

Python 3.8 语言参考



本参考手册描述了 Python 的语法和“核心语义”。本参考是简洁的，但试图做到准确和完整。非必要的内建对象类型和内建函数、模块的语义描述在 Python 标准库中。有关该语言的非正式介绍，请参阅 Python 教程。对 C ...



下载手机APP
畅享精彩阅读

目 录

致谢

0. 介绍

1. 概述

1.1. 其他实现

1.2. 标注

2. 词法分析

2.1. 行结构

2.2. 其他形符

2.3. 标识符和关键字

2.4. 字面值

2.5. 运算符

2.6. 分隔符

3. 数据模型

3.1. 对象、值与类型

3.2. 标准类型层级结构

3.3. 特殊方法名称

3.3.1. 基本定制

3.3.2. 自定义属性访问

3.3.3. 自定义类创建

3.3.4. 自定义实例及子类检查

3.3.5. 模拟泛型类型

3.3.6. 模拟可调用对象

3.3.7. 模拟容器类型

3.3.8. 模拟数字类型

3.3.9. with 语句上下文管理器

3.3.10. 特殊方法查找

3.4. 协程

3.4.1. 可等待对象

3.4.2. 协程对象

3.4.3. 异步迭代器

3.4.4. 异步上下文管理器

4. 执行模型

4.1. 程序的结构

4.2. 命名与绑定

4.3. 异常

5. 导入系统

- 5.1. importlib
- 5.2. 包
- 5.3. 搜索
- 5.4. 加载
- 5.5. 基于路径的查找器
- 5.6. 替换标准导入系统
- 5.7. 包相对导入
- 5.8. 有关 main 的特殊事项
- 5.9. 开放问题项
- 5.10. 参考文献

6. 表达式

- 6.1. 算术转换
- 6.2. 原子
- 6.3. 原型
- 6.4. await 表达式
- 6.5. 幂运算符
- 6.6. 一元算术和位运算
- 6.7. 二元算术运算符
- 6.8. 移位运算
- 6.9. 二元位运算
- 6.10. 比较运算
- 6.11. 布尔运算
- 6.12. 条件表达式
- 6.13. lambda 表达式
- 6.14. 表达式列表
- 6.15. 求值顺序
- 6.16. 运算符优先级

7. 简单语句

- 7.1. 表达式语句
- 7.2. 赋值语句
- 7.3. assert 语句
- 7.4. pass 语句
- 7.5. del 语句
- 7.6. return 语句
- 7.7. yield 语句
- 7.8. raise 语句
- 7.9. break 语句
- 7.10. continue 语句

- 7.11. import 语句
- 7.12. global 语句
- 7.13. nonlocal 语句
- 8. 复合语句
 - 8.1. if 语句
 - 8.2. while 语句
 - 8.3. for 语句
 - 8.4. try 语句
 - 8.5. with 语句
 - 8.6. 函数定义
 - 8.7. 类定义
 - 8.8. 协程
- 9. 最高层级组件
 - 9.1. 完整的 Python 程序
 - 9.2. 文件输入
 - 9.3. 交互式输入
 - 9.4. 表达式输入
- 10. 完整的语法规范

致谢

当前文档《Python 3.8 语言参考》由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建,生成于 2019-11-26。

书栈网仅提供文档编写、整理、归类等功能,以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理,书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候,发现文档内容有不恰当的地方,请向我们反馈,让我们共同携手,将知识准确、高效且有效地传递给每一个人。

同时,如果您在日常工作、生活和学习中遇到有价值有营养的知识文档,欢迎分享到书栈网,为知识的传承献上您的一份力量!

如果当前文档生成时间太久,请到书栈网获取最新的文档,以跟上知识更新换代的步伐。

内容来源: [Python官网](https://docs.python.org/zh-cn/3/reference/index.html) <https://docs.python.org/zh-cn/3/reference/index.html>

