它可以由 sys.exc _ info() 返回的元组的第三项访问到。后一种是推荐的接口，因为它也可以使用多线程的程序中工作良好。当程序中未包括适当的异常处理器, 跟踪回溯对象就被打印到标准错误输出流上。如果工作在交互方式上，它也可以通过 sys.last _ traceback 。

特殊只读属性：tb _ text 是堆栈的下一层（向着异常发生的那一层结构），或者如果没有下一层，就为 None 。tb _ frame 指出当前层次的可执行结构；tb _ lineno 给出异常发生的行号；tb _ lasti 指出精确的指令；如

果异常发生在没有 except 或 finally 子句匹配的 try (https://docs.python.org/3/reference/compound_stmts.html#try) 语句中的话，这里的行号和指令可能与结构中的行号和指令不同。

片断对象 Slice objects
片断对象用片段方法 __getitem()__ (https://docs.python.org/3/reference/datamodel.html#object.__getitem__) 来描述。他们也是通过内建函数 slice() (https://docs.python.org/3/library/functions.html#slice) 来创建。

特殊只读属性：start 是下界，step 是上界，step 是步进值，如果在片段中忽略了它，就是 None 。这些属性可以是任意类型的。

片断对象支持一种方法：

片断索引 slice.indices(self, length)
这个方法接受一个整数参数 *length* 并计算片断信息，切片对象用于描述 *length* 项序列。它返回一个元组包含为三个整数，分别代表*开始*索引，*停止*索引启动和偏度的*步进*或*步长*，处理缺失或界外指数的方式与普通片断一致。处理缺失或界外索引的方式与普通片断一致。

静态方法对象 Static method objects
静态方法对象提供一种战胜上述转换函数对象方法。静态方法对象包装在其他对象周围，通常是一个用户定义的方法对象。当一个静态方法对象从一个类或一个类实例检索时，返回的对象实际上是包装对象，它不受任何进一步的转换。它本身并不是调用静态方法对象,尽管他们通常包装对象，但并不能自我调用。静态方法创建的对象是内建函数 staticmethod() (https://docs.python.org/3/library/functions.html#staticmethod) 构造。

类方法对象 Class method objects
类方法的对象是一个在其他对象周围的包装器，就像一个静态方法对象，它改变从类和类实例中检索的方式。上述的这种类方法对象的方式在“用户自定义方法”中有体现。 用内建函数 classmethod() (https://docs.python.org/3/library/functions.html#classmethod) 构建器来创建类方法对象。

特殊方法名

一个类可以实现以特殊句法才调用的某些操作（例如算术运算，下标操作及片断操作），这是通过以特殊名字定义方法来实现的。这是 Python 的操作符重载方法，允许类来定义自己的行为对语言操作符。例如，如果一个类定义了的方法名为 __getitem__ () (https://docs.python.org/3/reference/datamodel.html#object.__getitem__)，并且 x 是这个类的实例，那么 x[i] 就等价于 type(x).__ getitem__ （x,i） 。除了所提及的地方，试图执行没有适当的方法定义的操作会引起一个异常的抛出（典型的 AttributeError (https://docs.python.org/3/library/exceptions.html#AttributeError) 和 TypeError (https://docs.python.org/3/library/exceptions.html#TypeError) ）。

当实现一个模拟任何内建类型的类时，重要的地方在于模拟的程度只要使对象模拟时有效就行了。例如，某些有序类型的对象可能在单独提取某引起值时有效，但在使用片断时是没有意义的。（这样的一个例子是在W3C的文档对象模型中的NodeList接口。）

基本定制

object.__new__(cls[,...])
在实例创建调用时，__new__ 是一个静态方法（特殊情况下不需要声明），它要求类的实例作为第一个参数。剩余的参数传递到对象构造表达式（调用到类），__new__ 返回值应该是一个新的对象实例（通常是 *cls* 的实例）。

典型的实现方式是创建一个类的新实例，通过使用具有适当参数的 super(currentclass，cls).__new__(cls[,...]) 调用超级类的__new__，然后根据需要，在返回之前修改新创建的实例。

如果__new__ 返回一个 cls 实例，那么新实例的方法__init__() 将会像 ``__init__(self[,...]) 被调用, 这时候，新实例和剩余的参数被传递给__new__() 相同。

如果__new__ 不返回一个 cls 实例，那么就不会调用这个新实例的__init__()。

__new__ 的主要目的是允许子类不可变类型（如整数，字符或元组）定制实例创建。也通常覆盖在自定义原类中来创建自定义类。

object.__init__(self[, ...])
实例在创建（通过 __new__ 创建）之后才会被调用，但之前返回给调用者。构造函数的参数是指传递表达式。如果一个基本类具有方法__init__()或派生类的方法 __init__()，必须显式的调用它，以确保实例部分的基本类初始化；比如: BaseClass.__init__(self, [args...])。

因为__new__ 和__init__ 同时作用来构建对象（__new__ 创建它，__init__ 来定制），没有 non-'None' 值可能通过__init__ 来返回，这样做会导致在运行时上报 Typeerror 。

object.__del__(self)
在实例被删掉时被调用，也叫作析构器。 如果其基类也具有 del() 方法，继承类应该在其 del() 显式地调用它，以保证适当地删掉它的父类部分。注意，在 del() 通过创建新的引用来推迟删除操作是允许的，但这不是推荐的做法。它然后在最后这个引用删除时被调用。不能保证在解释器退出时，仍存在的对象一定会调用 del() 方法。

注意： del x 不直接调用 x.del() —— 前者将引用计数减一，而后者仅仅在引用计数减到零时才被调用。有一些经常使用的方法。 可以防止引用计数减为零：对象之间的循环引用（例如，一个双链表或一个具有父结点和子结点指针的树形数据结构）；对某函数堆栈结构上的引发异常的对象进行引用（跟踪回溯对象保存在 sys.exc_info()[2] 是以保持其有效）；或者在交互模式下对某函数堆栈上的引发了没有处理器的异常的对象做引用（跟踪回

溯对象保存在 sys.last_traceback 中以保持其有效）。第一种方法仅能通过显式地破坏循环才能恢复，后两种情况，可以通过将 sys.last_traceback 赋给None来恢复。仅当循环引用检测器选项被允许时（这是默认的）循环引用才能为垃圾回收机制所发觉, 但这只在没有相关的 Python 级的 del_() 方法时才会被清除。关于 del_() 方法怎样处理循环引用的进一步信息参见 gc (https://docs.python.org/3/library/gc.html#module-gc) 模块，该处具体地描述了垃圾回收。

告警：因为调用 del_() 方法的不确定性，在它执行时的异常会被忽略，而只是在 sys.stderr 打印警告信息。另外，当某模块被删除，相应的 del_() 方法调用时（例如，程序退出时），有些为 del_() 方法所引用的全局名字可能已经删除了。由于这个原因， del_() 方法应该对其外部要求保持到最小。 Python1.5 可以保证以单下划线开始的全局名字在其它全局名字被删除之前从该模块中被删除；如果没有其它对存在的全局名字的引用，这会对确定那些已导入的模块在调用 del_() 之后有那些还是有效 的时是有所帮助的。

object.__ repr__ (self)
由 repr() (https://docs.python.org/3/library/functions.html#repr) 内建函数调用或者在串转换（保留引号）时用来计算对象的“正式说明”字符串，尽可能的，这应该是一个能以相同的值重建一个对象的有效的 Python 表达式（在给定的适当有环境下），如果这不 可能的，也应该是返回一个“...某些有用的信息...”形式的字符串。返回值必须是一个字符串。如果类定义 repr() (https://docs.python.org/3/library/functions.html#repr) 而不是 __str__ () (https://docs.python.org/3/reference/datamodel.html#object.__str__) , 那么当一个“非正式”字符串表示的类的实例是必须的时候， repr() (https://docs.python.org/3/library/functions.html#repr) 也会被使用。

本函数典型地用法是用于调试，所以这个串表述成详尽并无歧义是十分重要的。

object.__ str__ (self)
由 str(object) (https://docs.python.org/3/library/stdtypes.html#str) 内建函数调用，或由 format() (https://docs.python.org/3/library/functions.html#format) 和 print() (https://docs.python.org/3/library/functions.html#print) 语句来计算该对象的“非正式”串描述。返回值必然是个字符串 *string* (https://docs.python.org/3/library/stdtypes.html#textseq) 对象。

这与 object.__ repr__ () (https://docs.python.org/3/reference/datamodel.html#object.__repr__) 是不同的，因为它不 要求必须为一个有效的 Python 表达式：可以采用一个更通俗或更简洁的表述方式。

这种典型的实现方法可以通过内建类型 object (https://docs.python.org/3/library/functions.html#object) 调用 object.__ repr__ () (https://docs.python.org/3/reference/datamodel.html#object.__repr__) 来定义。

object.__ bytes__ (self)
由 format() (https://docs.python.org/3/library/functions.html#format) 内建函数（和扩展，类 str (https://docs.python.org/3/library/stdtypes.html#str) 的 strformat() (https://docs.python.org/3/library/stdtypes.ht

ml#str.format) 方法) 调用，它产生一个“格式化”对象的字符串表示。 format_spec 参数是一个包含了要求格式化选项描述的字符串。 format_spec 参数的解释是实现 __format__ () (https://docs.python.org/3/reference/datamodel.html#object.__format__) ，然而大多数类代表格式化内置的类型之一，或者使用类似的格式化的语法。

详见 Format Specification Mini-Language (https://docs.python.org/3/library/string.html#formatspec) 标准格式化特征的描述。

它的返回值必须是一个字符串对象。

在版本*3.4*中的改变：如果传递一个空的字符串， 对象 本身的__ format__ 方法会提出类型错误 TypeError (https://docs.python.org/3/library/exceptions.html#TypeError) 。

object.__ lt__ (self,other)
object.__ le__ (self,other)
object.__ eq__ (self,other)
object.__ ne__ (self,other)
object.__ gt__ (self,other)
object.__ ge__ (self,other)
它们称为“厚比较”方法集, 具体的方法名和相应的运算符的对应关系如下： x<y 调用 x.__ l t__ (y) ， x<=y 调用 __le__`(y)`，`x==y` 调用 __eq__ (y) ， x! =y 调用 __ne__`(y)`，`x>y` 调用 __g t__ (y) ， x>=y 调用 ``__ge__ (y) 。

如果它对给定的参数对没有实现操作, 一个厚比较方法可以返回 NotImplemented 。按照惯例，一个成功的比较会返回 False 和 True 。不论如何，这些方法可以返回任何值，因此如果比较操作在布尔型文本中使用 (如， i f 条件语句) ，Python 将在值上调用 bool() (https://docs.python.org/3/library/functions.html#bool) 来决定结果是真还是假。

有隐含的比较运算符之间的关系。 x==y 为真相并不意味着 x!=y 为假。相应的，当定义 __eq__ () (https://docs.python.org/3/reference/datamodel.html#object.__eq__) 时，同时也应该定义 __ne__ () (https://docs.python.org/3/reference/datamodel.html#object.__ne__) ，比便于达到预期的操作。见支持自定义比较和可作为字典关键字使用的创建 hashable (https://docs.python.org/3/glossary.html#term-hashable) 的 __hash__ () (https://docs.python.org/3/reference/datamodel.html#object.__hash__) 。

对于这些方法是没有互换参数)版本的 (在左边参数不支持该操作, 但右面的参数支持时使用) 。虽然, __lt_ _()](https://docs.python.org/3/reference/datamodel.html#object.__lt__) 和 [__gt__ ()](https://docs.python.org/3/reference/datamodel.html#object.__gt__)，[__le__ ()](https://docs.python.org/3/reference/datamodel.html#object.__le__) 和 [__ge __()](https://docs.python.org/3/reference/datamodel.html#object.__ge__)，

[__eq__ ()](https://docs.python.org/3/reference/datamodel.html#object.__eq__)和 [__ne__` () 看起来是反函数。

厚比较方法的参数并不是非要有。

从一个根操作自动生成命令的操作，见 functools.total_ordering() (https://docs.python.org/3/library/functools.html#functools.total_ordering) 。

object.__ hash__ (self)
由内建函数 hansh() (https://docs.python.org/3/library/functions.html#hash) 调用，用于操作散列的集合，包括 set (https://docs.python.org/3/library/stdtypes.html#set) ， frozenset (https://docs.python.org/3/library/stdtypes.html#frozenset) ，以及 dict (https://docs.python.org/3/library/stdtypes.html#frozenset) . __hash__ () (https://docs.python.org/3/reference/datamodel.html#object.__hash__) ，应该返回一个整数。 仅有一个要求，具有相同的值的对象应该有相同的散列值，应该考虑以某种方式将散列值与在对象中比较中起作用的部分联系起来（例如，用排斥或）。

注意： hash() (https://docs.python.org/3/library/functions.html#hash) 将返回值从一个对象的自定义 __hash__ () (https://docs.python.org/3/reference/datamodel.html#object.__hash__) 方法返回到 Py_ssize_t 的大小。这通常是8个字节64位构建和4个字节32位构建。如果对象的 __hash__ () (https://docs.python.org/3/reference/datamodel.html#object.__hash__) 互相操作构建不同的位，要确保所支持的构建宽度，一种简单的确认方法是 python -c "improt sys;print(sys.hash_info.width)" 。

如果一个类不定义 __eq__ () (https://docs.python.org/3/reference/datamodel.html#object.__eq__) 方法，它也不应该定义 __hash__ () (https://docs.python.org/3/reference/datamodel.html#object.__hash__) 操作；如果它定义 __eq__ () (https://docs.python.org/3/reference/datamodel.html#object.__eq__) 而不定义 __hash__ () (https://docs.python.org/3/reference/datamodel.html#object.__hash__) ，它的实例将不能被用作散列集合的项。如果类定义可变对象并且实现了 __eq__ () (https://docs.python.org/3/reference/datamodel.html#object.__eq__) 方法, 它就不应该实现 __hash__ () (https://docs.python.org/3/reference/datamodel.html#object.__hash__) ，因为字典实现一个散列表的键值是不可变的（如果对象的散列值改变了，它会放在错误的散列表位置中）。

如果一个类覆盖 __eq__ () (https://docs.python.org/3/reference/datamodel.html#object.__eq__) 并且没有定义 __hash__ () (https://docs.python.org/3/reference/datamodel.html#object.__hash__) ，这个类的 __hash__ () (https://docs.python.org/3/reference/datamodel.html#object.__hash__) 默认设为 None 。当一个类的 __hash__ () (https://docs.python.org/3/reference/datamodel.html#object.__hash__) 方法是 None ，当程序试图检索散列值时，类的实例将报告 TypeError (https://docs.python.org/3/library/exceptions.html#TypeError) ，当检查 `isinstance(obj,collections.Hashable)时，它也可以正确的识别。

如果一个类覆盖 __eq__ () (https://docs.python.org/3/reference/datamodel.html#object.__eq__) 并且需要从父类中保留实现，通过 __hash__` = <ParentClass>.__` hash__ 设置可以明确的告诉翻译器。

如果一个类没有覆盖 __eq__ () (https://docs.python.org/3/reference/datamodel.html#object.__eq__) 并且希望废止 hash 支持，在类定义中应该包含 ``__hash__ = None 。一个类定义自己的 __hash__ () (https://docs.python.org/3/reference/datamodel.html#object.__hash__)，明确提出了一个 TypeError (https://docs.python.org/3/library/exceptions.html#TypeError)，将错误的认定为 isinstance(obj,collection.Hashable) 调用。

注意：默认情况下， __hash__ () (https://docs.python.org/3/reference/datamodel.html#object.__hash__) 字符， 字节和日期时间对象值是“有经验的”，伴随着一个不可预测的随机值。虽然它们在一个 Python 的过程中保持不变，但是在 Python 之间调用是不可预测的。
这是旨在提供一种反对拒绝服务的保护机制。更多细节见 http://www.ocert.org/advisories/ocert-2011-003.html。
散列值变化影响字典,集和其他映射的迭代顺序。Python 从来没有给这种顺序提供保护（典型变化在32位和64位构建时）。
同时参考 PYTHONHASHSEED (https://docs.python.org/3/using/cmdline.html#envvar-PYTHONHASHSEED) 。
版本3.3 以改变： 默认集成散列随机化。

object.__ bool__ (self)
用于实现真值测试盒构建操作 bool()，返回 假 或 真 。如果没有定义这个方法，调用 __len__ () (https://docs.python.org/3/reference/datamodel.html#object.__len__)，如果定义了，它的结构为非零，那么这个对象应该被当做真。如果一个类既没有定义 __len__ () (https://docs.python.org/3/reference/datamodel.html#object.__len__)，也没有定义 __bool()__ (https://docs.python.org/3/reference/datamodel.html#object.__bool__)，所有它的实例都认为是真。

定制属性访问

可以定义以下方法用于定制类实例属性的访问的含义（用于赋值，或删除 x.name ）。

object.__ getattr__ (self, name)
当以正常的方式的访问属性（就是说,要访问的属性既不是实例的属性,也在其所在的类树结构中找不到）。 name 是属性名。方法应该返回一个属性值，或抛出一个 AttributeError (https://docs.python.org/3/library/exceptions.html#AttributeError) 异常。

注意如果属性可以通过正常的机制访问，getattr() (https://docs.python.org/3/reference/datamodel.html#object.__getattr__) 不会被调用（是故意将 getattr() (https://docs.python.org/3/reference/datamodel.htm

l#object.__getattr__) 和 setattr() (https://docs.python.org/3/reference/datamodel.html#object.__setattr__) 设置成不对称的），这样的原因是由于效率并且 getattr() (https://docs.python.org/3/reference/datamodel.html#object.__getattr__) 不能访问实例的其它属性。注意，至少对于实例变量来说，你可能通过不往实例字典里插入任何值来伪装所有控制（但将它们插入到其它对象中）。在下述的 `__getattribute__` () (https://docs.python.org/3/reference/datamodel.html#object.__getattribute__) 方法中获取一个完全控制属性访问的方法。

object.`__getattribute__`(self, name)
无条件实现访问类属性时调用。如果类也定义为 getattr() (https://docs.python.org/3/reference/datamodel.html#object.__getattr__)，除非使用 `__getattribute__` () (https://docs.python.org/3/reference/datamodel.html#object.__getattribute__) 才能调用，或抛出一个 AttributeError (https://docs.python.org/3/library/exceptions.html#AttributeError) 异常。方法应该返回一个属性值，或抛出一个 AttributeError (https://docs.python.org/3/library/exceptions.html#AttributeError) 异常。为了避免这种方法中无限递归，其实现基本类调用方法应该使用相同的名称来访问任何属性，比如，`object.__getattribute__(self, name)`。

注意：查找特殊方法时，该方法仍有可能通过语言语法或内建函数作为隐含的调用的结果绕过该方法。见 Special method lookup (https://docs.python.org/3/reference/datamodel.html#special-lookup)。

object.`__setattr__`(self, name, value)
在属性将被赋值时调用。这是作为正常机制的代替使用的（就是地实例字典中存储值）。*name* 是实例值，*vaule* 是要赋的值。

如果在 `__setattr__` () (https://docs.python.org/3/reference/datamodel.html#object.__setattr__) 内部试图为一个实例属性赋值，应该同时调用基本类方法，如 `object.__setattr__(self, name, value)`。

object.`__delattr__`(self, name)
就像 `__setattr__` () (https://docs.python.org/3/reference/datamodel.html#object.__setattr__) 一样，不过其作用是删除而不是赋值。仅仅对于对象用 `del obj.name` 实现才有意义.

object.`__dir__`(self)
对象调用时用 dir() (https://docs.python.org/3/library/functions.html#dir) 调用，必须返回顺序，它将返回的顺序转换为列表和排序。

实现描述符

下列方法仅适用于一个包含方法类的实例（所谓的描述符类）出现在一个所有者类中（描述符必须在主类的字典中或者在其一个父类的类字典中）。下述的例子中，“属性”指的是其名字所有者类 `__dict__` 属性的关键属性。

object.__get__(self, instance, owner)
获得所有者类或者属性（类访问属性）或那个类实例（实例访问属性）时调用。所有者总是所有者类，而对于实例，它可以访问实例的属性，或通过所有者访问属性为 None。这种方法应该返回一个（计算）属性值或抛出属性异常 AttributeError (https://docs.python.org/3/library/exceptions.html#AttributeError)。

object.__set__(self, instance, value)
对所有者类的*实例*，将实例属性设一个新值及值时调用。

object.__delete__(self, instance)
对所有者类的*实例*，将实例属性删除时调用。

inspect (https://docs.python.org/3/library/inspect.html#module-inspect) 模块解释__object__ 属性，并指向定义该对象的类（适当设置这个可以在运行时协助动态类属性自查）。对于可调用的，它可能表明给丁磊的一个实例（或一个子类），预计或需要作为第一位参考（比如，CPython 设置这个方法的属性用C语言实现）。

调用描述符
一般而言，描述符是一个具有“绑定行为”的对象属性，这些访问属性在描述符协议里通过方法来覆盖：__get__() (https://docs.python.org/3/reference/datamodel.html#object.__get__)，__set__() (https://docs.python.org/3/reference/datamodel.html#object.__set__)，__delete__() (https://docs.python.org/3/reference/datamodel.html#object.__delete__)。如果一个对象定义了任何一个这种方法，那么就称为个描述符。

属性访问的默认行为是从一个对象的字典中获取、设置、或者删除属性。比如，a.x 是个起始为 a.__dict__['x'] 的查询链，然后是 type(a).__dict__['x']，紧接着通过 type(a) 基本类型，不包括元类。

不论如何，如果查询值是一个对象定义的描述符方法，那么 Python 覆盖默认行为，用调用描述符方法替代。这个在链中发生的优先级取决于定义何种描述方法和如何调用它们。

描述符调用的起点是个“绑定”，a.x。参数如何装配取决于 a：

直接调用
最简单最常见的调用是用户代码直接调用一个描述符方法：x.__get()__(a)。

实例绑定
如果绑定到一个类实例，a.x 转化为调用：type(a).__dict__['x'].__get__(a, type(a))。

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

超级绑定

如果 a 是一个超级 super (https://docs.python.org/3/library/functions.html#super) 实例，在基本类 A 之前立即 B，绑定 super(B, obj).m() 查找 obj.__class__.__mro__，然后调用描述符：A.__dict__['m'].__get__(obj, obj.__class__)。

对于实例绑定，描述符的优先调用取决于该描述符定义方法，一个描述符可以定义成 __get__() (https://docs.python.org/3/reference/datamodel.html#object.__get__)，__set__() (https://docs.python.org/3/reference/datamodel.html#object.__set__)，和 __delete__() (https://docs.python.org/3/reference/datamodel.html#object.__delete__) 的任何组合。如果没有定义 __get__() (https://docs.python.org/3/reference/datamodel.html#object.__get__)，那么访问对象本身的属性将返回描述符，除非有一个值在对象的实例的字典中。如果描述符同时或者选择定义 __set__() (https://docs.python.org/3/reference/datamodel.html#object.__set__) 及 __delete__() (https://docs.python.org/3/reference/datamodel.html#object.__delete__)，那么它是个数据描述符；反之，就不是一个数据描述符。一般情况下，数据描述符同时定义 __set__() (https://docs.python.org/3/reference/datamodel.html#object.__set__) 及 __delete__() (https://docs.python.org/3/reference/datamodel.html#object.__delete__)，相比非数据描述符仅仅只有 __get__() (https://docs.python.org/3/reference/datamodel.html#object.__get__) 方法。具有 __set__() (https://docs.python.org/3/reference/datamodel.html#object.__set__) 及 __get__() (https://docs.python.org/3/reference/datamodel.html#object.__get__) 的数据描述符总是在实例字典中覆盖重新定义。

Python 的实现方法（包括 staticmethod() (https://docs.python.org/3/library/functions.html#staticmethod) 和 classmethod() (https://docs.python.org/3/library/functions.html#classmethod)）都是作为非数据描述符，相应地，实例可以再定义和覆盖方法。这允许单个实例得到不用于其他相同类的实例。

property() (https://docs.python.org/3/library/functions.html#property) 功能是用于实现一个数据描述符。相应地，实例不能覆盖一个属性的行为。

__slots__

默认情况下,类的实例都有一本字典来存储属性，这对对象浪费空间有很少的实例变量。当创建大量实例时，空间消耗会变的很严重。

在类定义中可以重新定义 __*slots*__ 来覆盖默认值。__*slots*__ 声明实例变量序列，在每个实例变量中储备足够的空间来保有变了值。由于每个实例不创建 __*slots*__，可以节省空间。

object.__slots__

这类变量可以被分配一个可迭代字符串，或与所使用的变量名的字符串序列实例。__*slots*__ 储备空间声明的变量和防止自动为每个实例创建 __dict__ 和 __weakref__。