文档地址: <http://www.bookstack.cn/books/python-3.8-reference>

书栈官网: <https://www.bookstack.cn>

书栈开源: <https://github.com/TruthHun>

分享,让知识传承更久远! 感谢知识的创造者,感谢知识的分享者,也感谢每一位阅读到此处的读者,因为我们都将成为知识的传承者。

Python 语言参考

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

- 0. 介绍
- 1. 概述
 - 1.1. 其他实现
 - 1.2. 标注
- 2. 词法分析
 - 2.1. 行结构
 - 2.2. 其他形符
 - 2.3. 标识符和关键字
 - 2.4. 字面值
 - 2.5. 运算符
 - 2.6. 分隔符
- 3. 数据模型
 - 3.1. 对象、值与类型
 - 3.2. 标准类型层级结构
 - 3.3. 特殊方法名称
 - 3.3.1. 基本定制
 - 3.3.2. 自定义属性访问
 - 3.3.3. 自定义类创建
 - 3.3.4. 自定义实例及子类检查
 - 3.3.5. 模拟泛型类型
 - 3.3.6. 模拟可调用对象
 - 3.3.7. 模拟容器类型
 - 3.3.8. 模拟数字类型
 - 3.3.9. with 语句上下文管理器
 - 3.3.10. 特殊方法查找
 - 3.4. 协程
 - 3.4.1. 可等待对象
 - 3.4.2. 协程对象
 - 3.4.3. 异步迭代器
 - 3.4.4. 异步上下文管理器
- 4. 执行模型
 - 4.1. 程序的结构
 - 4.2. 命名与绑定
 - 4.3. 异常
- 5. 导入系统
 - 5.1. importlib
 - 5.2. 包
 - 5.3. 搜索

- 5.4. 加载
 - 5.5. 基于路径的查找器
 - 5.6. 替换标准导入系统
 - 5.7. 包相对导入
 - 5.8. 有关 main 的特殊事项
 - 5.9. 开放问题项
 - 5.10. 参考文献
- 6. 表达式
 - 6.1. 算术转换
 - 6.2. 原子
 - 6.3. 原型
 - 6.4. await 表达式
 - 6.5. 幂运算符
 - 6.6. 一元算术和位运算
 - 6.7. 二元算术运算符
 - 6.8. 移位运算
 - 6.9. 二元位运算
 - 6.10. 比较运算
 - 6.11. 布尔运算
 - 6.12. 条件表达式
 - 6.13. lambda 表达式
 - 6.14. 表达式列表
 - 6.15. 求值顺序
 - 6.16. 运算符优先级
- 7. 简单语句
 - 7.1. 表达式语句
 - 7.2. 赋值语句
 - 7.3. assert 语句
 - 7.4. pass 语句
 - 7.5. del 语句
 - 7.6. return 语句
 - 7.7. yield 语句
 - 7.8. raise 语句
 - 7.9. break 语句
 - 7.10. continue 语句
 - 7.11. import 语句
 - 7.12. global 语句
 - 7.13. nonlocal 语句
- 8. 复合语句
 - 8.1. if 语句
 - 8.2. while 语句
 - 8.3. for 语句
 - 8.4. try 语句
 - 8.5. with 语句
 - 8.6. 函数定义
 - 8.7. 类定义

- 8.8. 协程
- 9. 最高层级组件
 - 9.1. 完整的 Python 程序
 - 9.2. 文件输入
 - 9.3. 交互式输入
 - 9.4. 表达式输入
- 10. 完整的语法规范

1. 概述

本参考手册是对 Python 编程语言的描述。并不适宜作为教程使用。

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

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

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

1.1. 其他实现

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

知名的实现包括：

- CPython
- 这是最早出现并持续维护的 Python 实现，以 C 语言编写。新的语言特性通常在此率先添加。
- Jython
- 以 Java 语言编写的 Python 实现。此实现可以作为 Java 应用的一个脚本语言，或者可以用来创建需要 Java 类库支持的应用。想了解更多信息可访问 [Jython 网站](#)。
- Python for .NET
- 此实现实际上使用了 CPython 实现，但是属于 .NET 托管应用并且可以引入 .NET 类库。它的创造者是 Brian Lloyd。想了解更多详情可访问 [Python for .NET 主页](#)。
- IronPython
- 另一个 .NET 的 Python 实现，与 Python.NET 不同点在于它是生成 IL 的完全 Python 实现，并且将 Python 代码直接编译为 .NET 程序集。它的创造者就是当初创造 Jython 的 Jim Hugunin。想了解更多详情可访问 [IronPython 网站](#)。
- PyPy
- 完全使用 Python 语言编写的 Python 实现。它支持多个其他实现所没有的高级特性，例如非栈式支持和 JIT 编译器等。此项目的目标之一是通过允许方便地修改解释器（因为它用 Python 编写的），鼓励该对语言本身进行试验。想了解更多详情可访问 [PyPy 项目主页](#)。

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

1.2. 标注

句法和词法解析的描述采用经过改进的 BNF 语法标注。这包含以下定义样式：

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

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

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

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

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

2. 词法分析

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

Python 会将读取的程序文本转为 Unicode 码点；源文件的文本编码可由编码声明指定，默认为 UTF-8，详情见 [PEP 3120](#)。如果源文件无法被解码，将会引发 `SyntaxError` 。

2.1. 行结构

一个 Python 程序可分为许多 逻辑行。

2.1.1. 逻辑行

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

2.1.2. 物理行

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

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

2.1.3. 注释

一条注释以不包含在字符串面值内的井号（`#`）开头，并在物理行的末尾结束。一条注释标志着逻辑行的结束，除非存在隐含的行拼接规则。注释在语法分析中会被忽略。

2.1.4. 编码声明

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

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

这也是 GNU Emacs 认可的形式，以及

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

这是 Bram Moolenaar 的 VIM 认可的形式。

如果没有编码声明，则默认编码为 UTF-8。此外，如果文件的首字节为 UTF-8 字节顺序标志（`b'\xef\xbb\xbf'`），文件编码也声明为 UTF-8（这是 Microsoft 的 **notepad** 等软件支持的形式）。

编码声明指定的编码名称必须是 Python 所认可的编码。所有词法分析将使用此编码，包括语义字符串、注释和标识符。

2.1.5. 显式的行拼接

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

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

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

2.1.6. 隐式的行拼接

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

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

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

2.1.7. 空白行

一个只包含空格符，制表符，进纸符或者注释的逻辑行会被忽略（即不生成 `NEWLINE` 形符）。在交互模式输入语句时，对空白行的处理可能会因读取-求值-打印循环的具体实现方式而存在差异。在标准交互模式解释器中，一个完全空白的逻辑行（即连空格或注释都没有）将会结束一条多行复合语句。

2.1.8. 缩进

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

制表符会被（从左至右）替换为一至八个空格，这样缩进的空格总数为八的倍数（这是为了与 Unix 所用的规则一致）。首个非空白字符之前的空格总数将确定该行的缩进层次。一个缩进不可使用反斜杠进行多行拼接；首个反斜杠之前的空格将确定缩进层次。

在一个源文件中如果混合使用制表符和空格符缩进，并使得确定缩进层次需要依赖于制表符对应的空格数量设置，则被视为不合规则；此情况将会引发 `TabError`。

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

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

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

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

这是一个正确（但令人迷惑）的Python 代码缩进示例：

```
1. def perm(l):
2.     # Compute the list of all permutations of l
3.     if len(l) <= 1:
4.         return [l]
5.     r = []
6.     for i in range(len(l)):
7.         s = l[:i] + l[i+1:]
8.         p = perm(s)
9.         for x in p:
10.            r.append(l[i:i+1] + x)
11.     return r
```

以下示例显示了各种缩进错误：

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

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

2.1.9. 形符之间的空白

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

2.2. 其他形符

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

2.3. 标识符和关键字

标识符（或者叫做 名称）由以下词法定义进行描述。

Python 中的标识符语法是基于 Unicode 标准附件 UAX-31，并加入了下文所定义的细化与修改；更多细节还可参见 [PEP 3131](#)。

在 ASCII 范围内 (U+0001..U+007F)，可用于标识符的字符与 Python 2.x 一致：大写和小写字母 `A` 至 `z`，下划线 `_` 以及数字 `0` 至 `9`，但不可以数字打头。

Python 3.0 引入了 ASCII 范围以外的额外字符（见 [PEP 3131](#)）。这些字符的分类使用包含于 `unicodedata` 模块中的 Unicode 字符数据库版本。

标识符的长度没有限制。对大小写敏感。

```
1. identifier ::= xid_start xid_continue*
2. id_start  ::= <all characters in general categories Lu, Ll, Lt, Lm, Lo, Nl, the underscore, and characters
   with the Other_ID_Start property>
3. id_continue ::= <all characters in id_start, plus characters in the categories Mn, Mc, Nd, Pc and others
   with the Other_ID_Continue property>
4. xid_start  ::= <all characters in id_start whose NFKC normalization is in "id_start xid_continue">
5. xid_continue ::= <all characters in id_continue whose NFKC normalization is in "id_continue">
```

上文所用 Unicode 类别码的含义：

- *Lu* - 大写字母
- *Ll* - 小写字母
- *Lt* - 词首大写字母
- *Lm* - 修饰字母
- *Lo* - 其他字母
- *Nl* - 字母数字
- *Mn* - 非空白标识
- *Mc* - 含空白标识
- *Nd* - 十进制数字
- *Pc* - 连接标点
- *Other_ID_Start* - 由 [PropList.txt](#) 定义的显式字符列表，用来支持向下兼容
- *Other_ID_Continue* - 同上