使用 __*slots*__ 注意事项

继承一个没有 __*slots*__ 的类是，总是可以访问类的 __*slots*__ 属性，因此在子类中定义 __*slots*__ 是没有意义的。

没有一个 __*slots*__ 变量，不能分配实例新变量不在 __*slots*__ 定义中，如果试着去分配，那么会抛出异常 AttributeError (https://docs.python.org/3/library/exceptions.html#AttributeError) 。如果新变量需要动态分配，那么在 __*slots*__ 声明中添加 __dict__ 字符串顺序。

每个实例变量没有 __*weakref*__ ，类定义 __*slots*__ 不支持若引用到它的实例。如果需要若引用，那么在 __*slots*__ 声明中添加 __weakref__ 字符串顺序。

对于每一个变量，在类的等级上创建描述符来（ Implementing Descriptors (https://docs.python.org/3/reference/datamodel.html#descriptors) ）实现 __*slots*__ 。因此，类属性不能将实例变量的默认值设置为 __*slots*__ ；否则，类属性将覆盖描述符分配。

__*slots*__ 声明仅限于类定义。因此，除非子类定义 __*slots*__ ，否则将会有一个 __*dict*__ （必须只包含额外的插槽的名字）。

如果一个类在基本类中定义了插槽，基本类的实例变量定义的位置不可访问（除了从基本类中检索它的描述符）。这使得未被定义的程序具有意义。在未来，可能添加检查来防止这个。

对于诸如从可“变成”的内建类型 int (https://docs.python.org/3/library/functions.html#int) ，bytes (https://docs.python.org/3/library/functions.html#bytes) 和 tuple (https://docs.python.org/3/library/stdtypes.html#tuple) 继承的类，非空 __*slots*__ 不起作用。

任何一个非字符串的迭代器可能分配到 __*slots*__ ；映射也可以使用；但在未来，可能给每个键分配特殊的值。

只有2个类具有相同的 __*slots*__ ， __*class*__ 分配才起作用。

自定义创建类

默认情况下，使用 type() (https://docs.python.org/3/library/functions.html#type) 构建类。以一个新的命名空间和名字来运行类的主体，本地绑定结果到 type(name,bases,namespace) 。

在类定义行，通过关键参数 metaclass 可以自定义创建类，或者从一个已存在的包含参数的类中继承。在下面的例子中， MyClass 和 MySubclass 都是 Meta 的实例。

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

任何其他类定义中指定的关键字参数传递到所有元类操作描述如下。

当执行一个类定义时，会发生如下步骤：

- 确定适当的元素
- 准备类空间名
- 执行类主体
- 创建类对象

定义适当的元类

适当元类的定义如下：

- 如果没有给出基本、没有显式的元类，使用 type（）(https://docs.python.org/3/library/functions.html#type)
- 如果给出显式的元类，没有 type（）(https://docs.python.org/3/library/functions.html#type) 实例，直接使用它作为元类
- 如果给出 type（）(https://docs.python.org/3/library/functions.html#type) 实例作为显式的元类，或者定义了基本类，大多使用派生的元类

从显式指向的元类（如果有的话）和详细说明的基本元类（如 `type(cls)`）中选择最派生元类。最派生元类是所有候选元类的子类中的一个。如果没有元类符合标准，类定义将会抛出 `TypeError`。

类命名空间准备

一旦确定了适当的元类，那么类空间名也会准备好。如果元类具有 `__prepare__` 属性，它被调用为 `namespace = metaclass.__prepare__(name, bases, **kwds)`（如果有来自类定义的额外关键参数）。

如果元类没有 `__prepare__` 属性，类名门空间初始化为一个空 dict() (https://docs.python.org/3/library/stdtypes.html#dict) 实例。

也可以参考：

PEP 3115 – 在 Python 3000 中的元类

介绍了 `__prepare__` 命名空间挂钩

执行类主体

执行类主体（大约）是 `exec(body, globals(), namespace)`。从一个正常的调用到 exec() (https://docs.python.org/3/library/functions.html#exec) 的主要区别是类定义发生在函数里时，exec() (https://docs.python.org/3/library/functions.html#exec) 允许类主体（包括任何方法）来引用当前来自内外部范围的名称。

然而，即使类定义发生在函数里时，在类里定义方法仍不能明确在类范围内定义的名字。类变量必须通过实例或类方法的第一个参数，无法访问所有的静态方法。

创建类对象

一旦通过执行类主体填充了类命名空间，通过调用 `metaclass(name, bases, namespace, **kwds)` 创建类对象（这里传递的关键字相同于传递到 `__prepare__` ）。

这个类对象通过无参数形式 super() (https://docs.python.org/3/library/functions.html#super) 来引用，如果在类主体中的任何方法参考 `__class__` 或者 `super` ，编译器会创建一个隐式封闭引用 `__class__` 。这使得零参数形式的 super() (https://docs.python.org/3/library/functions.html#super) 正确识别字典范围定义的类，同时使用类或实例去做当前调用，并识别基于由第一个参数传递到方法的调用。

在创建类对象后，它是传递给在类中定义（如果有）的解释器，以及随之而来的在本地命名空间定义类的对象绑定。

也可以参考：

PEP 3135 – 新超级类

描述了 `__class__` 隐式封闭引用

元类实例

元类的潜在使用时无穷无尽的。一些想法，探讨包括日志，借口检查，自动授权，自动属性创建，代理，框架，以及自动资源锁定/同步。

这里有个元类的示例，它使用了 collections.OrderedDict (https://docs.python.org/3/library/collections.html#collections.OrderedDict) 来记忆类变量定义的顺序：

```
class OrderedClass(type):

    @classmethod
    def __prepare__(metacls, name, bases, **kwds):
        return collections.OrderedDict()

    def __new__(cls, name, bases, namespace, **kwds):
        result = type.__new__(cls, name, bases, dict(namespace))
        result.members = tuple(namespace)
        return result

class A(metaclass=OrderedClass):

    def one(self): pass
    def two(self): pass
    def three(self): pass
```

```
    def four(self): pass

->>>A.members
(`__module__`, `one`, `two`, `three`, `four`)
```

当类定义 *A* 得到执行，这一过程开始于调用元类的`__prepare__()`方法，并返回一个 collections.OrderedDict (https://docs.python.org/3/library/collections.html#collections.OrderedDict) 。映射记录 *A* 的方法和属性，它们的类主体语句中定义。一旦执行这些定义就会完全填充命令字典，元类的 `__new__()` (https://docs.python.org/3/reference/datamodel.html#object.__new__) 方法被调用。该方法构建的新类型和它在属性中保存命令字典键称为 `members` 。

自定义实例和子类检查

以下方法用来覆盖内建函数 isinstance() (https://docs.python.org/3/library/functions.html#isinstance) 和 issubclass() (https://docs.python.org/3/library/functions.html#issubclass) 的默认行为。特别是元类 abc.ABCMeta (https://docs.python.org/3/library/abc.html#abc.ABCMeta) 实现这些方法，目的是为了允许抽象基本类（ABCs）作为任何类或类型（包括内建类型）的“虚拟基本类”，包括其他的抽象基本类。

`class.__instancecheck__(self, instance)`
如果考虑*实例*应该是类的（直接或间接）实例，返回真。如果定义了它，调用实现 `isinstance(instance, class)` 。

`class.__subclasscheck__(self, subclass)`
如果考虑*子类*应该是类的（直接或间接）子类，返回真。如果定义了它，调用实现 `issubclass(subclass, class)` 。

注意，在一个类的类型（元类）上查询这些方法，在真实类里，它们不能定义成类方法。这是符合实例上调用特殊方法查找的，也只有在这种情况下，实例本身就是一个类。

也可以参考：

PEP 3119 – 介绍抽象基本类
详细介绍了通过 `__instancecheck__()` (https://docs.python.org/3/reference/datamodel.html#class.__instancecheck__) 和 `__subclasscheck__()` (https://docs.python.org/3/reference/datamodel.html#class.__subclasscheck__) ，自定义 isinstance() (https://docs.python.org/3/library/functions.html#isinstance) 和 issubclass() (https://docs.python.org/3/library/functions.html#issubclass) 的行为，这个功能推动了上下文添加抽象基本类（见 abc (https://docs.python.org/3/library/abc.html#module-abc) 模块）的语言。

模拟包装类型

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

当实例像一个函数使用时调用本方法。如果定义了这个方法，那么 x（arg1， arg2，...） 是 x. call（arg1，arg2， ...） 的缩写形式。

模拟包装器类型

定义以下方法可以实现包容器对象。包容器通常指有序类型（像列表或元组）或映射（像字典），但也可以表示其它包容器。 第一个方法集用于模拟有序类型或映射；有序类型的区别就在于，允许键可以是整数 *k*，其中 0 < k < N ，*N* 是有序类型的长度，或者是描述了一定范围的片断。在实现映射时，推荐提供 keys()，values()，items()，has key()，get()，clear()，copy()，和 update()，使其行为类似于 Python 标准的字典对象；collections (https://docs.python.org/3/library/collections.html#module-collections) 模块提供了一个 MutableMapping (https://docs.python.org/3/library/collections.abc.html#collections.abc.MutableMapping) 抽象基类来从 __getitem__ () (https://docs.python.org/3/reference/datamodel.html#object.__getitem__) ， __setitem__ () (https://docs.python.org/3/reference/datamodel.html#object.__setitem__) ， __delitem__ () (https://docs.python.org/3/reference/datamodel.html#object.__delitem__) ，和 keys() 抽象基本类集合中创建这些方法。可变的有序类型应该提供方法append()，count()，index()，insert()，pop()，remove()，reverse() 和sort()，像Python标准的列表类型。最后，有序类型应该通过定义下述的方法 __add__ () (https://docs.python.org/3/reference/datamodel.html#object.__add__) ， __radd__ () (https://docs.python.org/3/reference/datamodel.html#object.__radd__) ， __iadd__ () (https://docs.python.org/3/reference/datamodel.html#object.__iadd__) ， __mul__ () (https://docs.python.org/3/reference/datamodel.html#object.__mul__) ， __rmul__ () (https://docs.python.org/3/reference/datamodel.html#object.__rmul__) 和__ imul__ () (https://docs.python.org/3/reference/datamodel.html#object.__imul__) 实现加法运算（就是指连接）和乘法运算（指重复）。它们不应该定义其它数值运算操作。对于有序类型和字典都推荐实现 __contains__ () (https://docs.python.org/3/reference/datamodel.html#object.__contains__) ，以便于高效的使用 in 运算符；对于映射， in 应该搜索映射值；对于有序类型，应该通过值来搜索。进一步推荐映射和有序类型类型通过容器有效迭代去实现__ iter__ () (https://docs.python.org/3/reference/datamodel.html#object.__iter__) ，对于映射， __iter__ () (https://docs.python.org/3/reference/datamodel.html#object.__iter__) 应该等价于 keys() 方法；对于有序类型，通过值来迭代。

object.__ len__ (self)

实现内建函数 len() (https://docs.python.org/3/library/functions.html#len) 相仿的功能，应该返回对象的长度，并且返回一个 >= 0的整数。另外，一个没有定义 `__bool_ _（ ） (https://docs.python.org/3/reference/datamodel.html#object.__bool__) 的方法返回0被认为是返回一个逻辑假值。

object.__length_hint__(self)
调用实现 operator.length_hint() (https://docs.python.org/3/library/operator.html#operator.length_hint) 。应该返回估算长度的对象(可能是大于或小于实际长度)。长度必须是一个整数 >= 0。这纯粹是一种优化方法,从来没有要求正确性。

新的在版本3.4.

注意：用以下三种方法完全完成切断，调用像

a[1:2] = b

翻译成

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

等等。缺失的部分项用 None 来填充。

object.__getitem__(self, key)
实现 self[key] 相仿的功能。对于有序类型，可接受的键包括整数和片断对象。注意对负数索引（如果类希望模拟有序类型）的特殊解释也依赖于 getitem() 方法。如果*键*是不合适的类型，一个 TypeError (https://docs.python.org/3/library/exceptions.html#TypeError) 异常就会被抛出，如果某个值在有序类型的索引值集合之外（在任何负值索引的特定解释也不能行的通的情况下），会抛出一个 IndexError (https://docs.python.org/3/library/exceptions.html#IndexError) 的异常。

注意： for 循环可以通过对由于对无效索引值而抛出的 IndexError 异常进行捕获来对访问有序类型的结尾做适当地检测。

object.__missing__(self, key) 调用 dict.__getitem__() (https://docs.python.org/3/reference/datamodel.html#object.__getitem__) 实现 self[key]，当关键字不在子电子类时。

object.__setitem__(self, key, value)
在对 self[key] 进行赋值时调用。与__getitem__() (https://docs.python.org/3/reference/datamodel.html#object.__getitem__) 有着相同的注意事项。通常只对映射实现本方法，并且要求对象支持改变键所对应的值，或支持增加新键；也可以在有序类型中实现，此时支持单元可以替换。在使用无效的*键值*时，会抛出与https://docs.python.org/3/reference/datamodel.html#object.getitem相同的异常。

object.__delitem__(self, key) 在删除 self[key] 时调用，与 __getitem__() (https://docs.python.org/3/reference/datamodel.html#object.__getitem__) 有着相同的对象。本方法通常仅仅在映射中实现，并且对象支持键的删除；也可以在有序类型中实现，此时单元可以从有序类型删除。在使用无效的*键值*时，会抛出与 __getitem__() (https://docs.python.org/3/reference/datamodel.html#object.__getitem__) 相同的异常.

object.__iter__ (self)
要求使用包容器的子迭代时，这个方法被调用。本方法应该返回一个可以迭代包容器所有对象的迭代子对象。对于映射，应该在键的基础上进行迭代，

迭代子对象也需要实现这个方法，它们应该返回它自己。对于更多的关于迭代子对象的信息，参见 *Iterator Types* (https://docs.python.org/3/library/stdtypes.html#typeiter) 。

object.__reversed__ (self)
调用 reversed() (https://docs.python.org/3/library/functions.html#reversed) 实现反向迭代。它应该返回一个新的迭代器对象，以相反的顺序遍历所有的对象容器。

如果不支持 reversed() (https://docs.python.org/3/library/functions.html#reversed) 方法，reversed() (https://docs.python.org/3/library/functions.html#reversed) 将会求助于使用顺序协议（ __len__ () (https://docs.python.org/3/reference/datamodel.html#object.__len__) 和 __getitem__ () (https://docs.python.org/3/reference/datamodel.html#object.__getitem__) ）。支持序列协议的对象应该只提供 reversed() (https://docs.python.org/3/library/functions.html#reversed) ，除非它们能提供一个比 reversed() (https://docs.python.org/3/library/functions.html#reversed) 更有效的实现方法。

成员测试运算符（ in (https://docs.python.org/3/reference/expressions.html#in) 和 not in (https://docs.python.org/3/reference/expressions.html#not-in) ）一般通过对有序类型的迭代来实现。但是包容器也可以提供以下方法得到更有效的实现，不要对象是有序类型。

object.__contains__(self, item) 使用成员测试运算符时调用。如果 *item* 在 *self* 中，返回true；否则返回 false。对于映射对象，比较应该在键上进行，不应该是键值对。

对于没有定义 __contains__ () (https://docs.python.org/3/reference/datamodel.html#object.__contains__) 的对象，成员第一次迭代测试会通过 __iter__ () (https://docs.python.org/3/reference/datamodel.html#object.__iter__) ，那么旧的顺序迭代协议会通过__ getitem__ () (https://docs.python.org/3/reference/datamodel.html#object.__getitem__) ，见 this section in the language reference (https://docs.python.org/3/reference/expressions.html#membership-test-details) 。

模拟数值类型

以下方法用于模拟数值类型。其中，对于有些种类数值所不支持的操作对应的方法未定义（如，对非整数值的位运算）。

```
object.__add__(self,other)
object.__sub__(self,other)
object.__mul__(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)
```

这些方法用于实现二元算术运算（ + ， - ， * ， / ， // ， % ，divmod() (https://docs.python.org/3/library/functions.html#divmod) ，powe() (https://docs.python.org/3/library/functions.html#pow) ， ** ， << ， >> ， & ， ^ ， | ）。比如，对表达式 x + y 求值，*x* 是一个具有 __add__ (https://docs.python.org/3/reference/datamodel.html#object.__add__) 方法的类的实例，调用 x.__ add__ (y) 。 __divmod()__ (https://docs.python.org/3/reference/datamodel.html#object.__divmod__) 方法应该相当于使用 __floordiv__ () (https://docs.python.org/3/reference/datamodel.html#object.__floordiv__) 和 __mod__ () (https://docs.python.org/3/reference/datamodel.html#object.__mod__) ，它跟__ truediv__ () (https://docs.python.org/3/reference/datamodel.html#object.__truediv__) 不相关。注意如果内建函数 powe() (https://docs.python.org/3/library/functions.html#pow) 支持三值版，应该定义 ___ pow__ () (https://docs.python.org/3/reference/datamodel.html#object.__pow__) 来接受第三个参数。

如果这些方法其中之一不支持提供的参数运算，它应该返回 NotImplementd 。

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

这些方法用于实现反应（交换）操作数二元算术运算（ + ， - ， * ， / ， // ， % ，divmod() (https://docs.python.org/3/library/functions.html#divmod) ，powe() (https://docs.python.org/3/library/functions.html#pow) ， ** ， << ， >> ， & ， ^ ， | ）。

只有在左操作数不支持相应操作和操作数类型不同时才会调用。[2] 比如，表达式求值 x - y ，*y* 是一个具有 __r

sub__ () (https://docs.python.org/3/reference/datamodel.html#object.__rsub__) 方法的类的实例，如果 x.__sub__(y) 返回 *NotImplemented* 调用 y.__rsub__(x) 。

注意，三元函数 powe() (https://docs.python.org/3/library/functions.html#pow) 将不会尝试调用 __pow__ () (https://docs.python.org/3/reference/datamodel.html#object.__pow__) （强制规则会变的国过于复杂）。

注意：如果右操作数的类型是左操作数的类型的子类,子类提供了反映操作方法,这个方法将被称为前左操作数的 non-reflected 方法，这个动作允许子类覆盖他们的父类操作。

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

这些方法用于实现赋值运算符（ +=， -= ， *= ， /= ， %= ， **= ， <<= ， >>= ， &= ， &= ， |= ）。这些方法应该试图做就地操作（修改 *self* ）和返回结果（可能是，但不需要 *self* 操作），如果没有具体定义一个方法，赋值操作就返回到正常的方法。比如，*x* 是一个具有 __add__ (https://docs.python.org/3/reference/datamodel.html#object.__add__) 方法的类的实例， x += y 相当于 x = x.__iadd__(y) ，否则应当考虑 x.__add__(y) 和 y.__radd__(x) 成为 x + y 的求值。在某些特定情况下，参数赋值会产生意想不到的错误（见 Why does a_tuple[i] += [‘item’] raise an exception when the addition works? (https://docs.python.org/3/faq/programming.html#faq-augmented-assignment-tuple-error) ），但实际上这种行为也是数据模型的一部分。

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

调用这些方法实现一元算术运算（ - ， + ，abs() (https://docs.python.org/3/library/functions.html#abs) 和 ~ ）。

```
object.__complex__(self)
object.__int__(self)
```

```
object.__float__(self)
object.__round__(self[, n])
```

调用这些方法实现内建函数 complex() (https://docs.python.org/3/library/functions.html#complex)，int() (https://docs.python.org/3/library/functions.html#int)，float() (https://docs.python.org/3/library/functions.html#float) 和 round() (https://docs.python.org/3/library/functions.html#round)，应该返回适当的类型值。

object.__index__ (self)
调用这个方法实现 operator.index() (https://docs.python.org/3/library/operator.html#operator.index)，和任何时候 Python 需要无损的转换数值对象为一个整数对象（比如在片断内，或内建函数 bin() (https://docs.python.org/3/library/functions.html#bin)，hex() (https://docs.python.org/3/library/functions.html#hex) and oct() (https://docs.python.org/3/library/functions.html#oct)）。该方法的存在表明，数值对象时一个整形，必须返回一个整数。

注意：为了保持一致的整形类型，在定义了 __index__ () (https://docs.python.org/3/reference/datamodel.html#object.__index__) 时，__int__() (https://docs.python.org/3/reference/datamodel.html#object.__int__) 也应该被定义，两者返回相同的值。

with 语句的上下文管理器

当执行一个 with (https://docs.python.org/3/reference/compound_stmts.html#with) 语句时，上下文管理器会定义建立运行时的上下文，它处理运行上下文的代码块的进入和退出，通常调用 with (https://docs.python.org/3/reference/compound_stmts.html#with) 语句（ *The with statement* (https://docs.python.org/3/reference/compound_stmts.html#with) 章节描述）实现这个功能，但也可以通过直接调用方法来实现。

上下文管理器的典型应用包括保存和回复各种全局状态，锁定和释放资源，关闭打开文件等。

更多关于上下文管理器的信息，见 Context Manager Types (https://docs.python.org/3/library/stdtypes.html#typecontextmanager)。

object.__enter__ (self)
进入这个对象运行时相关的上下文。如果有的话，with 语句将该方法的返回值绑定到 as 语句的子句中。

object.__exit__ (self)
退出这个对象运行时相关的上下文。参数描述导致上下文的异常退出。如果上下文是退出没有例外,三个参数都将是 None (https://docs.python.org/3/library/constants.html#None)。

如果提供的是个异常，并且方法想要停止异常（如，防止它被传播），它应该返回一个真值。否则，异常处理会退出这个方法

注意，`__exit__` () (https://docs.python.org/3/reference/datamodel.html#object.__exit__) 方法不应该再次抛出异常，这是调用者负责的。

另请参考：

PEP 0343 – with 语句

Python 中 with 语句的规范，背景，以及用例。

特殊方法查找

对于自定义类，如果在对象类型上而不是字典对象的实例上定义它，特殊方法的隐式调用只能保证正常地工作。这就是为什么下面代码会抛出个异常：

```
>>> 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()
```

这种现象的基本原理在于可以被所有对象，包括类型对象，实现的方法，诸如 `__hash__` () (https://docs.python.org/3/reference/datamodel.html#object.__hash__) 和 `__repr__` () (https://docs.python.org/3/reference/datamodel.html#object.__repr__) 。如果这些方法的隐式查找使用传统的查找过程，在调用对象类型本身时会失败：

```
>>> 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
```

试图错误地以这种方式调用一个类的非绑定方法有时被称为“元类混乱”，而且避免了避开实例查找特殊方法：

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

处理避开实例属性的正确性，即使是对象的元类，一般隐式特殊方法查找一般也避开 __getattribute__() (https://docs.python.org/3/reference/datamodel.html#object.__getattribute__) 方法：

```
>>> 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__() # 通过实例隐式查找
Class getattribute invoked
10
>>> type(c).__len__(c) # 通过类型隐式查找
Metaclass getattribute invoked
10
>>> len(c) # 隐式查找
10
```

以这种方式避开 __getattribute__() (https://docs.python.org/3/reference/datamodel.html#object.__getattribute__) 的机制在翻译器内提供了重要的优化范围，使特殊方法的处理具有了一定的灵活性。

4

执行模型

命名空间和代码块

名字 *Names* 参考对象，名字绑定操作介绍了名字，程序代码中每一个名字的产生都参考从建立在包含使用的最里面功能块名字 *bingding*。

一个代码块是一个可以作为一个单元执行的Python程序文本，像模块，类定义或函数体。每个命令类型的交互是一块。一个脚本文件（一个文件作为标准输入翻译或作为一个命令行参数指定的翻译）是个代码块，一个脚本命令（可选 ‘ -c ’ 在命令解释行指定）是个代码块。传递给内建函数 eval() (https://docs.python.org/3/library/functions.html#eval) 和 exec() (https://docs.python.org/3/library/functions.html#exec) 的字符串参数是个代码块。