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

Unicode 4.1 中的所有可用标识符字符列表参见以下非规范 HTML 文件链接 <https://www.dcl.hpi.uni-potsdam.de/home/loewis/table-3131.html>

2.3.1. 关键字

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

1. <code>False</code>	<code>await</code>	<code>else</code>	<code>import</code>	<code>pass</code>
2. <code>None</code>	<code>break</code>	<code>except</code>	<code>in</code>	<code>raise</code>
3. <code>True</code>	<code>class</code>	<code>finally</code>	<code>is</code>	<code>return</code>
4. <code>and</code>	<code>continue</code>	<code>for</code>	<code>lambda</code>	<code>try</code>
5. <code>as</code>	<code>def</code>	<code>from</code>	<code>nonlocal</code>	<code>while</code>
6. <code>assert</code>	<code>del</code>	<code>global</code>	<code>not</code>	<code>with</code>
7. <code>async</code>	<code>elif</code>	<code>if</code>	<code>or</code>	<code>yield</code>

2.3.2. 保留的标识符类

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

- `_*`
- 不会被 `from module import *` 导入。特殊标识符 `_` 在交互式解释器中被用来存放最近一次求值结果；它保存在 `builtins` 模块中。当不处于交互模式时，`_` 无特殊含义也没有预定义。参见 [import 语句](#)。

注解

`_` 作为名称常用于连接国际化文本；请参看 [gettext](#) 模块文档了解有关此约定的详情。

- `*`
- 系统定义的名称。这些名称由解释器及其实现（包括标准库）所定义。现有系统定义名称相关讨论参见 [特殊方法名称](#) 等章节。未来的 Python 版本中还将定义更多此类名称。任何 不遵循文档指定方式使用 `*` 名称的行为都可能导致无警告的出错。
- `__*`
- 类的私有名称。这种名称在类定义中使用，会以一种混合形式重写以避免在基类及派生类的 "私有" 属性之间出现名称冲突。参见 [标识符（名称）](#)。

2.4. 字面值

字面值用于表示一些内置类型的常量。

2.4.1. 字符串和字节串字面值

字符串字面值由以下词法定义进行描述：

```

1. stringliteral ::= [stringprefix](shortstring | longstring)
2. stringprefix ::= "r" | "u" | "R" | "U" | "f" | "F"
3.                  | "fr" | "Fr" | "fR" | "FR" | "rf" | "rF" | "Rf" | "RF"
4. shortstring   ::= ''' shortstringitem* ''' | ''' shortstringitem* '''
5. longstring    ::= """ longstringitem* """ | """ longstringitem* """
6. shortstringitem ::= shortstringchar | stringescapeseq
7. longstringitem  ::= longstringchar | stringescapeseq
8. shortstringchar ::= <any source character except "\" or newline or the quote>
9. longstringchar  ::= <any source character except "\">
10. stringescapeseq ::= "\" <any source character>

```

```

1. bytesliteral   ::= bytesprefix(shortbytes | longbytes)
2. bytesprefix   ::= "b" | "B" | "br" | "Br" | "bR" | "BR" | "rb" | "rB" | "Rb" | "RB"
3. shortbytes    ::= ''' shortbytesitem* ''' | ''' shortbytesitem* '''
4. longbytes     ::= """ longbytesitem* """ | """ longbytesitem* """
5. shortbytesitem ::= shortbyteschar | bytesescapeseq
6. longbytesitem  ::= longbyteschar | bytesescapeseq
7. shortbyteschar ::= <any ASCII character except "\" or newline or the quote>
8. longbyteschar  ::= <any ASCII character except "\">
9. bytesescapeseq ::= "\" <any ASCII character>

```

这些条目中未提及的一个语法限制是 `stringprefix` 或 `bytesprefix` 与字面值的剩余部分之间不允许有空白。源字符集是由编码声明定义的；如果源文件中没有编码声明则默认为 UTF-8；参见 [编码声明](#)。

自然语言描述：两种字面值都可以用成对单引号（`'`）或双引号（`"`）来标示首尾。它们也可以用成对的连续三个单引号或双引号来标示首尾（这通常被称为 三引号字符串）。反斜杠（`\`）字符被用来对特殊含义的字符进行转义，例如换行，反斜杠本身或是引号等字符。

字节串字面值总是带有前缀 `'b'` 或 `'B'`；它们生成 `bytes` 类型而非 `str` 类型的实例。它们只能包含 ASCII 字符；字节对应数值在128及以上必须以转义形式来表示。

字符串和字节串字面值都可以带有前缀 `'r'` 或 `'R'`；这种字符串被称为 原始字符串 其中的反斜杠会被当作其本身的字面字符来处理。因此在原始字符串字面值中，`'\u'` 和 `'\u'` 转义形式不会被特殊对待。由于 Python 2.x 的原始统一码字面值的特性与 Python 3.x 不一致，`'ur'` 语法已不再被支持。

3.3 新版功能：新加入了表示原始字节串的 `'rb'` 前缀，与 `'br'` 的意义相同。

3.3 新版功能：对旧式统一码字面值（`u'value'`）的支持被重新引入以简化 Python 2.x 和 3.x 代码库的同步维护。详情见 [PEP 414](#)。

包含 `'f'` 或 `'F'` 前缀的字符串字面值称为 格式化字符串字面值；参见 [格式化字符串字面值](#)。`'f'` 可与 `'r'` 连用，但不能与 `'b'` 或 `'u'` 连用，因此存在原始格式化字符串，但不存在格式化字节串字面值。

在三引号字面值中，允许存在未经转义的换行和引号（并原样保留），除非是未经转义连续三引号，这标志着字面值的结束。（"引号" 是用来标示字面值的字符，即 `'` 或 `"`。）

除非带有 `'r'` 或 `'R'` 前缀，字符串和字节串字面值中的转义序列会基于类似标准 C 中的转义规则来解读。可用的转义序列如下：

转义序列	意义	注释
<code>\newline</code>	反斜杠加换行全被忽略	
<code>\</code>	反斜杠（ <code>\</code> ）	
<code>\'</code>	单引号（ <code>'</code> ）	
<code>\"</code>	双引号（ <code>"</code> ）	
<code>\a</code>	ASCII 响铃（BEL）	
<code>\b</code>	ASCII 退格（BS）	
<code>\f</code>	ASCII 进纸（FF）	
<code>\n</code>	ASCII 换行（LF）	
<code>\r</code>	ASCII 回车（CR）	
<code>\t</code>	ASCII 水平制表（TAB）	
<code>\v</code>	ASCII 垂直制表（VT）	
<code>\ooo</code>	八进制数 <i>ooo</i> 码位的字符	(1, 3)
<code>\xhh</code>	十六进制数 <i>hh</i> 码位的字符	(2, 3)

仅在字符串字面值中可用的转义序列如下：

转义序列	意义	注释
<code>\N{name}</code>	Unicode 数据库中名称为 <i>name</i> 的字符	(4)
<code>\uxxxx</code>	16位十六进制数 <i>xxxx</i> 码位的字符	(5)
<code>\Uxxxxxxxx</code>	32位16进制数 <i>xxxxxxxx</i> 码位的字符	(6)

注释：

- 与标准 C 一致，接受最多三个八进制数码。
- 与标准 C 不同，要求必须为两个十六进制数码。
- 在字节串字面值中，十六进制数和八进制数转义码以相应数值代表每个字节。在字符串字面值中，这些转义码以相应数值代表每个 Unicode 字符。
-

在 3.3 版更改：加入了对别名 1 的支持。

- 要求必须为四个十六进制数码。

- 此方式可用来表示任意 Unicode 字符。要求必须为八个十六进制数码。

与标准 C 不同，所有无法识别的转义序列将原样保留在字符串中，也就是说，反斜杠会在结果中保留。（这种方式在调试时很有用：如果输错了一个转义序列，更容易在输出结果中识别错误。）另外要注意的一个关键点是：专用于字符串字面值中的转义序列如果在字节串字面值中出现，会被归类为无法识别的转义序列。

在 3.6 版更改：无法识别的转义序列将引发 `DeprecationWarning`。在某个未来的 Python 版本中它们将引发 `SyntaxWarning` 并最终将改为引发 `SyntaxError`。

即使在原始字面值中，引号也可以加上反斜杠转义符，但反斜杠会保留在输出结果中；例如 `r"\\"` 是一个有效的字符串字面值，包含两个字符：一个反斜杠和一个双引号；而 `r"\` 不是一个有效的字符串字面值（即便是原始字符串也不能以奇数个反斜杠结束）。特别地，一个原始字面值不能以单个反斜杠结束（因为此反斜杠会转义其后的引号字符）。还要注意一个反斜杠加一个换行在字面值中会被解释为两个字符，而 不是 一个连续行。