代码块在*运行结构框架*内执行，一个框架包含一些管理信息（用来调试），决定执行完代码块后在哪继续执行和怎么执行。

一个*范围*定义代码块内名字的可视性，如果在一个快中定义了局部变量，它的范围就包含那个代码块，如果定义出现在一个函数块里，那么范围就扩展到任何一个包含这个定义的代码块，除非代码块引进一个不同的绑定名字。类代码块中定义名字范围仅限于类代码块；它不能扩展到方法代码块——包括理解式和生成表达式，因为它们使用一个函数实现范围。这意味着下面执行将会失败：

```
class A:

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

在代码块中使用名字时，使用最近封闭范围解决。所有这样的设置可视范围代码块的集合叫做块的*环境*。

如果一个名字绑定在代码块内，除非声明为 nonlocal (https://docs.python.org/3/reference/simple_stmts.html#nonlocal)，它的局部变量都是属于那个块的。如果一个名字绑定在模块级别，它是一个全局变量。（模块代码块变量是局部和全局。）如果一个变量没有定义却在模块中使用，那么它是*自由变量*。

当根本没发现一个名字时，抛出异常 NameError (https://docs.python.org/3/library/exceptions.html#NameError)。如果名字指向一个没有绑定的局部变量，抛出异常 UnboundLocalError (https://docs.python.org/3/library/exceptions.html#UnboundLocalError)。UnboundLocalError (https://docs.python.org/3/library/exceptions.html#UnboundLocalError) 是 NameError (https://docs.python.org/3/library/exceptions.html#NameError) 的子类。

结构绑定名字：正规参数函数，import (https://docs.python.org/3/reference/simple_stmts.html#import) 语句，类和函数定义（在代码块中定义的类或函数名字绑定），如果在分配中发生，目标就是标示符，for (https://docs.python.org/3/reference/compound_stmts.html#for) 循环首部，或者在 with (https://docs.pytho

n.org/3/reference/compound_stmts.html#with) 语句中 as (https://docs.python.org/3/reference/compound_stmts.html#as) 之后，或 except (https://docs.python.org/3/reference/compound_stmts.html#except) 子句。import (https://docs.python.org/3/reference/simple_stmts.html#import) 的形式语句 `from ... import ...*` 绑定所有导入模块中定义的名字，除非那些以下划线开始的。这种形式语句可能仅使用在模块级别。

目标发生在 del (https://docs.python.org/3/reference/simple_stmts.html#del) 语句中也考虑绑定这个目的（尽管真实的情况是不绑定名字）。

每次分配或导入语句发生在代码块内，它通过类或者函数定义或者在模块级别定义（顶端是代码块）。

如果一个名称绑定操作发生在一个代码块内，块内的所有使用的名称被视为对当前块的引用。这可能导致在一个块之前错误使用一个名字，这条规则是微妙的，Python 中在一个代码块内缺乏声明和允许名称进行绑定的操作。一个代码块的局部变量可以由扫描整个文本块的名称绑定操作。

如果 global (https://docs.python.org/3/reference/simple_stmts.html#global) 声明发生在一个代码块内，所有在声明中指定使用的名字是指绑定顶级名称空间的名字。顶级命名空间中的名字解析通过搜索全局命名空间，即模块的命名空间包含代码块，内建命名空间，内建模块 builtins (https://docs.python.org/3/library/builtins.html#module-builtins) 命名空间。先搜索全局命名空间，如果没有找到名字,继续搜索内建命名空间。global (https://docs.python.org/3/reference/simple_stmts.html#global) 声明必须优先于所有名字的声明。

内建命令名称空间相关联的一个代码块的执行实际上是发现通过它的全局名称空间查找名字 `__builtins__`，这应该是一个字典或一个模块（在后一种情况下所使用的模块的词典）。默认情况下，当在 `__main__` (https://docs.python.org/3/library/__main__.html#module-__main__) 模块中时，`__builtins__` 是内建模块 builtins (https://docs.python.org/3/library/builtins.html#module-builtins)，当在其他模块中时，`__builtins__` 是词典内键模块的别名。`__builtins__` 可以设置为用户创建字典,是创建一个限制执行的弱形式。

CPython 实现细节：用户不会接触到 `__builtins__`，它是个严格的细节实现。用户想要覆盖内建命名空间的值就应该 import (https://docs.python.org/3/reference/simple_stmts.html#import) 内建模块 builtins (https://docs.python.org/3/library/builtins.html#module-builtins) 并适当地修改它的属性。

一个模块的命名空间在第一次自动创建时总是被导入的。一个脚本的主模块总是叫 `__main__` (https://docs.python.org/3/library/__main__.html#module-__main__)。

在一个相同的代码块内，global (https://docs.python.org/3/reference/simple_stmts.html#global) 声明和名字绑定操作具有相同的范围，如果最近的封闭范围内自由变量包含一个全局声明,自由变量就会当做全局变量。

动态交互功能

Python 语句中有几种情况是在非法使用时结合最近的包含自由变量的范围。

如果在一个封闭范围中引用一个变量,删除这个名字是非法的。在编译时将产生一个错误报告。

如果函数和函数包含中使用导入 `improt*` 通配符形式，或者是最近的一个自由变量块,编译器将抛出异常 SyntaxError (https://docs.python.org/3/library/exceptions.html#SyntaxError) 。

函数 eval() (https://docs.python.org/3/library/functions.html#eval) 和 exec() (https://docs.python.org/3/library/functions.html#exec) 不能访问解释名字的全局变量。名字可能是解决局部和全局命名空间的调用者。自由变量不能解决最近的封闭命名空间,但在全局命名空间中可以。[1] (https://docs.python.org/3/reference/executionmodel.html#id3) eval() (https://docs.python.org/3/library/functions.html#eval) 和 exec() (https://docs.python.org/3/library/functions.html#exec) 函数可选参数覆盖全局和局部命名空间。如果只指定一个命名空间,两个函数都可以使用。

异常

异常就是为了处理出错或者处理其它意外情况而中断代码块的正常控制流。异常在错误被检测到的位置被抛出，它可以被其周围相关代码的处理，或者错误发生处直接或间接调用的代码块处理。

Python 解释器在它检测到一个运行时错误时抛出一个异常（比如除以零）。某个Python 程序也可以通过 raise 语句显式地抛出异常。异常处理器可以用 try ... except 语句指定。try ... finally 语句指定清理代码块，但是它不处理异常，只是无论先前代码中是否产生异常都会得到执行。

Python使用所谓的“中断”错误处理模型：一个异常处理器能在外层找出错误发生和继续执行的位置。但是它不能修复错误和重试错误的操作（除非重新从头进入该段出错的代码）。

当一个异常没有得到控制，解释器就中断程序的执行，或返回到它的主循环的迭代中。其它情况下，除了不是 SystemExit (https://docs.python.org/3/library/exceptions.html#SystemExit) 异常，它还打印一个堆栈跟踪回溯对象。

异常由一个字符串对象或一个类实例标识，所匹配的 except (https://docs.python.org/3/reference/compound_stmts.html#except) 子句的选择是基于对象标识的：except子句必须引用相同的类或其基类，可以接受处理程序，也可以携带更多关于异常条件的信息。

注意：异常消息不是 Python 标准处理借口 API 的一部分，它的内容可能会伴随 Python 版本升级改变，而不带有任何告警，不应依赖于在多个版本的编译器下运行代码。

在 *The try statement* (https://docs.python.org/3/reference/compound_stmts.html#try) 章节见 try (https://docs.python.org/3/reference/compound_stmts.html#try) 的描述，以及 *The raise statement* (https://docs.python.org/3/reference/simple_stmts.html#raise) 章节见 raise (https://docs.python.org/3/reference/simple_stmts.html#raise) 的描述。

5

导入系统

Python 中，一个模块的代码通过导入 (https://docs.python.org/3/glossary.html#term-importing) 程序访问另一个模块 (https://docs.python.org/3/glossary.html#term-module) 的代码。导入 (https://docs.python.org/3/reference/simple_stmts.html#import) 语句是调用导入机制的最常用的方法，但不是唯一方式。像函数 importlib.import_module() (https://docs.python.org/3/library/importlib.html#importlib.import_module) 和内建函数 __import__() (https://docs.python.org/3/library/functions.html#__import__) 也能用来调用导入机制。

导入 (https://docs.python.org/3/reference/simple_stmts.html#import) 语句包含两个操作：首先查找指定的模块，然后将查找结果绑定到局部作用域内的一个名字上。导入 (https://docs.python.org/3/reference/simple_stmts.html#import) 语句中的查找操作被定义为一个具有适当参数的 __import__() (https://docs.python.org/3/library/functions.html#__import__) 函数的调用。 __import__() (https://docs.python.org/3/library/functions.html#__import__) 的返回值被用作执行导入语句的名称绑定操作。名称绑定操作的准确细节请见 __import__() (https://docs.python.org/3/library/functions.html#__import__) 语句。

直接调用 __import__() (https://docs.python.org/3/library/functions.html#__import__) 仅执行模块搜索和模块创建操作（如果查找到的话）。虽然可能发生某些附带后果，比如导入父包以及更新不同的缓存（包括 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) ），但只有导入 (https://docs.python.org/3/reference/simple_stmts.html#import) 语句执行名称绑定操作。

当调用 import() (https://docs.python.org/3/library/functions.html#__import__) 作为导入语句的一部分，导入系统首先在模块的全域命名空间中查找指定名称的函数。假如没有找到，则调用标准内建 __import__() (https://docs.python.org/3/library/functions.html#__import__) 。调用导入系统的其他机制，例如 importlib.import_module() (https://docs.python.org/3/library/importlib.html#importlib.import_module) ，不执行该项核对，并将一直使用标准导入系统。

当一个模块第一次被导入，Python 搜索该模块，如果找到，它便创建一个模块对象[1] (https://docs.python.org/3/reference/import.html#fnmo) ，并初始化。假如不能找到指定的模块，将引发 ImportError (https://docs.python.org/3/library/exceptions.html#ImportError) 异常 。当调用导入机制时，Python 实现不同的策略去搜索指定的模块。通过使用不同的钩子程序能够修改和扩展这些策略，钩子程序将在下面部分阐述。

3.3版本中的变化： 导入系统已更新到完全能够实现 PEP 302 (http://www.python.org/dev/peps/pep-0302) 的第二阶段。不再有隐含的导入机制——通过 sys.meta_path (https://docs.python.org/3/library/sys.html#sys.meta_path) ，整个导入系统是暴露的。此外，已实现本地命名空间包支持（见PEP 420 (http://www.python.org/dev/peps/pep-0420) ）。

导入库

importlib (https://docs.python.org/3/library/importlib.html#module-importlib) 模块提供丰富的 API 用以与导入系统交互。例如，为调用导入机制， importlib.import_module() (https://docs.python.org/3/library/importlib.html#importlib.import_module) 提供了一个推荐的，比内建函数 `__import__()` (https://docs.python.org/3/library/functions.html#__import__) 更简单的API。参阅 importlib (https://docs.python.org/3/library/importlib.html#module-importlib) 库文件了解更多细节。

包

无论模块是在 Python、C或是其他语言中实现，Python 只有一个模块对象型态，而且所有模块都是这个型态。为了帮助组织模块以及提供一个命名体系，Python 提供了一个包 (https://docs.python.org/3/glossary.html#term-package) 的概念。

你可以将包当成一个文件系统的目录，将模块当成目录中的文件，但不能太随便地做这样的类比，因为包和模块不需要来自文件系统。为了本文件的目的，我们将使用目录和文件的这个方便的类比。类似文件系统目录，包被分级组织起来，而且包本身也可以包含子包，常规模块也是如此。

重要的是要牢记所有的包都是模块，但不是所有的模块都是包。或者换一种说法，包仅是一种特殊的模块。具体地说，任何包含 `__path__` 属性的模块被认为是包。

所有的模块都有一个名称。类似于 Python 标准属性访问语法，子包与他们父包的名字之间用点隔开。因此，你可能有一个称为 sys (https://docs.python.org/3/library/sys.html#module-sys) 的模块和一个称为 email (https://docs.python.org/3/library/email.html#module-email) 的包，相应地你可能有一个称为 email.mime (https://docs.python.org/3/library/email.mime.html#module-email.mime) 的子包和该子包中一个称为 `email.mime.text` 的模块。

常规包

Python 定义两种型态包，常规包 (https://docs.python.org/3/glossary.html#term-regular-package) 和命名空间包 (https://docs.python.org/3/glossary.html#term-namespace-package) 。常规包是存在于 Python 3.2 及更早版本中的传统包。常规包通常被当作含 `__init__.py` 文件的目录来实现。当导入一个常规包时，该 `__init__.py` 文件被隐式执行，而且它定义的对象被绑定到包命名空间中的名称。 `__init__.py` 文件能包含其他任何模块能够包含的相同的 Python 代码，而且在导入它时，Python 将给模块增加一些额外的属性。

举个例子，下面这个文件系统布局定义了一个有三个子包的顶层 `parent` 包：

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

导入 `parent.one` 将隐式执行 `parent/__init__.py` 和 `parent/one/__init__.py` 。随后导入 `parent.two` 或者 `parent.three` 将分别执行 `parent/two/__init__.py` 和 `parent/three/__init__.py` 。

命名空间包

命名空间包是不同文件集 (https://docs.python.org/3/glossary.html#term-portion) 的复合，每个文件集给父包贡献一个子包。文件集可以存于文件系统的不同位置。导入过程中 Python 搜索的压缩文件，网络或者其他地方也可以找到文件集。命名空间包可以也可以不与文件系统的对象直接对应。他们可以是真实的模块但没有具体的表述。

命名空间包不使用寻常列表作为他们 `__path__` 的属性。相反，他们使用自定义迭代器型态，如果他们父包的路径（或者高阶包的 sys.path (https://docs.python.org/3/library/sys.html#sys.path) ）改变，它将在下次试图导入时在该包中自动重新搜索包部分。

没有带有命名空间包的 `parent/__init__.py` 文件。实际上，导入搜索中可能有多种 `parent` 目录存在，而每一个目录都被不同的部分提供。因此， `parent/one` 在物理上可能不是紧挨着 `parent/two` 边。在这种情况下，无论在什么时候导入该父包或它的子包时，Python 将为高级别的父包创建一个命名空间包。

命名空间包的详细说明也参见 PEP 420 (http://www.python.org/dev/peps/pep-0420) 。

搜索

搜索之前，Python 需要导入模块的全限定名 (https://docs.python.org/3/glossary.html#term-qualified-name) （或者包名称，但为了讨论的目的，两者之间的区别并不重要）。该名称可以来自导入 (https://docs.python.org/3/reference/simple_stmts.html#import) 语句的不同参数，或者，来自 importlib.import_module() (https://docs.python.org/3/library/importlib.html#importlib.import_module) 或 `__import__()` (https://docs.python.org/3/library/functions.html#__import__) 函数的参数。

该名称将被用在导入搜索的不同阶段，并且其可以是一个子模块的虚线路径，比如 `foo.bar.baz.` 。在这种情况中，Python 首先试图导入 `foo` ，然后是 `foo.bar` ，最后导入 `foo.bar.baz` 。如果中间的任何导入失败，将引发 ImportError (https://docs.python.org/3/library/exceptions.html#ImportError) 异常。

模块缓存

进行搜索时，搜索的第一个地方是 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 。这个映射作为前期已导入的所有模块的一个缓存，包括中间路径。所以，假如 `foo.bar.baz` 前期已被导入，那么，sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 将包含进入 `foo` ， `foo.bar` 和 `foo.bar.baz` 的入口。每个键都有自己的数值，都有对应的模块对象。

导入过程中，在 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 中查找模块名称，如果存在，相关的值是满足导入的模块，那么程序完成。然而，如果值为 `None` ，则引发 ImportError (https://docs.python.org/3/library/exceptions.html#ImportError) 异常。如果未找到模块名称，Python 将继续搜索模块。

sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 是可写的。删除一个键可能不会毁坏相关模块（因为其他模块可以保持引用它），但是它将使指定模块的缓存入口无效，导致 Python 在下次导入时重新搜索指定的模块。也可以分配键值为 `None` ，迫使下一次模块导入时引发 ImportError (https://docs.python.org/3/library/exceptions.html#ImportError) 异常。

注意，假如你保持引用模块对象，并使 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 中缓存入口无效，然后再重新导入指定的模块，则这两个模块对象将不相同。对比之下， imp.reload() (https://docs.python.org/3/library/imp.html#imp.reload) 将重新使用相同的模块对象，并通过重新运行模块代码简单地重新初始化模块内容。

查找器和加载器

如果指定模块没在 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 中找到，将调用 Python 的导入协议来查找和加载模块。这个协议包含两个概念性的对象，查找器 (https://docs.python.org/3/glossary.html#term-finder) 和 加载器 (https://docs.python.org/3/glossary.html#term-loader) 。一个查找器的任务是决定它是否能够通过运用其所知的任何策略找到指定模块。实现这两个接口的对象称为 导入器 (https://docs.python.org/3/glossary.html#term-importer) 。当他们发现他们能够加载所需的模块，他们返回自身。

Python 包括许多默认的查找器和导入器。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。第三个默认查找器搜索模块的导入路径 (https://docs.python.org/3/glossary.html#term-import-path) 。导入路径

(https://docs.python.org/3/glossary.html#term-import-path) 是一个位置的列表，这些位置可以命名文件系统路径或者压缩文件。它也可扩展到搜索任何可定位的资源，例如被 URLs 识别的资源。

导入机制是可扩展的，因此新的查找器能够被添加来扩展模块搜索的范围和广度。

事实上，查找器不真正加载模块。如果他们能够找到指定的模块，他们返回一个模块分支，即模块导入相关信息的封装，在加载模块时导入机制运用的信息。

以下部分将更具体讲述查找器 和 加载器的协议，包括如何创建和注册新的协议。
版本3.4中的变化： 在 Python 以前的版本中，查找器直接返回加载器 (https://docs.python.org/3/glossary.html#term-loader) ，然而现在他们返回含有加载器的模块分支。加载器在导入中仍被使用，但几乎没有责任。

导入钩子程序

导入机制被设计为可扩展的；其基础的运行机制是导入钩子程序。存在两种导入钩子程序的型态：元钩子程序和导入路径钩子程序。

在其他任何导入程序运行之前，除了 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 缓存查找，在导入处理开始时调用元钩子程序。这就允许元钩子程序覆盖 sys.path (https://docs.python.org/3/library/sys.html#sys.path) 处理程序，冻结模块，或甚至内建模块。如下所述，通过给 sys.meta_path (https://docs.python.org/3/library/sys.html#sys.meta_path) 添加新的查找器对象注册元钩子程序。

当他们的相关路径项被冲突时，导入路径钩子程序作为 sys.path (https://docs.python.org/3/library/sys.html#sys.path) (或者 `package.__path__`)处理程序的一部分被调用。如下所述，通过给 sys.path_hooks (https://docs.python.org/3/library/sys.html#sys.path_hooks) 添加新的调用来注册导入路径钩子程序。

元路径

在sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 中没有找到指定模块时，Python 紧接着会搜索包含一个元路径查找器对象的列表的sys.meta_path (https://docs.python.org/3/library/sys.html#sys.meta_path) 。为了查看他们是否知道如何处理指定模块，这些查找器将被质疑。元路径查找器必须实现 find_spec() (https://docs.python.org/3/library/importlib.html#importlib.abc.MetaPathFinder.find_spec) 方法，该函数需要三个参数：名称，导入路径以及（可选）对象模块。元路径查找器能够使用任何想要使用的策略来确定是否它能够处理指定的模块。

如果元路径查找器知道如何处理指定模块，它就返回一个分支对象。如果不能处理，就返回 `None` 。假如 sys.meta_path (https://docs.python.org/3/library/sys.html#sys.meta_path) 处理直到其列表最后也没有返回

一个分支，则引发 ImportError (https://docs.python.org/3/library/exceptions.html#ImportError) 异常。仅仅传播引起的其他任何异常情况将中止导入处理程序。

调用元路径查找器的 find_spec() (https://docs.python.org/3/library/importlib.html#importlib.abc.MetaPathFinder.find_spec) 方法需要两个或三个参数。第一个是导入模块的全限定名，比如 `foo.bar.baz` 。 第二参数是用来模块搜索的路径入口。对于高阶模块来说，第二个参数是 `None` ,但是对子模块或者子包来说，第二参数是父包 `__path__` 属性值。如果不能访问准确的 `__path__` 属性，将引发 ImportError (https://docs.python.org/3/library/exceptions.html#ImportError) 异常。第三个参数是一个已存在的对象，其随后将是加载的目标。导入系统仅在重新加载时通过一个目标模块。

一个单一的导入要求可以多次遍历元路径。举例说明，假设还没有缓存任何所涉的模块，在每个元路径查找器(`mpf`)上调用 `mpf.find_spec("foo", None, None)` ，导入 `foo.bar.baz` 将首先执行一个高阶导入。导入 `foo` 后，调用 `mpf.find_spec("foo.bar", foo.__path__, None)` ，通过再一次遍历元路径， `foo.bar` 将被导入。一旦完成导入 `foo.bar` ，最后的遍历将调用 `mpf.find_spec("foo.bar.baz", foo.bar.__path__, None)` 。

一些元路径查找器仅支持高阶导入。当除 `None` 以外的的任何东西作为第二参数通过时，这些导入器将一直返回 `None` 。

Python 默认的 sys.meta_path (https://docs.python.org/3/library/sys.html#sys.meta_path) 有三个元路径查找器，一个知道如何导入内建模块，一个知道如何导入冻结模块，另一个知道如何从一个导入路径 (https://docs.python.org/3/glossary.html#term-import-path) (即路径查找器 (https://docs.python.org/3/glossary.html#term-path-based-finder))中导入模块。

版本3.4中的变化： 元路径查找器的 find_spec() (https://docs.python.org/3/library/importlib.html#importlib.abc.MetaPathFinder.find_spec) 方法取代了现已不被推荐使用的 find_module() (https://docs.python.org/3/library/importlib.html#importlib.abc.MetaPathFinder.find_module) 。尽管它将继续无变化地运行，但仅当查找器不执行 `find_spec()` 时，导入机制才将试图实现它。

加载过程

当找到一个模块分支时，在加载模块时导入机将使用它（及它包含的加载器）。下面是导入加载部分时所发生的近似值：

```
module = None
if spec.loader is not None and hasattr(spec.loader, 'create_module'):
    module = spec.loader.create_module(spec)
if module is None:
    module = ModuleType(spec.name)
# The import-related module attributes get set here:
```

```
    _init_module_attrs(spec, module)

    if spec.loader is None:
        if spec.submodule_search_locations is not None:
            # namespace package
            sys.modules[spec.name] = module
        else:
            # unsupported
            raise ImportError
    elif not hasattr(spec.loader, 'exec_module'):
        module = spec.loader.load_module(spec.name)
        # Set __loader__ and __package__ if missing.
    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]
```

注意以下细节：

- 如果在 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 中存在一个指定名称的模块对象，导入将已返回它。
- 在加载器执行模块代码前，模块存在在 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 中。这是重要的，因为模块代码可以自我导入（直接或间接地）；预先把它添加到 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 中能阻止最坏情况下的无限递推和最好情况下的多重加载。
- 假如加载失败了，这个失败的模块（仅是这个失败的模块）将从 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 里移除。任何已在 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 缓存中的模块，以及任何附带成功加载的模块，都一定留存在缓存之中。这与重新加载时失败模块也能存留在 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 中的情况相反。
- 如后面部分 (https://docs.python.org/3/reference/import.html#import-mod-attrs) 总结的，创建模块后但在执行前，导入机制设置导入相关模块的属性（以上伪代码案例中的 “_init_module_attrs”）。
- 模块运行是模块命名空间密集加载的关键时刻。运行完全交给了能够决定什么密集以及怎样密集的加载器。

- 加载期间创建的和通过 exec_module() 的模块可能不是导入结束时返回的那个[2 (https://docs.python.org/3/reference/import.html#fnlo)]。
  3.4版本中的变化： 导入系统接替了加载器的公式化任务。这些以前由 importlib.abc.Loader.load_module() (https://docs.python.org/3/library/importlib.html#importlib.abc.Loader.load_module) 方法执行。

加载器

模块加载器提供加载的判定函数：模块执行。导入机制用要执行的模块对象的单一参数调用 importlib.abc.Loader.exec_module() (https://docs.python.org/3/library/importlib.html#importlib.abc.Loader.exec_module) 方法。任何从 exec_module() (https://docs.python.org/3/library/importlib.html#importlib.abc.Loader.exec_module) 返回的值将被忽略。
加载器必须满足下列要求：

- 假设模块是一个 Python 模块（与内建模块或动态加载扩展相反），加载器应该在模块全域命名空间（`module.__dict__`）中执行该模块代码。
- 假如加载器不能执行模块，它应该引发 ImportError (https://docs.python.org/3/library/exceptions.html#ImportError) 异常，即使将传播 exec_module() (https://docs.python.org/3/library/importlib.html#importlib.abc.Loader.exec_module) 引发的任何其他异常情况。

在许多例子中，查找器和加载器可以是相同的对象；在这种情况中，find_spec() (https://docs.python.org/3/library/importlib.html#importlib.abc.MetaPathFinder.find_spec) 方法将仅返回一个加载器设置为 `self` 的分支。

模块加载器可以选择在加载时通过实现一个 create_module() (https://docs.python.org/3/library/importlib.html#importlib.abc.Loader.create_module) 方法来创建模块对象。它需要一个参数，即模块分支，并在加载中返回新的模块对象用以使用。`create_module()` 不需要在模块对象上设置任何属性。假如加载器不定义 `create_module()`，导入机制本身将创建新的模块。

3.4版本中的更新： 加载器的 `create_module()` 方法。

3.4版本中的变化： load_module() (https://docs.python.org/3/library/importlib.html#importlib.abc.Loader.load_module) 方法被 exec_module() (https://docs.python.org/3/library/importlib.html#importlib.abc.Loader.exec_module) 取代，并且导入机制承担了所有加载的公式化任务。

为了与存在的加载器相兼容，导入机制将使用加载器的 `load_module` 方法（如果存在，并且加载器也不实现 `exec_module()` 的话）。然而， `load_module()` 已不被推崇，加载器应替代实现 `exec_module()`。

除了运行模块之外，`load_module()` 方法必须实现以上所述的所有的公式化加载功能。经过一些补充说明，所有相同的约束条件都可应用。

- 假设在 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 中存在一个命名的模块对象，加载器必须使用这个存在的模块。（否则，importlib.reload() (https://docs.python.org/3/library/importlib.html#importlib.reload) 不能正确运行）假如在 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 中不存在指定模块，加载器必须创建一个新的模块对象并把它添加到 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 中。
- 在加载器运行模块代码前，sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 中必须存在模块，以阻止无限递推或多重加载。
- 假如加载失败，加载器必须移除其嵌入 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 中的所有模块。但仅当加载器本身已显示加载过它时，它必须只移除失败模块。

模块分支

导入机制在导入中，尤其在加载前，使用有关每个模块的多种信息。大部分信息对所有模块都是共用的。模块分支的目的是在每个模块基础上封装导入相关信息。

在导入时使用一个分支允许语句在导入系统组件间传输，例如，在创建模块分支的查找器和运行它的加载器之间传输。最重要的是，它允许导入机制执行加载的公式化操作，而如果没有模块分支，加载器承担那项任务。

有关模块分支能拥有的相关信息，更详细地参见 ModuleSpec (https://docs.python.org/3/library/importlib.html#importlib.machinery.ModuleSpec) 。

3.4版本的更新.

导入关联模块属性

在加载器运行模块前，根据模块分支，导入机制在加载中在每个模块上填入下列属性。

`__name__`

`__name__` 属性必须设置为模块的全限定名。该名称用来唯一识别导入系统的模块。

`__loader__`

`__loader__` 属性必须设置为加载模块时导入机制使用的加载器对象。这主要是为了自我检查，但也可以用作加载器专用的附加功能，例如获得一个加载器相关的数据。

`__package__`

模块的 `__package__` 属性必须设置。它的值必须是一个字符串，但可以与其 `__name__` 是相同的值。当模块是包时，其 `__package__` 值应该设定为它的 `_name__` 。当模块不是包时，对于高阶模块 `__package__` 应该设置为空字符串，对于子模块，应设置为父包名称。具体详见 PEP 366 (http://www.python.org/dev/peps/pep-0366) 。

该属性可以替代 `__name__` 用作计算主要模块的显示相关导入，如 PEP 366 (http://www.python.org/dev/peps/pep-0366) 所定义的。

`__spec__` `__spec__` 属性必须设置为导入模块时所使用的模块分支。这主要用作自查以及主要用在加载中。准确设置 `__spec__` 同样应用于解释器启动中初始化的模块。有一个例外是 `__main__` ，在一些例子中在 `__spec__` 设置为 `None` 。

3.4版本的更新：

`__path__`

假如模块是个包（不管常规包还是命名空间包），该模块对象的 `__path__` 属性必须设置。值必须是迭代的，但如果 `__path__` 没有更深的意义，值也可以是空的。如果 `__path__` 不是空的，迭代结束时它必须产生字符串。更多有关 `__path__` 语义学的细节将在下面给出。无包模块不应有 `__path__` 属性。

`__file__`
`__cached__`

`__file__` 是可选的。如果设置，该属性的值必须是一个字符串。如果没有语义学意义（例如从数据库中加载的一个模块）,导入系统可以选择不设 `__file__` 。

如果设置了 `__file__` ，也可以适当设置 `__cached__` 属性，该属性是任何代码编译版本的路径（例如字节编译文件）。设置该属性不需要文件存在；路径能够简单地指向编译文件存在的位置。（见 PEP 3147 (http://www.python.org/dev/peps/pep-3147) ）。

没有设置 `__file__` 时，也可以设置 `__cached__` 。然而，那种情况相当非典型。最终，加载器是利用 `__file__` 和/或 `__cached__` 的使用者。所以假如一个加载器能够从一个缓存模块中加载而不能从一个文件中加载的话，适用这个非典型的情况可能是合适的。

`module.__path__`

根据定义，如果一个模块有一个 `__path__` 属性，无论它的值是什么，它就是一个包。

一个包的 `__path__` 属性在导入其子包时使用。在导入机制中，它与 sys.path (https://docs.python.org/3/library/sys.html#sys.path) 的功能很相同，即在导入时提供一个搜索模块的位置列表。然而，通常 `__path__` 比 sys.path (https://docs.python.org/3/library/sys.html#sys.path) 限制更多。

`__path__` 必须是字符串的一个迭代，但它可以为空。sys.path (https://docs.python.org/3/library/sys.html#sys.path) 中使用的相同的规则也适用包的 `__path__`，并且当遍历包路径时，将求助sys.path_hooks (https://docs.python.org/3/library/sys.html#sys.path_hooks) 下面将讲到）。

一个包的 `__init__.py` 文件可以设置或改变该包的 `__path__` 属性，并且这也是 PEP 420 (http://www.python.org/dev/peps/pep-0420) 之前实现命名空间包的典型方式。采用 PEP 420 (http://www.python.org/dev/peps/pep-0420) 之后，命名空间包不再需要提供只包含 `__path__` 操作代码的 `__init__.py` 的文件；导入机制自动准确设置命名空间包的 `__path__`。

模块 reprs

通过默认，所有模块都有一个可用的表示，然而，根据上述所设的各属性，在模块分支中，你可以更明确地控制各模块对象的表示。

假如模块存在一个分支（`__spec__`），导入机制将试图从它那生成一个表示。如果失败或不存在分支，导入系统将使用模块上任何可用信息制作一个默认的表示。它将尝试使用 `module.__name__`，`module.__file__` 和 `module.__loader__` 输入到该表示中，并默认缺失的任何信息。

以下是所使用的确切的规则：

- 假如模块有一个 `__spec__` 属性，该分支上的信息被用作产生表示。“name”，“loader”，“origin”，和 “has_location”这些属性将被求助。
- 假如模块有一个 `__file__` 属性，这用作模块表示的一部分。
- 假如模块没有 `__file__` 但确有一个不是None的 `__loader__`，那么该 加载器的表示用作模块表示的一部分。
- 或者，仅在表示中使用模块的 `__name__`。

3.4版本的变化： 使用loader.module_repr() (https://docs.python.org/3/library/importlib.html#importlib.abc.Loader.module_repr) 已不被推崇，而且模块分支现在被导入机制用作产生模块表示。

为了与 Python 3.3 向后兼容，在尝试以上所述任一方法前，假如定义过，将通过调用加载器的 module_repr() (https://docs.python.org/3/library/importlib.html#importlib.abc.Loader.module_repr) 方法产生模块表示。然而，不推荐使用该方法。

基于路径的查找器

如前所提及的，Python 本身带有几个默认的元路径查找器。其中之一，称作基于路径的查找器 (https://docs.python.org/3/glossary.html#term-path-based-finder) (路径查找器 (https://docs.python.org/3/library/importlib.html#importlib.machinery.PathFinder)),搜索含有一个路径入口列表 (https://docs.python.org/3/glossary.html#term-path-entry) 的一个导入路径 (https://docs.python.org/3/glossary.html#term-import-path) 。每个路径入口都指定一个搜索模块的位置。

基于路径的查找器本身不知道如何导入东西。相反，它遍历单一路径入口，将每个入口与知道如何处理这种特定路径的路径入口查找器结合起来。

路径入口查找器的默认设置实现所有在文件系统寻找模块的语义，处理类似 Python 资源代码 (`.py` 文件), Python 字节代码 (`.pyc` 和 `.pyo` 文件) 以及 共享库 (即 `.so` 文件)的特殊文件型态。当标准库中 zipimport (https://docs.python.org/3/library/zipimport.html#module-zipimport) 模块支持的话，默认路径入口查找器也处理从压缩文件中加载这些文件型态 (除了共享库以外) 的一切。

路径入口不需要受制于文件系统位置。他们可以引用 URLs，数据库查询，或者其他能够用一个字符串指定的位置。

路径基础查找器提供附加的钩子程序和协议，因此你能够扩展和自定义可搜索的路径入口型态。例如，假如你想要支持路径入口作为网络 URL，你可以写一个实现 HTTP 语义并能在网络上查找模块的钩子程序。这个钩子程序 (一个调用) 将返回一个支持下述协议的路径入口查找器 (https://docs.python.org/3/glossary.html#term-path-entry-finder) ，其随后用作从网络获得模块的一个加载器。

注意事项：这节和上一节都使用术语查找器，区别它们通过使用术语元路径查找器 (https://docs.python.org/3/glossary.html#term-meta-path-finder) 和路径入口查找器 (https://docs.python.org/3/glossary.html#term-path-entry-finder) 。这两种查找器的型态非常相似，都支持相似的协议，以及在导入程序中起相似的作用，但重要的是要牢记他们稍微有所不同。尤其，元路径查找器在导入程序开始时操作，切断 sys.meta_path (https://docs.python.org/3/library/sys.html#sys.meta_path) 遍历。

相比之下，在某种意义上，路径入口查找器是实现基于路径查找器的一个详述，并且事实上，如果将路径基础查找器从 sys.meta_path 中移除，没有任何路径入口查找器语义将被调用。

路径入口查找器

基于路径的查找器 (https://docs.python.org/3/glossary.html#term-path-based-finder) 负责查找和加载 Python 模块和包，他们可以利用一个字符串 路径入口 (https://docs.python.org/3/glossary.html#term-path-entry) 指明地址。大多数路径入口在文件系统中指定地址，但他们不必受限于此。

作为一个元路径查找器， 基于路径的查找器 (https://docs.python.org/3/glossary.html#term-path-based-finder) 实现先前所述的 `find_spec()` 协议，然而它曝光可以用来自定义如何从导入路径查找和加载模块的附加钩子程序。

基于路径的查找器 (https://docs.python.org/3/glossary.html#term-path-based-finder) 使用三个变量，sys.path (https://docs.python.org/3/library/sys.html#sys.path) ，sys.path_hooks (https://docs.python.org/3/library/sys.html#sys.path_hooks) 和 sys.path_importer_cache (https://docs.python.org/3/library/sys.html#sys.path_importer_cache) 。 同时也使用包对象上的 `__path__` 属性。这些提供了能够自定义导入机制的其他方法。

sys.path (https://docs.python.org/3/library/sys.html#sys.path) 含有一个提供模块和包的搜索位置的字符串列表。它从 PYTHONPATH 环境变量和其他不同安装专用和实现专用的默认中初始化。sys.path (https://docs.python.org/3/library/sys.html#sys.path) 的入口能够指定文件系统，压缩文件和其他潜在“位置”（见site (https://docs.python.org/3/library/site.html#module-site) 模块）的目录，这些“位置”，比如 URLs, 或者数据库查询，应被用作搜索模块。sys.path (https://docs.python.org/3/library/sys.html#sys.path) 应仅显示字符串和字节；忽略其他所有数据型态。字节入口的编码由个别路径入口查找器 (https://docs.python.org/3/glossary.html#term-path-based-finder) 决定。

基于路径的查找器 (https://docs.python.org/3/glossary.html#term-path-based-finder) 是一个元路径查找器 (https://docs.python.org/3/glossary.html#term-meta-path-finder) ，所以导入机制通过调用先前所述的路径入口查找器的 find_spec() (https://docs.python.org/3/library/importlib.html#importlib.machinery.PathFinder.find_spec) 方法开启导入路径 搜索。当给出 find_spec() (https://docs.python.org/3/library/importlib.html#importlib.machinery.PathFinder.find_spec) 的 `path` 参数，它将是一个要遍历的字符串路径的列表，通常是包内导入中一个包的 `__path__` 属性。假如 path 变量是 `None` ，这表明使用一个高阶导入和 sys.path (https://docs.python.org/3/library/sys.html#sys.path) 。

基于路径的查找器在搜索路径中迭代每个入口，并且每个为路径入口寻找一个适当的路径入口查找器 (https://docs.python.org/3/glossary.html#term-path-entry-finder) 。因为这可能是一个耗时的操作程序（例如，本次搜索有可能存在的 stat() 调用的耗时），基于路径的查找器维持一个路径入口查找器的缓存映射路径入口。这个缓存维持在 sys.path_importer_cache (https://docs.python.org/3/library/sys.html#sys.path_importer_cache) （尽管名称如此，这个缓存实际上储存查找器对象而不限制于导入器对象）。因此，这个耗时搜索，即查

找个别路径入口位置的路径入口查找器，仅需要完成一次即可。用户代码可以自由将缓存入口从迫使path based finder再一次执行路径入口搜索的sys.path_importer_cache中移除[3 (https://docs.python.org/3/reference/import.html#fnpic)]。

假如路径入口不在缓存中显示，基于路径的查找器将迭代 sys.path_hooks (https://docs.python.org/3/library/sys.html#sys.path_hooks) 中每个调用。使用单参数调用该列表中的每个路径入口钩子程序 (https://docs.python.org/3/glossary.html#term-path-entry-hook) ，即要搜索的路径入口。这个调用或者可能返回一个能够处理路径入口的路径入口查找器 (https://docs.python.org/3/glossary.html#term-path-entry-finder) ，或者可能引发 ImportError (https://docs.python.org/3/library/exceptions.html#ImportError) 异常。，基于路径的查找器使用 ImportError (https://docs.python.org/3/library/exceptions.html#ImportError) 异常来暗示钩子程序无法找到指定路径入口 (https://docs.python.org/3/glossary.html#term-path-entry) 的路径入口查找器 (https://docs.python.org/3/glossary.html#term-path-entry-finder) 。忽略异常情况， 导入路径迭代继续。钩子程序应该预料到或者会出现一个字符串或者会出现一个字节对象；字节对象的编程取决于钩子程序（例如，它可能是一个文件系统编程，UTF-8 或其他），而且假如钩子程序不能解码参数，它会引发 ImportError (https://docs.python.org/3/library/exceptions.html#ImportError) 异常。

假如 sys.path_hooks (https://docs.python.org/3/library/sys.html#sys.path_hooks) 迭代结束后没有路径入口查找器 (https://docs.python.org/3/glossary.html#term-path-entry-finder) 返回，那么基于路径的查找器的 find_spec() (https://docs.python.org/3/library/importlib.html#importlib.machinery.PathFinder.find_spec) 方法将在 sys.path_importer_cache (https://docs.python.org/3/library/sys.html#sys.path_importer_cache) 储存 None （表示没有该路径入口的查找器）并返回 None ，表示该元路径查找器 (https://docs.python.org/3/glossary.html#term-meta-path-finder) 不能找到模块。

假如 sys.path_hooks (页 0)调用返回一个路径入口查找器 (https://docs.python.org/3/glossary.html#term-path-entry-finder) ，则使用下面的协议用作向查找器请求一个加载模块时使用的模块分支。

路径入口查找器协议

为了支持导入模块和初始化包以及扩充命名空间，路径入口查找器必须实现 find_spec() (https://docs.python.org/3/library/importlib.html#importlib.abc.PathEntryFinder.find_spec) 方法。

find_spec() (https://docs.python.org/3/library/importlib.html#importlib.abc.PathEntryFinder.find_spec) 方法需要两个参数，导入模块的全限定名，以及（可选）目标模块。 find_spec() 返回完全填充的模块分支。该分支将一直拥有“加载器”设置（但有一个例外）。

导入机制中，分支代表命名空间的一部分 (https://docs.python.org/3/glossary.html#term-portion) 。路径入口查找器设置分支的 “loader” 为 None 并且设置 “submodule_search_locations” 为一个包含该部分的列表。

3.4版本的变化： find_spec() (https://docs.python.org/3/library/importlib.html#importlib.abc.PathEntryFinder.find_spec) 替代了 find_loader() (https://docs.python.org/3/library/importlib.html#importlib.abc.PathEntryFinder.find_loader) 和 find_module() (https://docs.python.org/3/library/importlib.html#importlib.abc.PathEntryFinder.find_module)，这两个现在都不被推荐使用，但在没有定义 `find_spec()` 时扔将使用。

早期的路径入口查找器可能实现这两个不被推崇的函数之一，而代替 find_spec()。为了向后兼容，这些方法仍被赞誉。但是如果在路径入口查找器上实现 find_spec()，那么遗留的方法将被忽略。

find_loader() (https://docs.python.org/3/library/importlib.html#importlib.abc.PathEntryFinder.find_loader) 需要一个参数，即导入模块的全限定名。`find_loader()` 返回一个二元运算符，其中第一项是加载器，第二项是一个命名空间的部分。当第一项(即加载器)是 `None` 时，这表明即使路径入口查找器没有指定模块的加载器，但它知道路径入口扩展指定模块的命名空间[部分]。这种情况几乎一直存在于请求Python导入没有物理显示的命名空间包的文件系统时。路径入口查找器返回加载器的值为None时，二元运算符的返回值的第二项必须是一个序列，尽管它可以是空的。

假如 `find_loader()` 返回一个不是 `None` 的加载器的值，该部分忽略，并且从基于路径查找器返回加载器，并通过路径入口结束搜索。

为了向后兼容其他导入协议的实现程序，许多路径入口查找器同时支持与元路径查找器支持的相同的、传统的 `find_module()` 方法。然而路径入口查找器的 `find_module()` 方法从来没有用一个 `path` 参数调用（他们预计会记录路径钩子程序的初始调用的准确路径信息）。

由于不允许路径入口查找器扩展命名空间包，因此路径入口查找器 上的 `find_module()` 方法不被推荐使用。如果 `find_loader()` 和 `find_module()` 同时存在于一个路径入口查找器，导入系统将一直优先调用 `find_loader()` 而不是 `find_module()`。

代替标准导入系统

最可靠的完全替代导入系统的机制是删除 sys.meta_path (https://docs.python.org/3/library/sys.html#sys.meta_path) 的默认内容，用一个自定义的元路径钩子程序完全代替他们。

如仅改变导入语句的行为而不影响访问导入系统的其他 APIs 是可以接受的话，那么代替内建 `__import__()` 函数可能就足够了。也可以在模块级上采用这个技术仅在该模块中改变导入语句行为。

在元路径早期，钩子程序选择性地阻止一些模块的导入（而不是完全将标准导入系统无效），直接在 find_spec() (https://docs.python.org/3/library/importlib.html#importlib.abc.MetaPathFinder.find_spec) 引发 ImportError (https://docs.python.org/3/library/exceptions.html#ImportError) 异常而不是返回 `None`。后者表明元路径搜索应该继续，而引发异常情况将立即终止程序。

__main__

__main__ (https://docs.python.org/3/library/__main__.html#module-__main__) 模块相对 Python 导入系统是一个特殊案例。像 elsewhere (https://docs.python.org/3/reference/toplevel_components.html#programs) 中明确的，更像 sys (https://docs.python.org/3/library/sys.html#module-sys) 和 builtins (https://docs.python.org/3/library/builtins.html#module-builtins)， __main__ 模块在解释器启动中直接初始化。然而，不像这两个，它不是严格意义上的内建模块。这是因为 __main__ 初始化的方法取决于标志和用来调用解释器的其他选项。

__main__.__spec__

根据初始化 __main__ (https://docs.python.org/3/library/__main__.html#module-__main__) 的方式，__main__.__spec__ 获得正确地设置或者设置为 None 。

Python 以-m (https://docs.python.org/3/using/cmdline.html#cmdoption-m) 选项开始时，__spec__ 设置为相应模块或包的模块分支。当加载 __main__ 模块作为部分执行目录，压缩文件或其他 sys.path (https://docs.python.org/3/library/sys.html#sys.path) 入口时，__spec__ 也将密集。

在其他情况 (https://docs.python.org/3/using/cmdline.html#using-on-interface-options) 下，__main__.__spec__ 设置为 None ，因为密集 main (https://docs.python.org/3/library/__main__.html#module-__main__) 使用的的代码不直接与可导入模块相对应。

- 交互提示
- -c 交换
- 标准输入
- 直接运行资源或字节代码文件

要注意在最后的案例中 __main__.__spec__ 一直是 None ，即使技术上文件可以作为模块直接导入。假如在 __main__ (https://docs.python.org/3/library/__main__.html#module-__main__) 中需要有效的模块元数据，则使用 -m (https://docs.python.org/3/using/cmdline.html#cmdoption-m) 交换。

也要注意，即使当 __main__ 对应一个可导入模块并且对应设置 __main__.__spec__ ，但仍认为他们是远程模块。这是因为 if __name__ == "__main__" 保护的区块：检查仅在模块用作密集 __main__ 命名空间时执行，且不是在常规的导入程序中。

开放式问题

XXX 做一个图表真的很有帮助。

XXX *（import_machinery.rst）一个区域仅做模块和包的属性，或许扩展或取代数据模块参考页的相关数据条目，怎么样?

XXX runpy, pkgutil 等人在图书馆手册都可以获得在上方的 “See Also” 连接，其指向新的导入系统部分。

XXX 增加有关 `__main__` 初始化不同方式的更多解释?

XXX 增加有关 `__main__` 异常/陷阱的更多信息（即从 PEP 395 (http://www.python.org/dev/peps/pep-0395) 复制）?

参考

自 Python 早期开始至今，导入机制已进化地相当大了。虽然自从写文档起一些细节已经改变了，但最初的包说明 (http://legacy.python.org/doc/essays/packages.html) 仍可读。

sys.meta_path (https://docs.python.org/3/library/sys.html#sys.meta_path) 最初的说明在 PEP 302 (http://www.python.org/dev/peps/pep-0302)，PEP 420 (http://www.python.org/dev/peps/pep-0420) 紧接着有展开。

PEP 420 (http://www.python.org/dev/peps/pep-0420) 介绍了 Python 3.3 的命名空间包 (https://docs.python.org/3/glossary.html#term-namespace-package)。PEP 420 (http://www.python.org/dev/peps/pep-0420) 也介绍了替代 `find_module()` 的 `find_loader()`。

PEP 366 (http://www.python.org/dev/peps/pep-0366) 描述了主模块中明确相关导入的 `__package__` 属性的附加。

PEP 328 (http://www.python.org/dev/peps/pep-0328) 介绍了绝对且明确的相关导入，以及 PEP 366 (http://www.python.org/dev/peps/pep-0366) 将最终说明包的语义的初始建议的 `__name__`。

PEP 338 (http://www.python.org/dev/peps/pep-0338) 定义执行模块为脚本。

PEP 451 (http://www.python.org/dev/peps/pep-0451) 在分支对象增加每个模块导入语句的封装。它也卸载加载机的大部分程式化任务到导入机制上。这些改变允许导入系统几个 API 的淘汰，同时也允许增加查找器和加载器的新方法。

脚注

[1]参考 types.ModuleType (https://docs.python.org/3/library/types.html#types.ModuleType) 。

[2]实现导入库避免直接使用返回数值。反而，它通过查找 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 中的模块名称获得模块对象。这间接影响的是导入的模块可能替代 sys.modules (https://docs.python.org/3/library/sys.html#sys.modules) 中的它本身。这就是在其他 Python 实现中没有保证运行的特定的实现行为。

[3]在遗留的代码中，可能发现 sys.path_importer_cache (https://docs.python.org/3/library/sys.html#sys.path_importer_cache) 中 imp.NullImporter (https://docs.python.org/3/library/imp.html#imp.NullImporter) 的实例。推荐改变代码而使用 `None` 代替。更多细节请见 Porting Python code (https://docs.python.org/3/whatsnew/3.3.html#portingpythoncode) 。

6
表达式

本章描述了 Python 中表达式的组成元素的含义。

句法注意：在本章和之后的章节中，描述句法时使用与词法分析时不同的扩展 BNF 记法。当某个句法规则（可能是可选的）具有如下形式：