2.4.2. 字符串字面值拼接

多个相邻的字符串或字节串字面值（以空白符分隔），所用的引号可以彼此不同，其含义等同于全部拼接为一体。因此，`"hello" 'world'` 等同于 `"helloworld"`。此特性可以减少反斜杠的使用，以方便地将很长的字符串分成多个物理行，甚至每部分字符串还可分别加注释，例如：

```
1. re.compile("[A-Za-z_]"      # letter or underscore
2.           "[A-Za-z0-9_]*"   # letter, digit or underscore
3.           )
```

注意此特性是在句法层面定义的，但是在编译时实现。在运行时拼接字符串表达式必须使用 `'+'` 运算符。还要注意字面值拼接时每个部分可以使用不同的引号风格（甚至混合使用原始字符串和三引号字符串），格式化字符串字面值也可与普通字符串字面值拼接。

2.4.3. 格式化字符串字面值

3.6 新版功能.

格式化字符串字面值 或称 *f-string* 是带有 `'f'` 或 `'F'` 前缀的字符串字面值。这种字符串可包含替换字段，即以 `{}` 标示的表达式。而其他字符串字面值总是一个常量，格式化字符串字面值实际上是会在运行时被求值的表达式。

转义序列会像在普通字符串字面值中一样被解码（除非字面值还被标示为原始字符串）。解码之后，字符串内容所用的语法如下：

```
1. f_string      ::= (literal_char | "{" | "}" | replacement_field)*
2. replacement_field ::= "{" f_expression ["!" conversion] [":" format_spec] "}"
3. f_expression  ::= (conditional_expression | "*" or_expr)
4.                ("," conditional_expression | "," "*" or_expr)* [","]
5.                | yield_expression
6. conversion    ::= "s" | "r" | "a"
7. format_spec   ::= (literal_char | NULL | replacement_field)*
8. literal_char  ::= <any code point except "{", "}" or NULL>
```

字符串在花括号以外的部分按其字面值处理，除了双重花括号 `'{{'` 或 `'}}'` 会被替换为相应的单个花括号。单个左花括号 `'{'` 标示一个替换字段，它以一个 Python 表达式打头，表达式之后可能有一个以叹号 `!'` 标示的转换字段。之后还可能带有一个以冒号 `':'` 标示的格式说明符。替换字段以一个右花括号 `'}'` 作为结束。

格式化字符串字面值中的表达式会被当作包含在圆括号中的普通 Python 表达式一样处理，但有少数例外。空表达式不被允许，`lambda` 和赋值表达式 `:=` 必须显式地加上圆括号。替换表达式可以包含换行（例如在三重引号字符串中），但是不能包含注释。每个表达式会在格式化字符串字面值所包含的位置按照从左至右的顺序被求值。

如果指定了转换符，表达式的求值结果会先转换再格式化。转换符 `!'s'` 即对结果调用 `str()`，`!'r'` 为调用 `repr()`，而 `!'a'` 为调用 `ascii()`。

在此之后结果会使用 `format()` 协议进行格式化。格式说明符会被传入表达式或转换结果的 `format()` 方法。如果省略格式说明符则会传入一个空字符串。然后格式化结果会包含在整个字符串最终的值当中。

顶层的格式说明符可以包含有嵌套的替换字段。这些嵌套字段也可以包含有自己的转换字段和 [格式说明符](#)，但不可再包含更深层嵌套的替换字段。这里的 [格式说明符微型语言](#) 与字符串 `.format()` 方法所使用的相同。

格式化字符串字面值可以拼接，但是一个替换字段不能拆分到多个字面值。

一些格式化字符串字面值的示例：

```
1. >>> name = "Fred"
2. >>> f"He said his name is {name!r}."
3. "He said his name is 'Fred'."
4. >>> f"He said his name is {repr(name)}." # repr() is equivalent to !r
5. "He said his name is 'Fred'."
6. >>> width = 10
7. >>> precision = 4
8. >>> value = decimal.Decimal("12.34567")
9. >>> f"result: {value:{width}.{precision}}" # nested fields
10. 'result:      12.35'
11. >>> today = datetime(year=2017, month=1, day=27)
12. >>> f"{today:%B %d, %Y}" # using date format specifier
13. 'January 27, 2017'
14. >>> number = 1024
15. >>> f"{number:#0x}" # using integer format specifier
16. '0x400'
```