```
name ::= othername
```

并且未给出特定语义时，name 的这种形式的意义就与 othername 相同。

数值间的转换

当用以下短语“数值型参数转换为通用类型”描述数值型操作数时，参数使用第三章结尾处的强制规则进行强制转换。如果两个参数都属于标准数值型，就使用以下的强制规则：

- 如果其中一个参数是复数，另一个也要转换成复数；
- 否则，如果其中一个参数是浮点数，另一个也要转换成浮点数；
- 否则，两个都是整数， 不需要转换。

对于某些运算符有特殊的规则（例如，作为运算符‘%’左侧参数的字符）。扩展必须制定转换规则。

原子

原子是表达式最基本的组成单位，最简单的原子是标识符或者字面值。 以圆括号、方括号或大括号括住的符号在句法上也看成是原子。原子的句法如下：

atom ::= identifier (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-identifier) | literal (https://docs.python.org/3/reference/expressions.html#grammar-token-literal) | enclosure (https://docs.python.org/3/reference/expressions.html#grammar-token-enclosure)

enclosure ::= parenth_form (https://docs.python.org/3/reference/expressions.html#grammar-token-parenth_form) | list_display (https://docs.python.org/3/reference/expressions.html#grammar-token-list_display) | dict_display (https://docs.python.org/3/reference/expressions.html#grammar-token-dict_display) | set_display (https://docs.python.org/3/reference/expressions.html#grammar-token-set_display) | generator_expression (https://docs.python.org/3/reference/expressions.html#grammar-token-generator_expression) | yield_atom (https://docs.python.org/3/reference/expressions.html#grammar-token-yield_atom)

标识符（名字）

作为一个原子出现的标识符是一个名字。参看标识符和关键字 (https://docs.python.org/3/reference/lexical_analysis.html#identifiers) 以及命名空间和结构框架 (https://docs.python.org/3/reference/executionmodel.html#naming) 。

当某名字捆绑的是一个对象时，使用该原子就是使用那个对象。当某名字没有捆绑就直接使用它，则会抛出一个 NameError (https://docs.python.org/3/library/exceptions.html#NameError) 异常。

私有名字变换: 在类定义中， 以两个或多个下划线开始，并且尾部不是以两个或多个下划线结束的标识符，它被看作是类的私有名字。在产生它的代码之前， 私有名字被变换成更长的形式。这种变换是在将去掉前导下划线的类名插入到名字前，再在类名前插入一个下划线。例如，在类 `Ham` 中定义的标识符 `__spam` ，会被变换成 `_Ham_spam` 。本变换是不依赖于使用该标识符处代码的句法上的上下文的。如果变换后的结果过长（超过 255 个字符）， 就会执行该Python实现定义的截短名字的操作。如果某类的名字仅仅由下划线组成，这种变换是不会发生的。

字面值

Python 支持字符串、字节和各种数值型的字面值:

`literal ::` = stringliteral (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-stringliteral) | bytesliteral (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-bytesliteral) | integer (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-integer) | floatnumber (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-floatnumber) | imagnumber (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-imagnumber)

使用一个字面值会得到一个具有给定值的相应类型的对象（字符串、字节、整数、浮点数、复数），如果是浮点数和复数，那么这个值可能是个近似值，详见 Literals (https://docs.python.org/3/reference/lexical_analysis.html#literals) 。

所有字面值都属于不可变的数据类型，因此对象的标识比起它们的值来说显得次要一些。多次使用相同值的字面值（在程序代码中以相同形式出现或者以不同的形式出现）可能获得的是相同的对象或具有相同值的不同对象。

括号表达式

一个括号表达式是位于一对小括号内可选的表达式表。

parenth_form ::= "(" [expression_list] (https://docs.python.org/3/reference/expressions.html#grammar-token-expression_list) ")"

表达式表生成什么类型的值括号表达式也就生成什么类型的值：如果表达式表中包括了至少一个逗号，它就生成一个元组；否则，就生成那个组成表达式表的唯一的表达式。

一个空的表达式表会生成一个空的元组对象。因为元组是不可变的，因此这里适用字符串所用的规则（即两个具有空表达式的元组可能是同一个对象也可能是不同的对象）。

请注意元组不是依靠小括号成定义的，而是使用逗号。其中空元组是个例外，此时要求有小括号——在表达式中允许没有小括号的“空”可能会引起歧义，并容易造成难以查觉的笔误。

列表、集合和字典的表示

为了构建一个列表、一个集合或者一个字典，Python 提供了一种特殊的语法叫“表示”，每一个都有两种方式：

- 容器中的内容被详细列出
- 或者是通过一套循环和过滤指令来计算，被称为“推导式”。

推导式的一般语法元素是：

comprehension ::= expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression) comp_for (https://docs.python.org/3/reference/expressions.html#grammar-token-comp_for)
comp_for ::= "for" target_list (https://docs.python.org/3/reference/simple_stmts.html#grammar-token-target_list) "in" or_test (https://docs.python.org/3/reference/expressions.html#grammar-token-or_test) [comp_iter (https://docs.python.org/3/reference/expressions.html#grammar-token-comp_iter)]
comp_iter ::= comp_for (https://docs.python.org/3/reference/expressions.html#grammar-token-comp_for) | comp_if (https://docs.python.org/3/reference/expressions.html#grammar-token-comp_if)
comp_if ::= "if" expression_nocond (https://docs.python.org/3/reference/expressions.html#grammar-token-expression_nocond) [comp_iter (https://docs.python.org/3/reference/expressions.html#grammar-token-comp_iter)]

推导式是由至少一个 for (https://docs.python.org/3/reference/compound_stmts.html#for) 子句以及后跟零个或多个 for (https://docs.python.org/3/reference/compound_stmts.html#for) 或 if (https://docs.python.org/3/reference/compound_stmts.html#if) 子句构成的一个表达式组成，在这种情况下，新列表的元素由每个 for (https://docs.python.org/3/reference/compound_stmts.html#for) 或 if (https://docs.python.org/3/reference/compound_stmts.html#if) 子句决定，嵌套是从左至右方向的，而且每执行到最内部的语句块就产生一个列表元素。

请注意，推导式是在一个单独的范围内执行的，所以目标列表的名字分配不能泄漏到封闭的范围内。

列表的表示

一个列表用一对方括号括住的表达式序列（可能为空）表示：

list_display ::= "[" [expression_list (https://docs.python.org/3/reference/expressions.html#grammar-token-expression_list) | comprehension (https://docs.python.org/3/reference/expressions.html#grammar-token-comprehension)] "]"

使用一个列表会生成一个新的列表对。它的值由表达式表或由推导式给出。当给出一个逗号分隔的表达式表时，从左到右地对每个元素求值然后按顺序放进列表对象中。 如果给出的是推导式， 列表是由从推导式得出的元素组合而成。

集合的表示

一个集合是用花括号来表示的，它和字典的区别是缺少分隔键和值的冒号。

set_display ::= "{" (expression_list (https://docs.python.org/3/reference/expressions.html#grammar-token-expression_list) | comprehension (https://docs.python.org/3/reference/expressions.html#grammar-token-comprehension)) "}"

集合的表示将产生一个新的集合对象，内容是由一个表达式序列或者一个推导式来制定。当提供一个由逗号分隔的表达式列表时，将从左到由计算它的元素并把它们加入到集合对象中。当提供一个推导式时，集合将由从推导式中得出的元素组成。

一个空的集合不能由 `{}` 构建，这种形式将构建一个空的字典。

字典的表示

一个字典用一对大括号括住的键/数据对的序列（可能为空）表示：

dict_display ::= "{" [key_datum_list (https://docs.python.org/3/reference/expressions.html#grammar-token-key_datum_list) | dict_comprehension (https://docs.python.org/3/reference/expressions.html#grammar-token-dict_comprehension)] "}"
key_datum_list ::= key_datum (https://docs.python.org/3/reference/expressions.html#grammar-token-key_datum) ("," key_datum (https://docs.python.org/3/reference/expressions.html#grammar-token-key_datum))* [","]
key_datum ::= expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression) ":" expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression)
dict_comprehension ::= expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression) ":" expression comp_for (https://docs.python.org/3/reference/expressions.html#grammar-token-expression)

使用一个字典会生成一个新的字典对象。

键/数据对按在字典中定义的从左到右的顺序求值：每个键对象作为键嵌入到字典中存储相应的数据。这意味着你可以在键/数据列表中多次指定相同的键，键值是最后一次指定的值。

字典推导式，与列表和集合推导式相反，需要两个由冒号分开的表达式，并且其后跟一般的 “for” 和 “if” 子句。当推导式在运行时，产生的键和值元素将按照它们生成的顺序插入到新的字典中去。

关于键值类型的限制已在前述章节 标准类型层次 (https://docs.python.org/3/reference/datamodel.html#types) 提及（总而言之，键的类型应该是可散列的，这就排除了所有的可变对象）。重复键之间的冲突不会被检测到；对给定的（有重复的）键来说，最后出现的数据（就是文字显示中出现在最右边的）成为（最终的）胜利者。

生成器表达式

生成器表达式是由圆括号括起来的紧凑的生成器符号。

generator_expression ::= "(" expression comp_for (https://docs.python.org/3/reference/expressions.html#grammar-token-expression) ")"

一个生成器表达式产生一个新的生成器对象。它是被括在圆括号中而不是方括号或者花括号中，除此之外，它的语法和推导式一样。

当 __next__() (https://docs.python.org/3/reference/expressions.html#generator.__next__) 方法被生成器对象调用时，在生成器表达式中用到的变量将被计算（和普通生成器相同）。然而，最左边的 for 子句将被立即计算，这样由它产生的错误将在处理生成器代码中其它可能出现的错误之前被发现。随后的 for (https://doc

s.python.org/3/reference/compound_stmts.html#for) 子句将不会被立即计算，因为它们将取决于之前的 for (https://docs.python.org/3/reference/compound_stmts.html#for) 循环。例如 `(x*y for x in range(10) for y in bar(x))` 。

当调用生成器表达式时，圆括号可以被忽略，只保留一个参数。详情请参见调用 (https://docs.python.org/3/reference/expressions.html#calls) 章节。

Yield 表达式

`yield_atom ::=` "("yield_expression (https://docs.python.org/3/reference/expressions.html#grammar-token-yield_expression) ")"
`yield_expression ::=` "yield" [expression_list (https://docs.python.org/3/reference/expressions.html#grammar-token-expression_list) | "from" expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression)]

y ield 表达式只有在定义一个生成器 (https://docs.python.org/3/glossary.html#term-generator) 功能时才会被使用，所以只能被用在函数定义的过程中。在函数体内使用 yield 表达式将会使该函数变成一个生成器。

当一个生成器函数被调用时，它将返回一个被称为生成器的迭代器。该生成器将控制生成器函数的执行。当生成器的某个方法被调用时，生成器函数将开始执行。同时，在执行第一个 yield 表达式的过程中，即上次被挂起的地方，将返回表达式列表 (https://docs.python.org/3/reference/expressions.html#grammar-token-expression_list) 的值给生成器调用程序。所谓挂起，即所有的本地状态都将被保留，包括局部变量的绑定情况、指令指针、内部计算堆栈以及一些异常处理的状态。当调用生成器的某种方法，该执行过程恢复时，该函数可以像 yield 表达式是另外一个外部表达式调用一样运行。恢复执行后 yeild 表达式的值将取决于使该执行过程恢复的方法。如果使用 `__next__()` (https://docs.python.org/3/reference/expressions.html#generator.__next__) 函数（通常是通过 for (https://docs.python.org/3/reference/compound_stmts.html#for) 语句或者内部 next() (https://docs.python.org/3/library/functions.html#next) 函数），则结果为 None (https://docs.python.org/3/library/constants.html#None) 。如果使用 send() (https://docs.python.org/3/reference/expressions.html#generator.send) 函数，结果为传递到该方法中的值。

所有这些特性让生成器函数和协同程序非常像，它们可以多次输出，有多于一个的程序执行入口点，而且它们的执行过程可以被挂起。唯一的区别是当输出一次执行结果后，需要继续执行时，生成器函数无法控制，只有当生成器被调用时才可以被控制。

在 try (https://docs.python.org/3/reference/compound_stmts.html#try) 块中的任何地方都可以使用 yield 表达式。如果一个生成器在它结束（引用计数达到0或者被垃圾收集）之前没有被恢复，迭代器的 close() (https://docs.python.org/3/reference/expressions.html#generator.close) 方法将会被调用，允许各种 finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 语句执行。

当使用 `yield from <expr>` 时，认为表达式为从迭代器。所有由该从迭代器提供的值将直接返回给当前生成器方法的调用程序。如果有合适的方法，所有由 send() (https://docs.python.org/3/reference/expressions.html#generator.send) 函数传递的值和所有由 throw() (https://docs.python.org/3/reference/expressions.html#generator.throw) 函数传递的异常都将传送给底层的迭代器。如果找不到合适的方法，send() (https://docs.python.org/3/reference/expressions.html#generator.send) 函数将抛出AttributeError (https://docs.python.org/3/library/exceptions.html#AttributeError) 或者 TypeError (https://docs.python.org/3/library/exceptions.html#TypeError) 异常，但是 throw() (https://docs.python.org/3/reference/expressions.html#generator.throw) 函数只抛出传递异常。

当底层叠加器完成计算时，抛出的 StopIteration (https://docs.python.org/3/library/exceptions.html#StopIteration) 异常实例的值属性将变成 yield 表达式的值。它可以在抛出异常 StopIteration (https://docs.python.org/3/library/exceptions.html#StopIteration) 异常时明确的设置，或者当子迭代器是一个生成器时（通过子生成器返回值）自动生成。

3.3版本修改的地方： 添加 `yield from <expr>` 语句使子生成器具备控制功能。

当 yield 表达式是赋值表达式右边唯一的表达式时，圆括号可以被忽略。

可参见： PEP 0255 (http://www.python.org/dev/peps/pep-0255) – **简单生成器**
提议在 Python 中 添加生成器和 yield (https://docs.python.org/3/reference/simple_stmts.html#yield) 的说明.
PEP 0342 (http://www.python.org/dev/peps/pep-0342) – **通过高级生成器实现的协同程序**
提议增强生成器的 API 和语法，让它们和简单生成器一样便于使用。
PEP 0380 (http://www.python.org/dev/peps/pep-0380) – **子生成器的授权语法**
提议介绍 yield-from 语法，让授权子生成器变得容易。

生成器-迭代器方法

这一部分将描述生成器迭代器的方法，该方法可以控制生成器程序的执行。

请注意：当生成器已经开始运行的时候，调用以下任何一个方法将抛出 ValueError (https://docs.python.org/3/library/exceptions.html#ValueError) 异常。

`generator.__next__()`

开启生成器函数的执行或者在上次执行 yield 表达式的地方恢复执行。当一个生成器函数被 *next*() (https://docs.python.org/3/reference/expressions.html#generator.__next__) 方法恢复执行，最近的 yield 表达式通常计算为 None (https://docs.python.org/3/library/constants.html#None) 。当生成器再次被挂起，表达式列表 (https://docs.python.org/3/reference/expressions.html#grammar-token-expression_list) 的值被返回给 *next*() (https://docs.python.org/3/reference/expressions.html#generator.__next__) 函数的调用

者，执行过程将继续到下一个 yield 表达式。如果生成器没有生成另外一个值就直接退出执行，将引发一个 StopIteration (https://docs.python.org/3/library/exceptions.html#StopIteration) 异常。

该方法通常被间接调用，如通过一个 for (https://docs.python.org/3/reference/compound_stmts.html#for) 循环或者通过内建 next() (https://docs.python.org/3/library/functions.html#next) 函数。

generator.send(value)

恢复执行过程并且传递给生成器函数一个值。value 参数的值将被赋予当前 yield 表达式。send() (https://docs.python.org/3/reference/expressions.html#generator.send) 方法返回生成器生成的下一个值。如果生成器没有产生其它值并退出执行后，将引发 StopIteration (https://docs.python.org/3/library/exceptions.html#StopIteration) 异常。当调用 send() (https://docs.python.org/3/reference/expressions.html#generator.send) 函数启动生成器时，必须以 None (https://docs.python.org/3/library/constants.html#None) 作为参数，因为没有接收该值得 yield 表达式。

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

当生成器被暂停将抛出一个 type 类型的异常，并返回由生成器函数生成的 next 值。如果生成器没有返回其它值并退出执行，将抛出一个 StopIteration (https://docs.python.org/3/library/exceptions.html#StopIteration) 异常。如果生成器函数没有对传入的异常进行捕获处理时，异常将返回给调用者。

generator.close()

当生成器函数被暂停执行时将抛出 GeneratorExit (https://docs.python.org/3/library/exceptions.html#GeneratorExit) 异常（正常退出或者已被关闭）。如果生成器函数抛出一个 StopIteration (https://docs.python.org/3/library/exceptions.html#StopIteration) （正常退出或者已被关闭）或者 GeneratorExit (https://docs.python.org/3/library/exceptions.html#GeneratorExit) 异常时（不捕获处理异常），生成器将正常结束。如果生成器返回其它值，将抛出 RuntimeError (https://docs.python.org/3/library/exceptions.html#RuntimeError) 异常。如果生成器抛出其它类型的异常，则异常将直接被返回给调用者。如果生成器因为异常或者正常退出而结束运行，则 close() (https://docs.python.org/3/reference/expressions.html#generator.close) 函数不起任何作用。

举例

此处用一个简单的例子来说明生成器和生成器函数的执行。

```
>>> 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.
```

使用 yield from 语句的例子,请参见PEP380：委托至子生成器的语法 (https://docs.python.org/3/whatsnew/3.3.html#pep-380) 。

基元

基元指和语言本身中接合最紧密的若干操作。它们的语法如下:

primary ::= atom (https://docs.python.org/3/reference/expressions.html#grammar-token-atom) | attributeref (https://docs.python.org/3/reference/expressions.html#grammar-token-attributeref) | subscription (https://docs.python.org/3/reference/expressions.html#grammar-token-subscription) | slicing (https://docs.python.org/3/reference/expressions.html#grammar-token-slicing) | call (https://docs.python.org/3/reference/expressions.html#grammar-token-call)

属性引用

一个属性引用是由一个主元（primary）后跟一个句号和一个名字构成:

attributeref ::= primary (https://docs.python.org/3/reference/expressions.html#grammar-token-primary) "." identifier (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-identifier)

主元必须是一个支持属性引用的类型的对象。引用对象属性时，即要求该被对象生成指定名字的属性。 该过程可以通过重写 _ *getattr* _() (https://docs.python.org/3/reference/datamodel.html#object.__getattr__) 方

法具体化。 如果该属性无效，将会抛出异常 AttribError (https://docs.python.org/3/library/exceptions.html#AttributeError) 。否则，对象决定生成的属性的类型和值。对同一属性的引用多次求值是有可能生成不同对象的。

下标

一个下标选择一个有序类型对象（字符串，元祖或列表）或映射（字典）对象的一项:

subscription ::= primary (https://docs.python.org/3/reference/expressions.html#grammar-token-primary) "[" expression_list (https://docs.python.org/3/reference/expressions.html#grammar-token-expression_list) "]"

主元（primary）必须是一个支持下标计算的对象（例如列表或字典）。用户自定义的对象通过定义 __getitem__() (https://docs.python.org/3/reference/datamodel.html#object.__getitem__) 方法来支持下标。

如果主元是一个映射，则对表达式表求值的结果必须是映射中的一个键，然后此下标操作在主元映射中选择与该键所对应的值。（如果表达式表只有一项，那么它就是一个元组）。

如果主元是一个有序类型，表达式（列表）的计算结果应该是一个整数或片段（见下方讨论部分）。

正式的语法对序列中的负索引没有特别的规定；但是，所有内建序列都提供了一个 __getitem__() (https://docs.python.org/3/reference/datamodel.html#object.__getitem__) 方法来解决负索引，即如果索引值是负数， 就加上该序列的长度(例如， x[-1] 选择 x 的最后一项)。

字符串的元素是字符，字符不是单独的数据类型而仅仅是只有一个字符长的字符串。

片段

一个片断选择某个有序类型对象（如字符串、元组、列表）一段范围之内的项。片断可以作为表达式使用，或者是赋值和 del 语句的目标。 下面是片断的句法:

slicing ::= primary (https://docs.python.org/3/reference/expressions.html#grammar-token-primary) "[" slice_list (https://docs.python.org/3/reference/expressions.html#grammar-token-slice_list) "]"
slice_list ::= slice_item (https://docs.python.org/3/reference/expressions.html#grammar-token-slice_item) ("," slice_item (https://docs.python.org/3/reference/expressions.html#grammar-token-slice_item))* [","]
slice_item ::= expression (https://docs.python.org/3/reference/expressions.html#grammar-token-slice_item) | proper_slice (https://docs.python.org/3/reference/expressions.html#grammar-token-proper_slice)

proper_slice ::= [lower_bound (https://docs.python.org/3/reference/expressions.html#grammar-token-lower_bound)] ":" [upper_bound (https://docs.python.org/3/reference/expressions.html#grammar-token-upper_bound)] [":" [stride (https://docs.python.org/3/reference/expressions.html#grammar-token-stride)]]
lower_bound ::= expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression)
upper_bound ::= expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression)
stride ::= expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression)

在这里形式句法的说明中有点含糊： 任何看起来像表达式表的语法构件也能看作是片断表，所以任何下标都可以解释为片断。但这样要比更复杂的句法要合适，该定义是没有歧义的，在这种情况下（在片断表中没有包括适当的片断或者省略写法）优先将其解释为下标，而不是片断。

一个片断的语义如下：主要是由该片断表构成的键作索引（使用相同的 __getitem__() (https://docs.python.org/3/reference/datamodel.html#object.__getitem__) 方法作为正常下标）。如果一个片断表包括至少一个逗号，那么键就是一个包括由片断中的所有项转换而来的元组； 否则， 就用独立的片断项作转换成为键。为表达式的片断项转换后仍是该表达式。正常的片断转换后是一个 start，stop 和step 属性为给定的下限，上限和步长的片断对象（见3.2节），对于缺少的表达式用 None 替代。

调用

一个调用就是以一系列参数 (https://docs.python.org/3/glossary.html#term-argument) （可能为空）调用一个可调用对象（例如，函数 (https://docs.python.org/3/glossary.html#term-function) ）：

call ::= primary (https://docs.python.org/3/reference/expressions.html#grammar-token-primary) "(" [argument_list (https://docs.python.org/3/reference/expressions.html#grammar-token-argument_list) [","] | comprehension (https://docs.python.org/3/reference/expressions.html#grammar-token-comprehension)] ")"
argument_list ::= positional_arguments (https://docs.python.org/3/reference/expressions.html#grammar-token-positional_arguments) ["," keyword_arguments (https://docs.python.org/3/reference/expressions.html#grammar-token-keyword_arguments)]
["","*" expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression)] ["," keyword_arguments (https://docs.python.org/3/reference/expressions.html#grammar-token-keyword_arguments)]
["," "**" expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expressio

n)]
| keyword_arguments (页 0)]
["," keyword_arguments (https://docs.python.org/3/reference/expressions.html#grammar-token-keyword_arguments)] ["," "**" expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression)]
| "*" expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression)
["," keyword_arguments (https://docs.python.org/3/reference/expressions.html#grammar-token-keyword_arguments)] ["," "**" expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression)]
| "**" expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression)
positional_arguments ::= expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression) ("," expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression))*
keyword_arguments ::= keyword_item (https://docs.python.org/3/reference/expressions.html#grammar-token-keyword_item) ("," keyword_item (https://docs.python.org/3/reference/expressions.html#grammar-token-keyword_item))*
keyword_item ::= identifier (https://docs.python.org/3/reference/lexical_analysis.html#grammar-token-identifier) "=" expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression)

在关键字参数表后面可以出现一个逗号，但它在语义上是没有任何作用的。

首先的工作是导出一个可调用对象（用户自定义函数，内建函数，内建方法对象，类定义，类实例方法，包含方法 `__call__()` (https://docs.python.org/3/reference/datamodel.html#object.__call__) 的对象都是可调用的）。所有的参数表达都在试图调用之前被计算。关于形参 (https://docs.python.org/3/glossary.html#term-parameter) 列表的句法请参考章节函数定义 (https://docs.python.org/3/reference/compound_stmts.html#function) 。

如果给出了关键字参数，它们首先被转换为位置参数。具体如下:第一步，根据形参表创建一串空闲槽，如果有N个位置参数，它们就被放在前N个槽中。然后，对于每个关键字参数，它的标识符用于检测其对应的槽(如果其标识符与第一个形参数名相同，它就占用第一个槽，以此类推)如果发现某个槽已经被占用，则引发 TypeError (https://docs.python.org/3/library/exceptions.html#TypeError) 异常。否则将参数的值(即使为 `None`)放进槽中。当所有关键字参数处理完成后， 所有未填充的槽用在函数定义中的相应的默认值填充。(默认值是由函数定义时计算出来的，所以，像列表和字典这样的可变类型对象作默认值时，它们会被那些没有相应槽指定参数值的调用所共享，通常要避免这样做)。如果仍有未填充默认的槽位，就会引发一个 TypeError (https://docs.python.org/3/library/exceptions.html#TypeError) 异常。否则，所有被填充的槽当作调用的参数表。

CPython实现细节：CPython实现了一种位置参数没有名字的内建函数，即使位置参数被命名也是为了阅读方便，因此无法提供关键词。在实现，这是实现在 C 使用 PyArg_ParseTuple() (https://docs.python.org/3/c-api/arg.html#c.PyArg_ParseTuple) 解析他们的论点的情况下。

如果位置参数的个数比形参槽数多，并且在未使用 `*identifier` 句法的情况下，会引发 TypeError (https://docs.python.org/3/library/exceptions.html#TypeError) 异常。使用该种句法时，形参接受一个包括有额外位置参数的元组(如果没有额外和位置参数，它就为空)。

如果任何一个关键字参数与形参名不匹配，并且在未使用 `**identifier` 句法的情况下，会引发 TypeError (https://docs.python.org/3/library/exceptions.html#TypeError) 异常。使用该种句法时，形参接受一个包括有额外关键字参数的字典(关键字作为键，参数值作为该键对应的值)，字典如果没有额外和关键字参数，它就为空。

如果在函数调用中使用了 `*exprsiones` 句法， 那么 `exprsiones` 的结果必须是有序类型的。这个有序类型对象的元素被当作附加的位置参数处理; 如果存在有位置参数 *x1，...，xN*，并且 `*exprsiones` 的计算结果为 *y1，...，yM*，那么它与具有 M+N 个参数 *x1，...xN，y1，...，yM* 的调用等效。

由此可以得到一个推论： 尽管 `*exprsiones` 句法出现在任何关键字参数之后，但它在处理关键字参数之前计算。（如果有的话， `**exprsiones` 也是如此，参见下述），所以：

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

一同使用关键字语法和 `*expression` 的情况十分罕见，所以实际上这种混乱是不会发生的。

如果在函数调用中使用 `**expression` 句法， `expression` 计算结果必须是一个字典（的子类）。其内容作为附加的关键字参数。如果一个关键字出现在 `expression` 中并且是一个显式关键字参数，就会引发 TypeError (https://docs.python.org/3/library/exceptions.html#TypeError) 异常。

使用 `*identifier”` 或 `**identifier` 句法的形参不能作为位置参数或关键字参数名使用。

一个元组如果没有引发异常，通常会返回一些值，可能为 `None` 。怎样计算这个值依赖于可调用对象的类型。

如果它是一

用户自定义函数：

执行此函数的代码块，并把参数传给它。它要做的第一件事就是将形参与实参对应起来。关于这点参见函数定义 (https://docs.python.org/3/reference/compound_stmts.html#function) 当代码块执行到 return (https://docs.python.org/3/reference/simple_stmts.html#return) 语句时，会指定这次函数调用的返回值。

内建函数或内建方法：

结果依赖于解释器，详见 Built-in Functions (https://docs.python.org/3/library/functions.html#built-in-funcs) 。

类对象：

返回该类的一个新实例。

类实例的方法：

调用对应的用户自定义函数，其参数个数比普通的函数调用多一：该实例成为方法的第一个参数。

类实例：

类实例必须定义方法 call ()；效果与调用该方法相同。

幂运算符

幂运算符比在操作数左边的一元运算符有更高的优先级；但比右面的一元运算符要低。句法为:

power ::= primary (https://docs.python.org/3/reference/expressions.html#grammar-token-primary) ["**" u_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-u_expr)]

因此，在一串没有括号的由幂运算符和一元运算符组成的序列，会从左到右面求值 (没有强制改变求值顺序) ： -1**2 结果为 -1。

当以两个参数调用 pow() (https://docs.python.org/3/library/functions.html#pow) 时，幂运算符与内建函数 pow() 有相同的语义: 生成左边参数的右边参数次方。数值型参数首先转换成通用类型。结果类型是参数经强制规则转换后的结果。

对于整数操作数，其结果的类型与操作数相同除非第二个参数为负数；在这种情况下，所有的参数被转换为 float 进而结果也是 float。例如， `10\*\*2` 返回 `100` ，但 `10\*\*-2` 返回 `0.01` 。

`0.0` 的任意负数指幂会引发 ZeroDivisionError (https://docs.python.org/3/library/exceptions.html#ZeroDivisionError) 异常。一个负数的分数指幂结果是一个 complex (https://docs.python.org/3/library/functions.html#complex) 。 (早期的版本中会引发 ValueError (https://docs.python.org/3/library/exceptions.html#ValueError) 异常。)

一元算术运算符和位运算符

所有一元算术运算符（和位运算符）有相同的优先级:

u_expr ::= power (https://docs.python.org/3/reference/expressions.html#grammar-token-power) | "-" u_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-u_expr) | "+" u_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-u_expr) | "~" u_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-u_expr)

一元运算符-（减）对其数值型操作数取负。

一元运算符+（加）不改变其数值型操作数。

一元运算符 （取反）对其普通整数或长整数参数求逆（比特级）。x的比特级求逆运算定义为 -(x+1) 。它仅仅用于整数型的操作数。

在以上所有的三种情况下，如果参数的类型不合法，就会引发一个 TypeError (https://docs.python.org/3/library/exceptions.html#TypeError) 异常。

二元算术运算符

二元算术运算符的优先级符合我们的正常习惯。注意其中有些运算符也可以应用于非数值型操作数。除了幂运算符，它们只分两个优先级:一个是乘法类运算，一个是加法类运算。

m_expr ::= u_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-u_expr) | m_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-m_expr) "*" u_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-u_expr) | m_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-m_expr) "//" u_expr (页 0) "/" u_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-u_expr) | m_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-m_expr) "%" u_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-u_expr)
a_expr ::= m_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-m_expr) | a_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-a_expr) "+" m_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-m_expr) | a_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-a_expr) "-" m_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-m_expr)

`*`（乘）运算符计算其操作数的积，其两个参数必须是数值型的，或一个是整数另一个是有序类型。第一种情况下，数值参数被转换成通用类型然后计算积。后一种情况下，重复连接有序类型对象。一个负连接因子产生一个有空的有序类型对象。

`/`（除）和 `//` (整除)运算符生成参数的商。数值型参数首先转换成通用类型。整数相除（`/`）生成一个浮点数，整数进行整除的结果就是对商的精确结果执行 `floor()` 函数的返回值。除（`/`）以零会引发 ZeroDivisionError (https://docs.python.org/3/library/exceptions.html#ZeroDivisionError) 异常。

`%`（取模）计算第一个参数除以第二参数得到的余数.数值型参数首先转换成通用类型。右面的参数为零，会引发 ZeroDivisionError (https://docs.python.org/3/library/exceptions.html#ZeroDivisionError) 异常。参数可以是浮点数，例如 `3.14 % 0.7 = 0.34`（因为 `3.14 = 4 * 0.7 + 0.34`）。模运算结果的符号与其第二个操作数相同（或零）；结果的绝对值小于第二个操作数的绝对值[1 (https://docs.python.org/3/reference/expressions.html#id16)]。

整除和取模运算可以用以下等式联系起来: `x == (x//y)*y + (x%y)` 。整除和取模运算也可以用内置函数 divmod() (https://docs.python.org/3/library/functions.html#divmod) : `divmod(x, y) == (x//y, x%y)` 联系起来。[2 (https://docs.python.org/3/reference/expressions.html#id17)]

`+`（加）运算符计算参数的和。参数必须都是数值型的，或都是相同有序类型的对象。对于前一种情况，它们先转换成通用类型，然后相加。后一种情况下，所有有序类型对象被连接起来。

`-`（减）计算参数的差，数值型的参数首先转换成通用类型。

移位运算符

移位运算符的优先级比算术运算符低。

`shift_expr ::=` a_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-a_expr) | shift_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-shift_expr) ("<<" | ">>") a_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-a_expr)

这些运算符接受整数作为参数。它们将第一个参数向左或向右移动第二个参数指出的位数。

右移 *n* 位可以定义为除以 `pow(2,n)` ,左移 *n* 位可以定义为乘以 `pow(2,n)` ;

注意：在目前的实现中，右操作数应不大于 sys.maxsize (https://docs.python.org/3/library/sys.html#sys.maxsize) 。如果右操作数是一个大于 sys.maxsize (https://docs.python.org/3/library/sys.html#sys.maxsize) 会引发 OverflowError (https://docs.python.org/3/library/exceptions.html#OverflowError) 异常。

二元位运算符

三个二元位运算符具有各不相同的优先级:

and_expr ::= shift_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-shift_expr) | and_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-and_expr) "&" shift_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-shift_expr)
xor_expr ::= and_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-and_expr) | xor_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-xor_expr) "^" and_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-and_expr)
or_expr ::= xor_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-xor_expr) | or_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-or_expr) "|" xor_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-xor_expr)

`&` 运算符进行比特级的 AND（与）运算，参数必须是整数。

`^` 运算符进行比特级的 XOR（异或）运算，参数必须是整数。

`|` 运算符进行比特级的 OR（同或）运算，参数必须是整数。

比较运算符

不像 C 语言，在 Python 中所有比较运算符具有相同的优先级，比所有算术运算符、移位运算符和位运算符的优先级都低，并且，表达式 `a < b < c` 具有和数学上一样的含义:

comparison ::= or_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-or_expr) (comp_operator (https://docs.python.org/3/reference/expressions.html#grammar-token-comp_operator) or_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-or_expr))*
comp_operator ::= "<" | ">" | "==" | ">=" | "<=" | "!=" | "is" ["not"] | ["not"] "in"

比较运算返回逻辑值: `True` 意味着结果为真， `False` 意味着结果为假。

比较运算符可以任意的连接，例如， `x < y <= z` 等价于 `x < y and y <= z` ，除了 `y` 只求值一次（但在这两种情况下，只要发现 `x < y` 为假， `z` 就不会被计算）。

形式上，如果 `a，b，c，...，y，z` 为表达式， `op1，op2，...，opN` 为比较运算符，则 `a op1 b op2 c ... y opN z` 等价于 `a op1 b and b op2 c and ... y opN z` ，但是每个表达式最多只求值一次。

注意 a op1 b op2 c 并没有隐式地规定 a 和 c 之间的比较运算种类，所以 x < y > z 是完全合法的（虽然看起来不太漂亮）。

运算符 <，>，==, >=，<=， 和 != 比较两个对象的值，这两个对象不需要具有相同的的类型 。如果两个都是数值型的，它们都转换成通用类型。否则， == 和 != 认为不同类型的对象之间的比较通常是不等的，而 <，>，>= 和 <= 则会引发 TypeError (https://docs.python.org/3/library/exceptions.html#TypeError) 异常。你可以通过定义丰富的比较方法，比如 __gt__() (https://docs.python.org/3/reference/datamodel.html#object.__gt__)（详见基本定制 (https://docs.python.org/3/reference/datamodel.html#customization)）控制非内建类型对象的比较行为。

相同类型的对象的比较法则依赖于该类型:

- 数值型按大小比较。
- float('NaN')和Decimal('NaN')的值比较特殊，它们是相同的，即 x is x，但是它们是不相等的，即 x != x。另外，拿任何值和非数值型进行比较都将返回 False。例如，3 < float('NaN') 和 loat('NaN') < 3都将返回 False。
- Bytes 对象是通过使用它的元素的值来进行字典序比较。
- 串按字典序比较（每个字符的序数用内建函数ord() (https://docs.python.org/3/library/functions.html#ord) 得到）。[3 (https://docs.python.org/3/reference/expressions.html#id18)]串和字节对象不能相互比较。
- 元组和按列表字典序比较（通过比较对应的项）。这意味着如果元组和列表相等，则它们的每个元素都要相等并且两个序列要有相同的类型和相同的长度。
  如果不相等，序列的排列顺序则和它们第一个不同的元素的排列顺序相同。例如[1,2，x] <= [1,2,y] 和 x <= y 的值是一样的。如果对应的元素不存在，则稍短序列的值较小。(例如： [1,2] < [1,2,3])。
- 映射（字典）仅当它们的存储（键值对）表一样时相等。顺序比较 ('<'，'<='，'>='，'>') 会引发 TypeErro (https://docs.python.org/3/library/exceptions.html#TypeError) 异常。
- 可变集合和不可变集合定义比较运算符的含义是测试为子集或者父集。这些关系不完全是定义排序（两个序列{1,2}和{2,3}是不相等的，互相之间也不是子集或者父集的关系）。所以，可变集合不适合作为完全依赖比较的函数的参数。例如，如果输入一系列的集合，则函数 min()，max() 和 sorted()将产生不确定的结果。
- 对大多数内建类型对象进行比较，如果不是相同的对象则结果就是不等的。一个对象被看作比另一个对象小或大，是不可以预知的，但在相同的程序其结果是前后一致的。

不同类型的对象的比较取决于该类型是否对比较算法提供了明确的支持。大多数数字类型可以互相比较。当不支持交叉式的比较时，方法会返回 NotImplemented 。

in () 运算符和 not in () 运算符用于测试集合成员。如果 x 是集合 s 的成员，那么 x in s 的结果为真，否则为假。x not in s 的结果与上相反。所有的内置序列和集合类型和和字典一样都支持 in 测试，测试字典中是否含有给定的键。对于容器类型，如列表、元组、可变集合、不可变集合、字典，或双向队列，表达式 x in y 等价于 any(x is e or x == e for e in y) 。

对于字符串和字节类型，当且仅当*X*是*y*的子字符串时 x in y 为真，等价于 y.find(X)!= 1 。空字符串被认为是任意一个字符串的子字符串，所以 "" in "abc" 返回 True 。

对于定义了 __contains__() (https://docs.python.org/3/reference/datamodel.html#object.__contains__) 方法的用户自定义类， x in y 为真仅当 y.__contains__(x) 为真.

对于没有定义 __contains__() (https://docs.python.org/3/reference/datamodel.html#object.__contains__) 方法但定义有 __iter__() (https://docs.python.org/3/reference/datamodel.html#object.__iter__) 方法的用户自定义类，计算x in y时，x==z并且在迭代时这些z值不断产生，则返回值为真。如果在迭代过程中抛出了一个异常，从外部看就像是in运算抛出的该异常。 最后，古老的迭代规则是可靠的：如果一个类定义了-getitem-（ ）函数，当且仅当有一个非负整数索引i且x==y[i]，则x in y的值为真，并且所有较低整数索引都不会抛出IndexErrorexception异常。（ 如果抛出了其它异常，表面上看也好像是in操作抛出的该异常 ）

运算符 not in (https://docs.python.org/3/reference/expressions.html#not-in) 可计算与运算符 in (https://docs.python.org/3/reference/expressions.html#in) 相反的结果.

运算符 is (https://docs.python.org/3/reference/expressions.html#is) 和is not (https://docs.python.org/3/reference/expressions.html#is-not) 用于测试对象标识: x is y 为真，当且仅当 x 和 y 是相同的对象， x is not y 取其反值。[4 (https://docs.python.org/3/reference/expressions.html#id19)]

布尔运算

or_test ::= and_test (https://docs.python.org/3/reference/expressions.html#grammar-token-and_test) | or_test (https://docs.python.org/3/reference/expressions.html#grammar-token-or_test) "or" and_test (https://docs.python.org/3/reference/expressions.html#grammar-token-and_test)
and_test ::= not_test (https://docs.python.org/3/reference/expressions.html#grammar-token-not_test) | and_test (https://docs.python.org/3/reference/expressions.html#grammar-token-and_test) "and" not_test (https://docs.python.org/3/reference/expressions.html#grammar-token-not_test)
not_test ::= comparison (https://docs.python.org/3/reference/expressions.html#grammar-token-co

mparison) | "not" not_test (https://docs.python.org/3/reference/expressions.html#grammar-token-not_test)

在布尔运算的上下文和控制流语句使用的表达式中，以下值解释为假： False 、 None 、所有类型的数值零、空的字符串和容器（字符串、元组、列表、字典、可变集合和不可变集合）。 所有其它值解释为真。用户自定义对象可以通过提供 *__bool__()* (https://docs.python.org/3/reference/datamodel.html#object.__bool__) 方法自定义真值。

如果运算符 not 的参数为 False ，它返回 True ，否则返回 False 。

表达式 x and y 首先计算 *x*; 如果 *x* 为假，就返回它的值; 否则，计算 *y* 的值，并返回其结果。

表达式 x or y 首先计算 x ; 如果 *x* 为真，就返回它的值; 否则，计算 *y* 的值，并返回其结果。

（注意 and (https://docs.python.org/3/reference/expressions.html#and) 和 or (https://docs.python.org/3/reference/expressions.html#or) 都没有限制它的结果的值和类型必须是 False 或 True ，仅仅是最后一个计算的参数。这在某些情况下是有用的，例如，如果 s 是一个若为空就应该为默认值所替换的串，表达式 s or 'foo' 就会得到希望的结果。因为 not (https://docs.python.org/3/reference/expressions.html#not) 根本不会生成一个值，就没必要让它的返回值类型与其参数的相同，所以， not 'foo' 返回 False ，而非"）。

条件表达式

conditional_expression ::= or_test (https://docs.python.org/3/reference/expressions.html#grammar-token-or_test) ["if" or_test (https://docs.python.org/3/reference/expressions.html#grammar-token-or_test) "else" expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression)]
expression ::= conditional_expression (https://docs.python.org/3/reference/expressions.html#grammar-token-conditional_expression) | lambda_expr (https://docs.python.org/3/reference/expressions.html#grammar-token-lambda_expr)
expression_nocond ::= or_test (https://docs.python.org/3/reference/expressions.html#grammar-token-or_test) | lambda_expr_nocond (https://docs.python.org/3/reference/expressions.html#grammar-token-lambda_expr_nocond)

条件表达式（有时也称为“三元算子”）在所有的Python操作中具有最低优先级。

表达式 x if C else y ，首先计算条件 *C*，而不是条件 *x*，如果 *C* 为真，则计算 *x* 并返回 *x* 的值；如果 *C* 为假，则计算 *y* 并返回 *y* 的值。

关于条件表达式的详细介绍请参见PEP 308 (http://www.python.org/dev/peps/pep-0308) 。

lambda 表达式

lambda_expr ::= "lambda" [parameter_list (https://docs.python.org/3/reference/compound_stmts.html#grammar-token-parameter_list)]: expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression)
lambda_expr_nocond ::= "lambda" [parameter_list (https://docs.python.org/3/reference/compound_stmts.html#grammar-token-parameter_list)]: expression_nocond (https://docs.python.org/3/reference/expressions.html#grammar-token-expression_nocond)

lambda 表达式(也称作 lambda 形式)被用于创建类型函数。表达式 lambda arguments: expression 生成一个行为与下面定义的函数一致的函数对象:

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

对于参数表句法，参见函数定义 (https://docs.python.org/3/reference/compound_stmts.html#function) 。注意由 lambda 形式创建的函数不能包括语句。

表达式表

expression_list ::= expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression) ("," expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression))* [","]

一个表达式表是一个包括至少一个逗号的元组，它的长是表中表达式的个数。其中表达式从左到右按顺序计算。

最后的逗号仅仅在创建单元素元组（又称为”独元”）时才需要; 在其它情况下，它是可选的。一个没有后缀逗号的表达式不会创建元组，但仍会计算该表达式的值。（可以使用一对空括号() 创建一个空元组）。

计算顺序

Python 从左到右计算表达式，注意当计算一个赋值表达式时，先计算右边再计算左边。

下边的内容，表达式将以他们后缀的顺序来计算

```
expr1, expr2, expr3, expr`

(expr1, expr2, expr3, expr4`
```

```
{expr1: expr2, expr3: expr4}

expr1 + expr2 * (expr3 - expr4)

expr1(expr2, expr3, *expr4, **expr5)

expr3, expr4 = expr1, expr2
```

运算符优先级

下表总结了 Python 中运算符的优先级，从低优先级（弱捆绑）到高优先级（强捆绑），在同一格子中的运算符具有相同的优先级。如果没有特殊的句法指定，运算符是二元的。一格子内的运算符都从左至右结合（幂运算符是个例外，它从右至左结合）。

注意比较运算、成员测试、标志测试它们都有相同的优先级，且可以从左到右连接起来，具体参考比较运算符 (https://docs.python.org/3/reference/expressions.html#comparisons) 。

运算符	*描述*
lambda	lambda 表达式
if – else	条件 表达式
or	布尔OR
and	布尔AND
not x	布尔NOT
in, not in, is, is not, <, <=, >, >=, !=, ==	成员测试，标志测试，比较
|	比特级OR
^	比特级XOR
&	比特级AND
<<, >>	移位
+, –	加减
*, /, //, %	乘，除，整除，取余[5] (https://docs.python.org/3/reference/expressions.html#id20)
+x, –x, ~x	正运算,负运算，比特级not
**	幂运算 [6] (https://docs.python.org/3/reference/expressions.html#id21)
x[index], x[index:index], x(arguments...), x.attribute	下标，片段，函数调用，属性
(expressions...), [expressions...], {key: value...}, {expressions...}	表达式分组或元祖，列表，字典，集合

脚注

[1]虽然在数学上表达式 `abs(x % y) < ABS( y` 的值为真，但在数值上，对于浮点数来说，由于舍入的原因却不一定为真。例如，假设在一个平台上，Python 的浮点数均为符合 IEEE 754 标准的双精度浮点数，为了使 `-1e-100 % 1e100` 的计算结果和 `1e100` 有相同的符号，计算结果为 `-1e-100 + 1e100`，即数值上等于 1e100。函数 math.fmod() (https://docs.python.org/3/library/math.html#math.fmod) 的返回值符合和第一个参数值的符号相同，所以在该例中将返回 `-1e-100`。哪种方式更合适取决于具体的应用。

[2]如果X的值约等于y的整数倍，由于舍入原因，`x//y` 的值很可能比 `x-x%y//y` 的值大。在这种情况下，为了保持 `divmod(x, y)[0] * y + x % y` 的值和 x 的值非常接近，Python 将返回后者的值。

[3]虽然字符串比较在字节水平上讲的通，但对用户来说是难以理解的。例如，字符串 `"/u00c7"` 和 `"u0327/u0043"` 比较起来是不同的，即便它们代表同一个 Unicode 字符（带有变音符号的拉丁文大写字母C）。为了使用更符合人类认知的方法去比较字符串，采用函数unicodedata.normalize() (https://docs.python.org/3/library/unicodedata.html#unicodedata.normalize)。

[4]由于自动垃圾回收、灵活的列表以及描述器的动态属性，你或许已经注意到了 is (https://docs.python.org/3/reference/expressions.html#is) 运算符在某些特定的应用中表现出不寻常的行为，比如涉及到实例方法或者是常数的比较。详情请参阅相关参考资料。

[5]运算符 `%` 也被用来格式化字符串；与“取余”操作的优先级相同。

[6]与算术运算符或位运算符相比，运算符 `**` 右边的数结合的并没有特别紧密，即 `2**-1` 的值为 `0.5`。

7

简单语句

简单语句可以在一个逻辑行内表示，一些简单语句可以写在一行由分号分隔，简单语句的句法如下：

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

表达式语句

表达式语句用于计算和写一个值(多用在交互方式下), 或者（通常）调用过程（一个返回没有意义的结果的函数; 在Python中,过程返回 None ）。允许其它表达式语句的使用方法，有时也用的到。表达式语句的句法如下：

```
expression_stmt ::=  expression_list
```

一个表达式语句要对该表达式列表(可能只是一个表达式)求值.

在交互模式下，如果该值不是空，就使用内建函数repr() (https://docs.python.org/3/library/functions.html#repr) 并自己将结果字符串写入一行标准输出，（除非结果为空，导致过程调用不会引起任何输出。）

赋值语句

使用赋值把名字绑定(重新绑定)到值，以及修改可变对象的属性或者项目：

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

(请参阅 *attributeref*、 *subscription*和*slicing*的基本 (https://docs.python.org/3/reference/expressions.html#primaries) 语法定义部分。)

一个赋值语句要对该表达式列表求值(请记住这可以是一个表达式或者一个逗号分隔的列表，后者导出一个元组)，然后从左到右地将对象结果依次赋给目的列表里的每个对象。

根据目标（列表）的形式，赋值被递归地定义。当目标是一个可变对象的一部分时(一个属性引用,subscription或者slicing),可变的对象必须最终执行赋值,并确定其有效性,如果在赋值不可接受的情况下，可以引发出一个异常。被各种类型所遵循的规则和抛出的异常给出针对对象类型的定义(见标准的类型层次结构 (https://docs.python.org/3/reference/datamodel.html#types) 部分)。

分配对象到目标列表,括号或方括号内为可选，递归地定义如下.

- 如果目标列表是单个目标，该对象就赋予该目标。
- 如果目标序列是一组用逗号分隔的目标：该对象必须是一个其子项个数与目标列表中的目标个数 一样多的迭代对象，且其子项，从左到右地逐个赋予相应目标。
    - 如果目标列表包含一个前缀带星号的目标，被成为“starred”目标：这个对象必须是一个序列在目标列表中，有尽量多的子项目，不能少于一个子项目，序列从第一项被赋值，在带星的目标之前，从左到右逐个分配到目标列表。序列最后一项被赋值,则赋值到星号之后的目标。列表中剩余的项目最后被赋值给带星的目标（列表可以是空的）。
    - 否则：这个对象必须是一个在目标列表中有相同数目子项的序列，并且子项从左到右地赋予相应目标。

一个对象向单个目标递归地赋值定义如下。

- 如果该目标是一个标志符（名字）：
    - 如果该名字不出现于当前代码块的global (https://docs.python.org/3/reference/simple_stmts.html#global) 或者nolocal (https://docs.python.org/3/reference/simple_stmts.html#nonlocal) 语句当中：该名字就约束到当前本地命名空间的对象上。
    - 否则：该名字分别约束到全局名字空间或者nolocal (https://docs.python.org/3/reference/simple_stmts.html#nonlocal) 确定的外部名字空间。

如果名字已经被约束了它就被重新约束。这可能导致早先约束到该名字的对象的引用计数降为 零，导致释放该对象的分配空间并调用其析构器，如果它有一个的话。

- 如果目标是一个用括号或者方括号括起来的目标序列：该对象必须是具有和目标序列中目标个数同样数目的迭代类型，且其子项从左到右地赋值给相应目标。

- 如果目标是一个属性引用：引用中的主元表达式被求值。它应该给出一个带可赋值属性的对象； 如果不是这种情况，就会抛出TypeError (https://docs.python.org/3/library/exceptions.html#TypeError) 异常。然后那个对象就被要求将被赋值的对象赋值给给定 的属性；如果它无法执行该赋值，就会抛出一个例外（ 通常，但不是必须AttributeError (https://docs.python.org/3/library/exceptions.html#AttributeError) 异常 ）。

注意：如果对象是类的实例并且属性引用的赋值操作发生在两端，RHS 表达式， a.x 既能访问一个实例属性，也能（ 如果没有实例属性存在 ）访问一个类的属性。这个LHS目标 a.x 通常设定一个实例属性，如果有必要就创建。因此，这两个 a.x 的发生不需要涉及同样的属性，如果RHS表达式涉及到类的属性，这个LHS会创建一个新的实例属性作为赋值的目标。

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

这个描述并不一定适用于描述符属性,像属性创建使用property() (https://docs.python.org/3/library/functions.html#property) 。

- 如果目标是一个下标：引用中的主元表达式被求值。这应给出或者一个可变有序对象(比如，一个列表)或者一个映射对象(比如，一个字典)。接着，下标表达式被求值。

如果基础对象是可变有序对象(例如列表)，下标必须给出一个整数。如果是负数，序列的长度就被加上。最后的值必须是一个小于该序列长度的非负整数，然后该序列就被请求将被赋对象赋值给 它带那个指标的项。如果指标超出范围，就会抛出IndexError (https://docs.python.org/3/library/exceptions.html#IndexError) 异常(给一个用下标引用的有序对象赋 值不会给列表增添新项)

如果基础对象是一个映射对象（ 比如字典 ），下标的类型必须和映射的键类型兼容，接着该映射就被 要求创建一个把下标映射到被赋对象的键/数据对。这(操作)要不用新键值取代已存在的具有相同 键的键/值对，要不插入一个新的键/值对(如果不存在相同的键)

用户自定义的对象, __setitem__() (https://docs.python.org/3/reference/datamodel.html#object.__setitem__) 方法被调用以适当的参数。

- 如果目标是一片断：引用中的主元表达式被求值。这应给出一个可变有序对象(比如列表)。被赋值 对象应该是同一类型的有序对象。下一步，在它们所出现的范围内，对上下限表达式求值；缺省值是零和序列长度。限值应求值为整数。如果任一限是负的，就加上序列的长度。结果限值被修整至零和序列长度之间(含零和序列长度)。最后，有序对象被要求用被赋有序对象的子项替换该片断。片断的长度可能和被赋序列的长度不同，那就改变目标序列的长度，如果该对象允许的话。

CPython实现细节：在当前实现中，目标对象的语法采取同一表达式，同时无效的语法在代码生成阶段内被拒绝,导致更少的详细的错误消息输出.

虽然赋值的定义隐含着左手边和右手边之间的重叠是“同时的”(比如，`a, b = b, a` 交换两个变量)，在所赋值变量间的重叠却是不安全的！在集合中出现从左到右分配变量间的重叠是不安全的，有时候值是混乱的，例如，下面的程序打印出 `[0, 2]`：