与正常字符串字面值采用相同语法导致的一个结果就是替换字段中的字符不能与外部的格式化字符串字面值所用的引号相冲突：

```
1. f"abc {a["x"]} def" # error: outer string literal ended prematurely
2. f"abc {a['x']} def" # workaround: use different quoting
```

格式表达式中不允许有反斜杠，这会引发错误：

```
1. f"newline: {ord('\n')}}" # raises SyntaxError
```

想包含需要用反斜杠转义的值，可以创建一个临时变量。

```
1. >>> newline = ord('\n')
2. >>> f"newline: {newline}"
3. 'newline: 10'
```

格式化字符串字面值不可用作文档字符串，即便其中没有包含表达式。

```
1. >>> def foo():
2. ...     f"Not a docstring"
3. ...
4. >>> foo.__doc__ is None
5. True
```

另请参见 [PEP 498](#) 了解加入格式化字符串字面值的提议，以及使用了相关的格式字符串机制的 `str.format()`。

2.4.4. 数字字面值

数字字面值有三种类型：整型数、浮点数和虚数。没有专门的复数字面值（复数可由一个实数加一个虚数合成）。

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

2.4.5. 整型数字面值

整型数字面值由以下词法定义进行描述：

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

整型数字面值的长度没有限制，能一直大到占满可用内存。

在确定数字大小时字面值中的下划线会被忽略。它们可用来将数码分组以提高可读性。一个下划线可放在数码之间，也可放在基数说明符例如 `0x` 之后。

注意非零的十进制数开头不允许有额外的零。这是为了避免与 Python 在版本 3.0 之前所使用的 C 风格八进制字面值相混淆。

一些整型数字面值的示例如下：

1. 7	2147483647	0o177	0b100110111
2. 3	79228162514264337593543950336	0o377	0xdeadbeef

```
3.      100_000_000_000      0b_1110_0101
```

在 3.6 版更改：允许在字面值中使用下划线进行分组。

2.4.6. 浮点数字面值

浮点数字面值由以下词法定义进行描述：

```
1. floatnumber ::= pointfloat | exponentfloat
2. pointfloat  ::= [digitpart] fraction | digitpart "."
3. exponentfloat ::= (digitpart | pointfloat) exponent
4. digitpart   ::= digit (["_"] digit)*
5. fraction    ::= "." digitpart
6. exponent    ::= ("e" | "E") ["+" | "-"] digitpart
```

注意整型数部分和指数部分在解析时总是以 10 为基数。例如，`077e010` 是合法的，且表示的数值与 `77e10` 相同。浮点数字面值允许的范围依赖于具体实现。对于整型数字面值，支持以下划线进行分组。

一些浮点数字面值的示例如下：

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

在 3.6 版更改：允许在字面值中使用下划线进行分组。

2.4.7. 虚数字面值

虚数字面值由以下词法定义进行描述：

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

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

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


2.5. 运算符

以下形符属于运算符：

1. +	-	*	**	/	//	%	@
2. <<	>>	&		^	~	:=	
3. <	>	<=	>=	==	!=		

2.6. 分隔符

以下形符在语法中归类为分隔符：

```
1. (      )      [      ]      {      }
2. ,      :      .      ;      @      =      ->
3. +=     -=     *=     /=     //=     %=     @=
4. &=     |=     ^=     >>=    <<=     **=
```

句点也可出现于浮点数和虚数字面值中。连续三个句点有表示一个省略符的特殊含义。以上列表的后半部分为增强赋值操作符，在词法中作为分隔符，但也起到运算作用。

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

```
1. '      "      #      \
```

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

```
1. $      ?      `
```

脚注

- 1
- <http://www.unicode.org/Public/11.0.0/ucd/NameAliases.txt>

3. 数据模型

- [3.1. 对象、值与类型](#)
- [3.2. 标准类型层级结构](#)
- [3.3. 特殊方法名称](#)
- [3.4. 协程](#)

3.1. 对象、值与类型

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

每个对象都有各自的编号、类型和值。一个对象被创建后，它的 编号 就绝不会改变；你可以将其理解为该对象在内存中的地址。 `' is '` 运算符可以比较两个对象的编号是否相同； `id()` 函数能返回一个代表其编号的整数。