```
x = [0, 1]
i = 0
i, x[i] = 1, 2 # i is updated, then x[i] is updated
print(x)
```

> 参见:
>
> PEP 3132 (http://www.python.org/dev/peps/pep-3132) – Extended Iterable Unpacking
>
> The specification for the *target feature.

增量的赋值语句

增量赋值就是在单条语句内合并一个二元运算和一个赋值语句。

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

（见最后三项符号基础 (https://docs.python.org/3/reference/expressions.html#primaries) 语法定义）

一条增量赋值语句对目标(和一般的赋值语句不同，它不能是展开的对象)和表达式列表求值，执行特定于两个操作数的赋值类型的二元运算，并将结果赋值给原先的目标。目标仅求值一次。

一条赋值语句，比如 `x+=1`，可以重写为 `x=x+1`，效果是类似的，但并不完全一样。在增量版本中，`x` 仅求值一次。而且，只要可能，实际的操作是*就地*进行的，意思是并非创建一个新对象然后将其赋值给目标，而是修改老的对象。

不像普通的赋值语句，增量赋值先计算左边再计算右边。例如，`a[i] += f(x)` 首先查看 `a[i]`，然后它计算 `f(x)` 并执行加法操作，最后，把结果写入 `a[i]`。

除了在一条语句中赋值给元组和多个对象的情况，增量赋值语句所完成的赋值用与普通赋值同样的方式处理。类似地，除了可能的*就地*方式，由增量赋值执行的二元运算和普通的二元运算也是一样的。

For targets which are attribute references, the same caveat about class and instance attributes (https://docs.python.org/3/reference/simple_stmts.html#attr-target-note) applies as for regular assignments.

assert 语句

assert语句是一个在程序中插入调试断言的常用方法:

assert_stmt ::= "assert" expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression) ["," expression (https://docs.python.org/3/reference/expressions.html#grammar-token-expression)]

简单形式的, ”assert expression”, 等价于:

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

扩展形式的, ”assert expression”, 等价于:

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

这些等价式假定了存在 __debug__ (https://docs.python.org/3/library/constants.html#__debug__) 和AssertionError (https://docs.python.org/3/library/exceptions.html#AssertionError) , 而不是具有相同名字的相应内建变量.在当前实现, 内建变量 __debug__ (https://docs.python.org/3/library/constants.html#__debug__) 在普通情况下为True, 在要求优化的情况下为False(命令行选项-O) 在编译要求优化时, 当前的代码生成器不产生任何断言语句的代码.注意在错误信息包括源代码的作法是多余的; 因为它们会作为跟踪回溯对象的一部分显示。

给 __debug__ (https://docs.python.org/3/library/constants.html#__debug__) 赋值是非法的,解释器是在启动时读取内建变量的值的。

pass 语句

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

pass (https://docs.python.org/3/reference/simple_stmts.html#pass) 是一个空操作 – 当执行它，什么也不会做。它在句法上要求有一个语句时但不需要写代码时用到,例如:

```
def f(arg): pass   # a function that does nothing (yet)
class C: pass      # a class with no methods (yet)
```

del 语句

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

与赋值的定义方法类似，删除也是递归的。下面是一些说明:

一个目标表的递归删除操作会从左到右地删除其中的每个对象地.

删除名字就是在本地命名空间和全局名字空间删除掉该名字的绑定(必须存在), 从哪个名字空间删除取决于该名字是否出现在其代码块的global (https://docs.python.org/3/reference/simple_stmts.html#global) 语句中。如果名字没有绑定，会抛出NameError (https://docs.python.org/3/library/exceptions.html#NameError) 异常。

对于属性引用, 下标和片断的删除会作用到相关的主元对象, 对片断的删除一般等价于对该片断赋予相应 类型的空片断(但这也受被截为片断的对象的限制).

3.2版本变化： 之前如果嵌套数据块中有空变量，在本地命名空间中删除名字是不合法的。

return 语句

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

return (https://docs.python.org/3/reference/simple_stmts.html#return) 在句法上仅可以出现在嵌套的函数定义中, 不能出现在嵌套的类定义中.

如果给出了表达式表, 就计算其值, 否则就用 `None` 代替.

return (https://docs.python.org/3/reference/simple_stmts.html#return) 的作用是离开当前函数调用, 并以表达式表的值(或 `None`)为返回值.

当return (https://docs.python.org/3/reference/simple_stmts.html#return) 放在具有finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 子句的try (https://docs.python.org/3/reference/compound_stmts.html#try) 语句中, finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 子句中的语句会在函数真正退出之前执行一次.

在生成器函数中, return (https://docs.python.org/3/reference/simple_stmts.html#return) 语句意味着生成器执行完成，并且将会引发一个StopIteration (https://docs.python.org/3/library/exceptions.html#StopItera

tion) 异常。返回值（如果有的话）是作为StopIteration (https://docs.python.org/3/library/exceptions.html#StopIteration) 构造方法的参数，并且会变成StopIteration.value属性。

yield 语句

```
yield_stmt ::=  yield_expression
```

yield (https://docs.python.org/3/reference/simple_stmts.html#yield) 语句在语义上等同于yield表达式 (https://docs.python.org/3/reference/expressions.html#yieldexpr) 。yield语法被用来省略括号，否则需要使用相同的yield表达式语句。例如，yield语句

```
yield <expr>
yield from <expr>
```

等同于yield表达式语句

```
(yield <expr>)
(yield from <expr>)
```

yield表达式和语句只有当在定义生成器 (https://docs.python.org/3/glossary.html#term-generator) 函数时被使用。只使用在生成器函数体中。在函数定义时使用yield，足够导致定义一个创建生成器函数而不是普通函数。

yield (https://docs.python.org/3/reference/simple_stmts.html#yield) 详细的语法，参考yield表达式 (https://docs.python.org/3/reference/expressions.html#yieldexpr) 部分。

raise 语句

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

如果不给出表达式，raise (https://docs.python.org/3/reference/simple_stmts.html#raise) 重新引发当前范围内上次引发的异常。如果当前范围没有可用的异常，会抛出RuntimeError (https://docs.python.org/3/library/exceptions.html#RuntimeError) 异常，暗示出现错误。

否则，raise (https://docs.python.org/3/reference/simple_stmts.html#raise) 使用第一个表达式作为异常对象求值。 对异常对象的第一个表达式求值。他必须是一个子类或者一个BaseException (https://docs.python.org/3/library/exceptions.html#BaseException) 的实例。如果是一个类，当实例化一个无参数类被需要的时候,异常实例将被获取，

异常的*类型*是异常实例类，*值*是实例本身。

traceback对象通常被自动创建，当异常发生并且被附加到它的__ traceback __ *属性上。你可以创建异常并设置你自己的traceback，在第一步使用异常方法with* _traceback()（这个方法返回相同的异常实例，traceback设置到他的参数中），就像：

```
raise Exception("foo occurred").with_traceback(tracebackobj)
```

from子句用于异常链接:如果给出，则第二个表达式必须是另一个异常类或实例，然后将附加到的 __cause__ 属性（它是可写）引发的异常。如果不处理引发的异常，则将打印两个例外情况：

```
>>>>>>>>> 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>
ZeroDivisionError: int division or modulo by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

一个类似的机制隐式工作如果里面的异常处理程序引发的异常或 finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 子句： 那么前一个异常附加作为新异常 __context__ 属性：

```
>>>>>>>>> try:
... print(1 / 0)
... except:
... raise RuntimeError("Something bad happened")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: int division or modulo by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

更多异常信息见异常 (https://docs.python.org/3/reference/executionmodel.html#exceptions) 章节，处理异常的信息Try语句 (https://docs.python.org/3/reference/compound_stmts.html#try) 章节。

break 语句

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

break (https://docs.python.org/3/reference/simple_stmts.html#break) 在句法上只能出现在for (https://docs.python.org/3/reference/compound_stmts.html#for) 或while (https://docs.python.org/3/reference/compound_stmts.html#while) 循环中, 但不能出现在循环中的函数定义或类定义中。

它中断最内层的循环, 跳过其可选的else (https://docs.python.org/3/reference/compound_stmts.html#else) 语句(如果有的话)。

如果for (https://docs.python.org/3/reference/compound_stmts.html#for) 循环被break (https://docs.python.org/3/reference/simple_stmts.html#break) 中断, 它的循环控制对象还保持当前值。

当break (https://docs.python.org/3/reference/simple_stmts.html#break) 放在具有finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 子句的try (https://docs.python.org/3/reference/compound_stmts.html#try) 语句中, finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 子句中的语句会在循环真正退出之前执行一次。

continue 语句

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

continue (https://docs.python.org/3/reference/simple_stmts.html#continue) 在句法上只能出现在for (https://docs.python.org/3/reference/compound_stmts.html#for) 或while (https://docs.python.org/3/reference/compound_stmts.html#while) 循环中, 但不能出现在循环中的函数定义或类定义，或在循环内部的finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 子句。 它重新开 始最内层的循环。

当continue (https://docs.python.org/3/reference/simple_stmts.html#continue) 通行控制离开一个包含finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 子句的try (https://docs.python.org/3/reference/compound_stmts.html#try) 语句，finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 子句执行之前真的开始下一个循环周期。

import 语句

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

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

基本的import语句（不是from (https://docs.python.org/3/reference/simple_stmts.html#from) 子句）执行下面两个步骤：

1. 查找module，如果需要加载并且初始化它。
2. 定义一个或多个名称在本地命名空间，在import (https://docs.python.org/3/reference/simple_stmts.html#import) 语句出现的范围内。

当语句包含多个子句（由逗号分隔）分别对每个子句实施两个步骤，就像子句被分隔成个体的import语句。

详细的第一步，查找并加载模块更详细的描述在import system (https://docs.python.org/3/reference/import.html#importsystem) 部分，它同样描述可以被导入的模块和各种类型的包，以及所有的钩子可用于自定义导入系统。请注意在此步骤中的失败可能表明该模块无法找到，或者模块初始化错误，包含可执行模块的代码错误。

如果请求的模块检测到成功，它将提供局部空间中的三种方式中的一种：

- 如果模块名后跟as (https://docs.python.org/3/reference/compound_stmts.html#as) ，那么这个名字后as会直接绑定到导入的模块。
- 如果没有指定其他名字，并且被导入的是顶层模块，模块名被绑定在本地的命名空间作为引用导入模块。
- 如果导入不是顶层模块，那么包含该模块的顶层包名被绑定到本地命名空间，作为一个顶级包的引用。导入的模块，必须可以使用完全限定名称访问而不是直接访问。

from (https://docs.python.org/3/reference/simple_stmts.html#from) 方式的使用过程稍微复杂：

1. 查找from (https://docs.python.org/3/reference/simple_stmts.html#from) 子句中指定的模块，如果需要加载并初始化它。
2. 对于每一个import子句的指定标识符：
    1. 检查import的模块具有该属性的名称
    2. 如果没有，尝试根据名称导入子模块，然后在导入的模块中再次检查该属性。
3. 如果属性没有找到，抛出ImportError (https://docs.python.org/3/library/exceptions.html#ImportError) 异常。

4. 否则，在本地命名空间中存储这个引用值，如果存在使用as (https://docs.python.org/3/reference/compound_stmts.html#as) 子句里的名称，否则使用属性名

Examples: 例子：

```
import foo               # foo imported and bound locally
import foo.bar.baz       # foo.bar.baz imported, foo bound locally
import foo.bar.baz as fbb  # foo.bar.baz imported and bound as fbb
from foo.bar import baz   # foo.bar.baz imported and bound as baz
from foo import attr      # foo imported and foo.attr bound as attr
```

如果标识符列表由星号('*')取代,该模块中定义的所有公共名称，都在import (https://docs.python.org/3/reference/simple_stmts.html#import) 语句所在的本地命名空间中被约束。

一个模块所定义的“*公共名称*”通过检查该模块的命名空间中的名为 `__all__` 的变量决定。如果(该变量)有定义，它必须是一个字符串的有序序列，这些字符串是由该模块定义或者导入的名字。在 `__all__` 中给出的名字都被认为是公共的且要求其存在。如果 `__all__` 没有定义，(该模块的)公共名字的集合就包含所有在该模块的名字空间中找到的，不以下划线(” _ ”)起首的所有名字。

导入的通配符形式 – `from module import *` – 只允许模块级别。试图在类或者函数中使用它会抛出SyntaxError (https://docs.python.org/3/library/exceptions.html#SyntaxError) 。

在指定你要导入的模块时，你并不一定要指定模块的绝对名字。当一个模块或包是在其他包之内时，在同一个顶层包下，我们有可能使用不需指明包名的相对加载。通过在from (https://docs.python.org/3/reference/simple_stmts.html#from) 之后的模块或者包之前指定几个句号, 就可以在没有精确名字的情况下，指定在当前包层次中向上搜索多少层。一个句号代表包括导入操作所在模块的当前包。两个句号代表上一个包层次，三个句号代表再上一个包层次,依次类推。所以,如果你在 pkg 包中执行 from.import mod ，你就会导入pkg.mod。如果你从包 pkg.subpkg1 执行 from ..subpkg2 import mod ，你就会导入 pkg.subpkg2.mod。相对导入的规范参见PEP328 (http://www.python.org/dev/peps/pep-0328)

函数importlib.import_module() (https://docs.python.org/3/library/importlib.html#importlib.import_module) 用于支持要确定哪些模块需要动态加载的应用程序。

future 语句

*future语句*是一个编译器指令,在指定的将来的Python版本应该可以使用语法或语义去编译一个特定的模块，到那时Python特性将成为标准。

future语句旨在缓解未来版本Python的迁移，这类版本会引入语言不兼容的变化。在发布前它允许在每个模块的基础上使用新特性，发布后这个特性将成为标准。

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

future语句必须出现在模块的顶部。唯一能出现在future前面的语句是:

- 该模块的文档字符串（如果有），
- 注释,
- 空白行，和
- 其它future语句。

Python 3.0 版本公认的特性是 absolute_import，division，generators，unicode_literals，print_function，nested_scopes 和 with_statement。他们都是多余的,因为他们总是在启用状态,同时只保持向后兼容的可编译性。

在编程的时候future语句被识别出来，并且被特殊地对待：更改核心构建的语义通常是通过生成不同的代码来实现的。甚至有可能一个新的功能会引入一个新的不兼容的语法（比如一个新的保留字），在这种情况下，编译器可能需要用区别对待的方式解析模块。此类决定不能被推迟到运行时。

对于任何发布的版本，编译器明确已经被定义的功能名称，如果future语句包含未知功能时，编译器会提示编译出错。

直接运行语义作为任何导入语句都是相同的：这里有一个标准的 __future__ (https://docs.python.org/3/library/__future__.html#module-__future__) 模块，稍后会解释，当future语句被执行的时候，将以通用方式被引入。

令人关注的运行时语义取决于被future语句启用的特定功能。

请注意，没有什么特别的语句。

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

那个不是future语句;这是一个普通的导入语句没有特殊的语义或语法的限制。

通过调用内置函数exec() (https://docs.python.org/3/library/functions.html#exec) 和compile() (https://docs.python.org/3/library/functions.html#compile) 来编译的代码，此代码在M模块发起，并包含future语句，在默认情况下，会结合future语句采用新的语法和语义。这可以由可选参数来控制去compile() (https://docs.python.org/3/library/functions.html#compile) ——有关详细信息,请参阅文档的功能。

在交互式解析器提示符下键入future语句，重启解析器器会话生效。如果解析器以-i选项开始，且是通过一个脚本名去执行，同时这个脚本包含future语句，它将作用在交互会话中，在脚本执行后开始。

> 参见:
> PEP 236 (http://www.python.org/dev/peps/pep-0236) - Back to the __future__
> The original proposal for the __future__ mechanism.

global 语法

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

global (https://docs.python.org/3/reference/simple_stmts.html#global) 语句是对整个代码块都有作用的一个声明. 它指出其所列的标识符要解释为全局的.如果某名字在本地命名空间中没有定义, 就自动使用相应的全局名字. 没有global (https://docs.python.org/3/reference/simple_stmts.html#global) 是不可能手动指定一个名字是全局的。

在global (https://docs.python.org/3/reference/simple_stmts.html#global) 中出现的名字不能在global (https://docs.python.org/3/reference/simple_stmts.html#global) 之前的代码中使用.

在global (https://docs.python.org/3/reference/simple_stmts.html#global) 中出现的名字不能作为形参, 不能作为for (https://docs.python.org/3/reference/compound_stmts.html#for) 循环的控制对象, 不能在类 (https://docs.python.org/3/reference/compound_stmts.html#class) 定义, 函数定义, import (https://docs.python.org/3/reference/simple_stmts.html#import) 语句中出现.

CPython详细实现：当前实现不强制两个限制，但程序不应该滥用这种自由，作为将来的实现可能会强制他们或默默地改变程序的意义。

程序员注意: global (https://docs.python.org/3/reference/simple_stmts.html#global) 是一个解析器的指示字. 它仅仅会对和global (https://docs.python.org/3/reference/simple_stmts.html#global) 一起解析的代码有效，尤其是，一个global (https://docs.python.org/3/reference/simple_stmts.html#global) 语句被包含在字符串或者代码对象提供内建的exec() (https://docs.python.org/3/library/functions.html#exec) 函数，并不影响包含该函数调用的代码块，相同的机制也应用于eval() (https://docs.python.org/3/library/functions.html#eval) 和compile() (https://docs.python.org/3/library/functions.html#compile) 函数。

nonlocal 语句

```
nonlocal_stmt ::=  "nonlocal" identifier ("," identifier)*
```

nonlocal (https://docs.python.org/3/reference/simple_stmts.html#nonlocal) 语句引发列出的标识符，适用于不包括global之外最近的封闭范围，前一个绑定变量。这是重要的因为绑定的默认行为是首先搜索本地命名空间。该语句允许封装的代码绑定除了global（模块）范围之外，局部作用域的变量。

在nonlocal (https://docs.python.org/3/reference/simple_stmts.html#nonlocal) 语句不同于那些在global语句中列出的名称，必须引用到预先存在的封闭范围中的绑定（不能明确的确定要在其中创建一个新绑定的范围）。

在nonlocal (https://docs.python.org/3/reference/simple_stmts.html#nonlocal) 语句中列出的名字，在本地范围必须不和已经存在的绑定冲突。

> 参见:
> PEP 3104 (http://www.python.org/dev/peps/pep-3104) – Access to Names in Outer Scopes
> The specification for the nonlocal (https://docs.python.org/3/reference/simple_stmts.html#nonlocal) statement.

8

复合语句

复合语句中包含了其他语句（语句组）；它以某种方式影响或控制着其他语句的执行。一般来讲，复合语句会跨越多个行，然而一个完整的复合语句也可以简化在一行中。

if (https://docs.python.org/3/reference/compound_stmts.html#if)，while (https://docs.python.org/3/reference/compound_stmts.html#while) 及for (https://docs.python.org/3/reference/compound_stmts.html#for) 语句实现了传统的流控制机制。try (https://docs.python.org/3/reference/compound_stmts.html#try) 语句为一组语句指定了异常处理器和/或资源清除代码，with (https://docs.python.org/3/reference/compound_stmts.html#with) 表达式允许在代码块上下文执行代码初始化并做后续处理。函数及类的定义也被看作是复合语句。

复合语句由一个或多个‘子句’组成。一个子句由一个头部和一个‘代码序列’组成。特定复合语句的子句头具有相同的缩进层次。每个子句头均以一个唯一的标识关键字开始，并以一个冒号结束。一个语句序列是由子句控制的一组语句。一个语句序列可以包含一个或多个以分号分隔且与子句头同行的语句。或者它也可以是一个或多个在后续各行中缩进的语句。只有在后者的情况下子句序列允许包括有嵌套的复合语句，一下形式是非法的，这样限制原因是 if (https://docs.python.org/3/reference/compound_stmts.html#if) 后面有 else (https://docs.python.org/3/reference/compound_stmts.html#else) 子句的话，会导致语义不明确。

```
if test1: if test2: print(x)
```

还需要注意的是，在这样的上下文中，分号的优先级比冒号的高，所以在下面的例子中，要么所有的 print() (https://docs.python.org/3/library/functions.html#print) 方法都会被执行，要么所有方法都不会被执行。

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

总结：

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

注意以 NEWLINE 结尾的语句可能后缀一个 DEDENT。同时需要注意的是，可选的续行子句通常以某个不能开始一个语句的关键字开头，因此这里没有歧义（‘不定义的 else (https://docs.python.org/3/reference/compound_stmts.html#else)’的问题已经由 Python 根据对嵌套的语句的缩进要求解决掉了）。

为了能够叙述清楚，以下章节中的每个子句的语法规则格式都被分行说明。

if 语句

if (https://docs.python.org/3/reference/compound_stmts.html#if) 语句用于条件性执行：

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

通过依次计算每个表达式的值，直到找到表达式为 true 的值时，它会准确地选择执行相应一个语句序列（对真和假的定义参见 Boolean operations (https://docs.python.org/3/reference/expressions.html#booleans) 章节）；然后该语句序列被执行（ if (https://docs.python.org/3/reference/compound_stmts.html#if) 语句的其它部分不会被执行或计算）。如果表达式的值都为 false，并且存在 else (https://docs.python.org/3/reference/compound_stmts.html#else) 子句，则 else 子句被执行。

while 语句

只要条件表达式的值为 true，while (https://docs.python.org/3/reference/compound_stmts.html#while) 语句会重复执行某个代码段：

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

在上例中，会重复计算表达式(expression)的值，并且如果表达式为真就执行第一个语句序列(suit)；如果为假（可能是在执行第一次计算时）就会执行 else (https://docs.python.org/3/reference/compound_stmts.html#else) 子句（如果给出的话），并退出循环。

第一个语句序列中的 break (https://docs.python.org/3/reference/simple_stmts.html#break) 语句可以实现不执行 else (https://docs.python.org/3/reference/compound_stmts.html#else) 子句而直接退出循环。第一个语句序列中的 continue (https://docs.python.org/3/reference/simple_stmts.html#continue) 子句可以跳过该子句的其余部分，直接进入下次表达式的计算。

for 语句

for (https://docs.python.org/3/reference/compound_stmts.html#for) 语句用于迭代有序序列或其他可迭代对象的元素（比如字符串，数组或列表）。

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

表达式列表（expression list）仅被计算一次，它应该生成一个可迭代的对象。为 `expression_list` 的结果创建一个迭代器。对于迭代器中的每一个元素，语句序列都会以迭代器返回的结果为序执行一次。每个元素使用标准的赋值规则（详见 Assignment statements (https://docs.python.org/3/reference/simple_stmts.html#assignment)）依次赋给循环控制对象表，然后执行语句序列。元素迭代完后（当迭代序列为空或迭代器抛出 StopIteration (https://docs.python.org/3/library/exceptions.html#StopIteration) 异常），执行语句序列中的 else (https://docs.python.org/3/reference/compound_stmts.html#else) 子句（如果存在）然后循环终止。

第一个语句序列中的 break (https://docs.python.org/3/reference/simple_stmts.html#break) 语句可以实现不执行 else (https://docs.python.org/3/reference/compound_stmts.html#else) 子句而终止循环。在第一个语句序列中的 continue (https://docs.python.org/3/reference/simple_stmts.html#continue) 语句可以跳过该子句的其余部分，直接进行下个元素的迭代计算，或者当迭代完成后进入 else (https://docs.python.org/3/reference/compound_stmts.html#else) 子句。

for循环语句序列可以对循环控制对象列表中的变量赋值。这将覆盖所有以前分配给那些变量的值，包括for循环中的语句序列中的变量：

```
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
```

循环结束时循环控制对象列表中的名字并未删除，但是如果序列为空，它在循环中根本不会被赋值。提示：内建函数 range() (https://docs.python.org/3/library/stdtypes.html#range) 返回一个整数列表，可以用于模拟 Pascal 语言中的 `for i := a to b do` 的行为；例如 `list(range(3))` 返回列表 `[0,1,2]`。

> 注意：如果序列对象在循环过程中被修改（只有可变类型会出现这种情况，例如列表），这里有一些需要注意的地方。有一个内部计数器用于跟踪下一轮循环使用的元素，并且每迭代一次便增加一次。当这个计数器的值达到了序列的长度时循环终止。这就意味着如果从语句序列中删除当前（或前一个元素）元素，下一个元素会被跳过而不被执行（因为当前索引值的元素已经处理过了）。另一方面，如果在当前元素前插入一个元素，下一轮循环时当前元素会被再次重复处理。这会导致难以察觉的错误，但可以通过使用含有整个有序类型对象的片段而生成的临时拷贝避免这个问题，例如：

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

try 语句

try (https://docs.python.org/3/reference/compound_stmts.html#try) 语句可以为一组语句指定异常处理器和/或资源清理代码：

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

except (https://docs.python.org/3/reference/compound_stmts.html#except) 子句指定了一个或多个异常处理器。 如果在 try (https://docs.python.org/3/reference/compound_stmts.html#try) 子句中未捕获任何异常，则异常处理器不会被执行。当 try (https://docs.python.org/3/reference/compound_stmts.html#try) 子句中有异常捕获时，就会开始查找异常处理器。它会依次查找异常处理子句，直到找到能够匹配该异常的子句。如果存在未指定异常的 except 语句，则必须放在最后，它会匹配任何异常。当 except 子句中声明了异常类型时，该类型表达式的值会被计算. 如果结果对象与该异常匹配, 那么该子句就匹配了这个异常。只有满足以下条件才认为一个对象匹配某个异常：1、该对象是异常对象本身或其基类；2、该对象是一个数组，包含了一个与该异常兼容的对象。

如果没有 except 子句能够匹配异常，将会在调用栈 [1 (https://docs.python.org/3/reference/compound_stmts.html#id5)] 的外围代码中继续查找异常处理器。

如果在 except 子句头部计算表达式时就引发了异常, 原来的异常处理器查找工作就会被中断, 并在外层代码及调用栈搜索新的异常处理器(就好像处理整个 try (https://docs.python.org/3/reference/compound_stmts.html#try) 语句发生了异常一样)。

当找到了一个匹配的 except 子句时，异常被赋值给 except 子句中 as (https://docs.python.org/3/reference/compound_stmts.html#as) 关键字后指定的对象（如果存在），并且执行该 except 语句序列。所有 except 子句都必须包含可执行的代码块。当该代码块执行结束后, 会转到整个 try 语句之后继续正常执行(这意味着, 如果有两个嵌套的异常处理器用于捕获同一个异常, 并且异常由内层的处理器处理, 那么外层处理器就不会响应这个异常)。

当使用 `as target` 关键词给异常赋值时，该 target 会在 except 子句结束后被销毁。例如：

```
except E as N:
foo
```

可以解释为：

```
except E as N:
try:
    foo
finally:
    del N
```

这就意味着必须给异常指派一个不同的名称，以保证在 except 子句结束后可以索引到它。这些异常最后会被清除，因为这些异常与堆栈信息会绑定在一起，他们与栈帧形成了循环引用。直到下次垃圾回收发生前，所有这些栈中的局部变量都是存活的。

except 子句被执行前，该异常的详细信息存储在 sys (https://docs.python.org/3/library/sys.html#module-sys) 模块中，可以通过 sys.exc_info() (https://docs.python.org/3/library/sys.html#sys.exc_info) .sys.exc_info() (https://docs.python.org/3/library/sys.html#sys.exc_info) 方法读取，该方法返回由异常类、异常实例以及能够标识程序中异常发生位置的堆栈信息（详见 The standard type hierarchy (https://docs.python.org/3/reference/datamodel.html#types) ）组成的三元组组成。当返回发生异常的函数时，sys.exc_info() (https://docs.python.org/3/library/sys.html#sys.exc_info) 的值会恢复调用之前的值。

当程序控制器从 try (https://docs.python.org/3/reference/compound_stmts.html#try) 子句中执行结束后，可以执行可选的 else (https://docs.python.org/3/reference/compound_stmts.html#else) 子句。[2 (https://docs.python.org/3/reference/compound_stmts.html#id6)]在 else (https://docs.python.org/3/reference/compound_stmts.html#else) 子句中引发的异常不会在前面的 except (https://docs.python.org/3/reference/compound_stmts.html#except) 子句中得到处理。

如果存在 finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 块，它是一个资源“清理”处理器。当执行try (https://docs.python.org/3/reference/compound_stmts.html#try) 代码块，包括 except (https://docs.python.org/3/reference/compound_stmts.html#except) 与 else (https://docs.python.org/3/reference/compound_stmts.html#else) 过程中有异常发生且未被处理时，这些异常就会被临时保存下来。finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 块会被执行，如果有临时保存的异常的话，该异常会在 finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 块执行结束后被重新抛出。如果 finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 块产生了另一个异常，已被保存的异常会作为新异常的上下文保存下来。如果 finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 块中执行了 return (https://docs.python.org/3/reference/simple_stmts.html#return) 或 break (https://docs.python.org/3/reference/simple_stmts.html#break) 语句，则忽略已保存的异常：

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

```
...     finally:
...         return 42
...
>>> f()
42
```

在执行 finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 块时，程序的异常信息是无效的。

try (https://docs.python.org/3/reference/compound_stmts.html#try) ...finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 语句中，try (https://docs.python.org/3/reference/compound_stmts.html#try) 代码块中的 return (https://docs.python.org/3/reference/simple_stmts.html#return) ，break (https://docs.python.org/3/reference/simple_stmts.html#break) 或者 continue (https://docs.python.org/3/reference/simple_stmts.html#continue) 执行后，finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 块也会被执行。在 finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 块中不允许出现 continue (https://docs.python.org/3/reference/simple_stmts.html#continue) 子句（该问题是由当前实现导致的——这个限制可能会在后续的版本中去掉）。

函数的返回值由程序最后的 return (https://docs.python.org/3/reference/simple_stmts.html#return) 语句的执行结果决定。由于 finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 块一定会被执行，所以finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 块中的 return (https://docs.python.org/3/reference/simple_stmts.html#return) 语句一定是最后才会被执行的。

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

更多关于异常的信息可以查看 Exceptions (https://docs.python.org/3/reference/executionmodel.html#exceptions) 章节，关于使用 raise (https://docs.python.org/3/reference/simple_stmts.html#raise) 语句抛出异常的详细信息可以从 The raise statement (https://docs.python.org/3/reference/simple_stmts.html#raise) 部分找到。

with 语句

with (https://docs.python.org/3/reference/compound_stmts.html#with) 语句使用上下文管理器中定义的方法将执行的代码块包裹起来（详见 With Statement Context Managers (https://docs.python.org/3/reference/datamodel.html#context-managers) ）。为了便于重用，允许使用传统的 try (https://docs.python.org/3/reference/compound_stmts.html#try) ...except (https://docs.python.org/3/reference/compound_stmts.html#except) ...finally (https://docs.python.org/3/reference/compound_stmts.html#finally) 异常处理模式对该语句进行封装。

```
with_stmt ::=  "with" with_item ("," with_item)* ":" suite
with_item ::=  expression ["as" target]
```

使用 with (https://docs.python.org/3/reference/compound_stmts.html#with) 语句执行一个“元素”的处理过程如下：

1、根据上下文表达式（ with_item (https://docs.python.org/3/reference/compound_stmts.html#grammar-token-with_item) 中给出的表达式 ）的值得到一个上下文管理器。

2、加载上下文管理器的 __exit__() (https://docs.python.org/3/reference/datamodel.html#object.__exit__) 方法，后续使用。

3、调用上下文管理器的 __enter __()` (https://docs.python.org/3/reference/datamodel.html#object.__enter__) 方法。

4、如果 with (https://docs.python.org/3/reference/compound_stmts.html#with) 语句中包含一个对象，则将 __enter__() (https://docs.python.org/3/reference/datamodel.html#object.__enter__) 方法的返回值赋值给该对象。

> 注意：with (https://docs.python.org/3/reference/compound_stmts.html#with) 语句能够保证只要 __enter__() (https://docs.python.org/3/reference/datamodel.html#object.__enter__) 方法能够正确返回，则 __exit__()) (https://docs.python.org/3/reference/datamodel.html#object.__exit__) 方法一定会被调用。因此，如果在给对象列表赋值过程中发生了错误，对该错误的处理方式与语句序列中的错误处理方式是一致的。见步骤6。

5、执行语句序列。

6、上下文管理器的 __exit__() (https://docs.python.org/3/reference/datamodel.html#object.__exit__) 方法被调用。如果异常导致必须退出语句序列，其类型、值以及堆栈信息会作为参数传递给 __exit__() (http

s://docs.python.org/3/reference/datamodel.html#object.__exit__) 。否则，会给该方法传递三个 None (https://docs.python.org/3/library/constants.html#None) 参数。

如果由于异常导致语句退出，且 __exit__() (https://docs.python.org/3/reference/datamodel.html#object.__exit__) 的返回值为 false，该异常会被再次抛出。如果返回值为 true，则异常被抑制并且会继续执行 with (https://docs.python.org/3/reference/compound_stmts.html#with) 语句之后的代码。

如果由于其他非异常原因导致语句退出，则 __exit__() (https://docs.python.org/3/reference/datamodel.html#object.__exit__) 的返回值被忽略，且会从退出的正常位置继续执行。

上下文管理器会像处理多个嵌套 with (https://docs.python.org/3/reference/compound_stmts.html#with) 语句一样处理多个元素：

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

等价于

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

3.1 版本中新特性：支持多上下文表达式。

> 参见：
> PEP 0343 (http://www.python.org/dev/peps/pep-0343) – “with” 语句
> 详细介绍了Python 中 with (https://docs.python.org/3/reference/compound_stmts.html#with) 语句的特性，知识背景以及一些参考实例。

函数定义

函数定义，定义了一个用户自定义的函数对象（参见 The standard type hierarchy (https://docs.python.org/3/reference/datamodel.html#types) ）。

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

```
parameter     ::=  identifier [":" expression]
defparameter   ::=  parameter ["=" expression]
funcname      ::=  identifier
```

函数定义是一个可执行语句。它在当前局部命名空间中将函数名称与函数对象（函数的可执行代码的组合）捆绑在一起。该函数对象包括着一个全局命名空间的引用，以便在调用时使用。

函数定义不执行函数体；只有当函数被调用时才会执行函数体。[3 (https://docs.python.org/3/reference/compound_stmts.html#id7)]

函数的定义可能被若干 decorator (https://docs.python.org/3/glossary.html#term-decorator) 修饰表达式修饰。当函数被定义时在函数定义的范围内计算修饰表达式的值。其结果必须是一个回调，该回调作为函数对象的唯一参数被调用。返回值绑定函数名称，而不是绑定函数对象。多装饰被应用于嵌套方式。例如下方的代码：

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

等价于

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

当一个或多个 parameters (https://docs.python.org/3/glossary.html#term-parameter) （参数）包含形参 `=` 表达式时，这样的函数就称为具有“默认参数值”的函数。在调用有默认参数值参数的函数时，其对应的参数就可以忽略，这种情况下默认值会用于替代该参数。如果一个参数有默认值，该参数之后，`*` 之前的所有参数都必须有默认值——这是一个在语法中未表述的句法限制。

默认参数值在函数定义被执行时从左至右计算。这意味着在函数定义时，该表达式仅被执行一次，且在每次调用时都使用相同的“预计算”值。这在理解默认参数值是一个像列表或者字典这样的可变对象时尤其值得注意。如果函数修改了这个对象（例如给一个列表中追加了一个元素），默认值也随之被修改。这显然不是我们期望的。一个避免这个问题的方法就是使用 `None` 作为默认值，并在函数体中做显式测试，例如：

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

函数调用语义的详细介绍请参见 Call (https://docs.python.org/3/reference/expressions.html#calls) 章节。通常一个函数调用会给所有出现在参数列表中的参数赋值，要么通过位置参数，或者通过关键字参数，或者通过默认值。如果参数列表中包含 `*identifier` 形式，它会被初始化为一个数组用于接收所有额外的位置参数，其默认为空数组。如果参数列表中包含 `**identifier` 形式，它会被初始化为一个新的字典用于接收所有二外的关键

字参数，其默认值为一个空字典。`*` 或 `*identifier` 后的参数为 keyword-only 参数，只能通过关键字参数传值。

参数列表可能包含“参数名称后紧跟 `: expression` ”形式的注解。所有参数都有注解，甚至 `*identifier` 或 `**identifier` 形式的参数也有注解。函数参数列表后可能包含 `-> expression` 形式的“return”注解。这些注解可以是任意合法的 Python 表达式，并且会在函数定义执行时计算这些表达式的值。注解的计算顺序可能会与其在源代码中出现的书序不同。注解的出现不会改变一个函数的语义。函数对象的 _ _ *annotations* _ _属性中的参数名称可以作为字典的 key，其对应的 value 为注解的值。

也可以创建能直接在表达式中使用的匿名函数（未与名字绑定的函数）。这是通过 lambda 表达式实现的，详见 Lambdas (https://docs.python.org/3/reference/expressions.html#lambda) 章节。注意 lambda 仅仅是一个简单函数的缩写形式；以“def (https://docs.python.org/3/reference/compound_stmts.html#def) ”定义的函数可以被传递或赋予一个新的名字，就像以 lambda 表达式定义的函数一样。以“def (https://docs.python.org/3/reference/compound_stmts.html#def) ”形式定义的函数功能要更强大些，因为它允许执行多条语句及注解。

程序员注意：函数是一类对象。在函数定义中执行“ `def` ”，定义了一个可返回或传递的局部函数。在嵌套函数中使用的自由变量可以获取包含 def 的函数对象的局部变量。详见 Naming and binding (https://docs.python.org/3/reference/executionmodel.html#naming) 章节。

> 参见：
> PEP 3107 (http://www.python.org/dev/peps/pep-3107) – 函数注解
> 函数注解的原始规范。

类定义

一个类定义定义了一个类对象（见 [The Standard type hierarchy] (https://docs.python.org/3/reference/datamodel.html#types)）：

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

一个类定义是一个可执行的语句。继承关系表通常会给出一个基类列表（获取更多高级应用可以参见 Customizing class creation (https://docs.python.org/3/reference/datamodel.html#metaclasses) ），因此继承关系列表中的每个元素都生成一个允许子类化的类对象。在继承关系列表中未包含的类默认继承自 object (https://docs.python.org/3/library/functions.html#object) 类；因此，

```
class Foo:
pass
```

等价于

```
class Foo(object):
pass
```

然后类的语句序列在一个新的堆栈结构中（见 Naming and binding (https://docs.python.org/3/reference/executionmodel.html#naming) ）使用新创建的局部命名空间以及原始的全局命名空间执行。（通常情况下，该套件包含的大多是函数定义。）当该类的语句序列执行结束后就丢弃其执行堆栈，但是它的局部命名空间会被保存下来。[4 (https://docs.python.org/3/reference/compound_stmts.html#id8)]然后使用其继承关系表创建基类，将保存下来的命名空间作为属性字典创建新的类对象。在原始的命名空间中，类的名称会被绑定在这个类对象上。

类的创建可以使用 metaclasses (https://docs.python.org/3/reference/datamodel.html#metaclasses) 大量定制。

类中也可以加修饰符：就像修饰函数一样，

```
@f1(arg)
@f2
class Foo: pass
```

等价于

```
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

类修饰符表达式的值计算方式与函数修饰符相同。其结果必须是一个类对象，并将该类对象与一个类名绑定。

> 程序员注意：类定义中定义的变量都是类的属性；类的实例共享这些属性。实例属性可以通过 `self.name = value` 的方式赋值。类属性及实例属性都可以通过 `self.name` 获取，并且如果使用这种方式属性的话，如果实例属性与类属性由相同的名称，那么实例属性会覆盖类属性。类属性可以作为实例属性的默认值，但是使用可变值可能会导致不可预期的结果。Descriptors (https://docs.python.org/3/reference/datamodel.html#descriptors) 可以用于创建包含不同实现细节的实例变量。

参见： PEP 3115 (http://www.python.org/dev/peps/pep-3115) – Python 3 中的 Metaclasses PEP 3129 (http://www.python.org/dev/peps/pep-3129) – 类修饰符

脚注

[1 (https://docs.python.org/3/reference/compound_stmts.html#id1)] 该异常会传递到调用堆栈中，除非在 finally 代码块中碰巧抛出了另一个异常。这个新异常会导致旧异常丢失。

[2 (https://docs.python.org/3/reference/compound_stmts.html#id2)] 在遇到异常或者执行 return () ，continue () 或 break () 语句的时，程序控制器会离开 except 块。

[3 (https://docs.python.org/3/reference/compound_stmts.html#id3)] 一个字符串文本作为函数体的第一段语句出现时会被转换为函数的 __doc__ 属性以及函数的 [docstring] (https://docs.python.org/3/glossary.html#term-docstring) 。

[4 (https://docs.python.org/3/reference/compound_stmts.html#id4)] 一个字符串文本作为类中的第一段语句出现时会被转换为命名空间的 __doc__ 元素以及类的docstring (https://docs.python.org/3/glossary.html#term-docstring) 。

9

顶层组件

Python解释器的可以有几种输入源：从标准输入或程序参数传入的脚本，交互方式下的输入，从模块源文件，等等。本章给出在这些情况下使用的语法。

完整的 Python 程序

尽管语法规格说明不需要指明语言的解释器是如何执行的，对一个完整的Python程序的了解也是有用的。一个完整的Python程序运行在一个最低限度的初始环境中：所有内置和标准模块都是可用的，但都没有被初始化，除了 sys (https://docs.python.org/3/library/sys.html#module-sys) (各种系统服务)模块、builtin (https://docs.python.org/3/library/builtins.html#module-builtins) (内置函数、异常和None)模块和 __main__ (https://docs.python.org/3/library/__main__.html#module-__main__) 模块。main被用来为完整程序的运行提供局部和全局名字空间。

针对于文件输入来说的完整的 Python 程序语法，在下一节给出描述。

解释器也可以以交互方式运行；在这种情况下，它并不读取和运行一个完整程序，而是一次读取和运行一条语句(可能是复合语句)。这种初始环境与完整程序环境是相同的；每条语句都是在 __main__ (https://docs.python.org/3/library/__main__.html#module-__main__) 名字空间下运行。

在Unix上，一个完整程序可以以三种形式传给解释器：使用-c字符串命令行选项，以一个文件作为命令行的第一个参数，或作为标准输入。如果文件或标准输入是一个tty(终端)设备，解释器进行交互模式；否则，它把文件作为一个完整程序来运行。

文件输入

所有从非交互文件的输入读取具有相同的形式：

file_input ::= (NEWLINE | statement (https://docs.python.org/3/reference/compound_stmts.html#grammar-token-statement))*

这个语法用于以下的情况：

- 当解析一个完整Python程序时(从文件或字符串中)；
- 当解析一个模块时；
- 当解析一个传给exec() (https://docs.python.org/3/library/functions.html#exec) 语句的字符串时；

交互式输入

交互模式输入使用以下语法进行解析：

interactive_input ::= [stmt_list (https://docs.python.org/3/reference/compound_stmts.html#grammar-token-stmt_list)] NEWLINE | compound_stmt (https://docs.python.org/3/reference/compound_stmts.html#grammar-token-compound_stmt) NEWLINE

请注意一个(顶层)复合语句后面在交互模式下必须跟着一个空行；需要用它来帮助解释器检测输入的结束。

表达式输入

eval()用于表达式的输入。它将忽略前面的空格。Eval()的字符串参数必须具有以下形式：

eval_input ::= expression_list (https://docs.python.org/3/reference/expressions.html#grammar-token-expression_list) NEWLINE*

10

完整的语法规范

这是一个完整的 Python 语法，它由解析器生成器读取，并用于解析 Python 源文件。

``` # Grammar for Python

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

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

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

decorator: '@' dotted_name ['(' [arglist] ')'] NEWLINE
decorators: decorator+
decorated: decorators (classdef | funcdef)
funcdef: 'def' NAME parameters ['->' test] ':' suite
parameters: '(' [typedargslist] ')'
typedargslist: (tfpdef ['=' test] (',' tfpdef ['=' test])* [','
  ['*' [tfpdef] (',' tfpdef ['=' test])* [',' '**' tfpdef] | '**' tfpdef]]
 | '*' [tfpdef] (',' tfpdef ['=' test])* [',' '**' tfpdef] | '**' tfpdef)
tfpdef: NAME [':' test]
varargslist: (vfpdef ['=' test] (',' vfpdef ['=' test])* [','
  ['*' [vfpdef] (',' vfpdef ['=' test])* [',' '**' vfpdef] | '**' vfpdef]]
 | '*' [vfpdef] (',' vfpdef ['=' test])* [',' '**' vfpdef] | '**' vfpdef)
vfpdef: NAME

stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | del_stmt | pass_stmt | flow_stmt |
 import_stmt | global_stmt | nonlocal_stmt | assert_stmt)
expr_stmt: testlist_star_expr (augassign (yield_expr|testlist) |
 ('=' (yield_expr|testlist_star_expr))*)
testlist_star_expr: (test|star_expr) (',' (test|star_expr))* [',']
augassign: ('+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' |
```
```

```
'<<=' | '>>=' | '**=' | '//=')
# For normal assignments, additional restrictions enforced by the interpreter
del_stmt: 'del' exprlist
pass_stmt: 'pass'
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist]
yield_stmt: yield_expr
raise_stmt: 'raise' [test ['from' test]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
import_from: ('from' (('.' | '...')* dotted_name | ('.' | '...')+)
  'import' ('*' | '(' import_as_names ')' | import_as_names))
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)* [',']
dotted_as_names: dotted_as_name (',' dotted_as_name)*
dotted_name: NAME ('.' NAME)*
global_stmt: 'global' NAME (',' NAME)*
nonlocal_stmt: 'nonlocal' NAME (',' NAME)*
assert_stmt: 'assert' test [',' test]

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

test: or_test ['if' or_test 'else' test] | lambdef
test_nocond: or_test | lambdef_nocond
lambdef: 'lambda' [varargslist] ':' test
lambdef_nocond: 'lambda' [varargslist] ':' test_nocond
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
```

```
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
# <> isn't actually a valid comparison operator in Python. It's here for the
# sake of a __future__ import described in PEP 401
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
star_expr: '*' expr
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
power: atom trailer* ['**' factor]
atom: ('(' [yield_expr|testlist_comp] ')' |
   '[' [testlist_comp] ']' |
   '{' [dictorsetmaker] '}' |
   NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False')
testlist_comp: (test|star_expr) ( comp_for | (',' (test|star_expr))* [','] )
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
subscriptlist: subscript (',' subscript)* [',']
subscript: test | [test] ':' [test] [sliceop]
sliceop: ':' [test]
exprlist: (expr|star_expr) (',' (expr|star_expr))* [',']
testlist: test (',' test)* [',']
dictorsetmaker: ( (test ':' test (comp_for | (',' test ':' test)* [','])) |
  (test (comp_for | (',' test)* [',']) )

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

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

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

yield_expr: 'yield' [yield_arg]
yield_arg: 'from' test | testlist
```

```
```