CPython implementation detail: 在 CPython 中， `id(x)` 就是存放 `x` 的内存的地址。

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

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

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

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

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

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

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

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

3.2. 标准类型层级结构

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

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

- None
- 此类型只有一种取值。是一个具有此值的单独对象。此对象通过内置名称 `None` 访问。在许多情况下它被用来表示空值，例如未显式指明返回值的函数将返回 `None`。它的逻辑值为假。
- NotImplemented
- 此类型只有一种取值。是一个具有此值的单独对象。此对象通过内置名称 `NotImplemented` 访问。数值方法和丰富比较方法如未实现指定运算符表示的运算则应返回此值。（解释器会根据指定运算符继续尝试反向运算或其他回退操作）。它的逻辑值为真。

详情参见 [实现算数运算](#)。

- Ellipsis
- 此类型只有一种取值。是一个具有此值的单独对象。此对象通过字面值 `...` 或内置名称 `Ellipsis` 访问。它的逻辑值为真。
- `numbers.Number`
- 此类对象由数字字面值创建，并会被作为算术运算符和算术内置函数的返回结果。数字对象是不可变的；一旦创建其值就不再改变。Python 中的数字当然非常类似数学中的数字，但也受限于计算机中的数字表示方法。

Python 区分整型数、浮点型数和复数：

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

整型数可细分为两种类型：

整型（`int`）

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

1. - 布尔型 (`bool`) (<https://docs.python.org/zh-cn/3/library/functions.html#bool>)
2. -

此类对象表示逻辑值 `False` 和 `True`。代表 `False` 和 `True` 值的两个对象是唯二的布尔对象。布尔类型是

整型的子类型，两个布尔值在各种场合的行为分别类似于数值 0 和 1，例外情况只有在转换为字符串时分别返回字符串 `"False"` 或 `"True"`。

整型数表示规则的目的是在涉及负整型数的变换和掩码运算时提供最为合理的解释。

- `numbers.Real` (`float`)
- 此类对象表示机器级的双精度浮点数。其所接受的取值范围和溢出处理将受制于底层的机器架构（以及 C 或 Java 实现）。Python 不支持单精度浮点数；支持后者通常的理由是节省处理器和内存消耗，但这点节省相对于在 Python 中使用对象的开销来说太过微不足道，因此没有理由包含两种浮点数而令该语言变得复杂。
- `numbers.Complex` (`complex`)
- 此类对象以一对机器级的双精度浮点数来表示复数值。有关浮点数的附带规则对其同样有效。一个复数值 `z` 的实部和虚部可通过只读属性 `z.real` 和 `z.imag` 来获取。
- 序列
- 此类对象表示以非负整数作为索引的有限有序集。内置函数 `len()` 可返回一个序列的条目数量。当一个序列的长度为 n 时，索引集包含数字 $0, 1, \dots, n-1$ 。序列 `a` 的条目 i 可通过 `a[i]` 选择。

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

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

序列可根据其可变性来加以区分：

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

以下类型属于不可变对象：

```
1. - 字符串
2. -
```

字符串是由 Unicode 码位值组成的序列。范围在 `U+0000 - U+10FFFF` 之内的所有码位值都可在字符串中使用。Python 没有 `char` 类型；而是将字符串中的每个码位表示为一个长度为 1 的字符串对象。内置函数 `ord()` 可将一个码位由字符串形式转换成一个范围在 `0 - 10FFFF` 之内的整型数；`chr()` 可将一个范围在 `0 - 10FFFF` 之内的整型数转换为长度为 1 的对应字符串对象。`str.encode()` 可以使用指定的文本编码将 `str` 转换为 `bytes`，而 `bytes.decode()` 则可以实现反向的解码。

```
1. - 元组
2. -
```

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

1. - 字节串
2. -

字节串对象是不可变的数组。其中每个条目都是一个 8 位字节，以取值范围 $0 \leq x < 256$ 的整型数表示。字节串面值（例如 `b'abc'`）和内置的 `bytes()` 构造器可被用来创建字节串对象。字节串对象还可以通过 `decode()` 方法解码为字符串。

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

目前有两种内生可变序列类型：

1. - 列表
2. -

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

1. - 字节数组
2. -

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

扩展模块 `array` 提供了一个额外的可变序列类型示例，`collections` 模块也是如此。

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

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

目前有两种内生集合类型：

- 集合
- 此类对象表示可变集合。它们可通过内置的 `set()` 构造器创建，并且创建之后可以通过方法进行修改，例如 `add()`。
- 冻结集合
- 此类对象表示不可变集合。它们可通过内置的 `frozenset()` 构造器创建。由于 `frozenset` 对象不可变且 `hashable`，它可以被用作另一个集合的元素或是字典的键。
- 映射
- 此类对象表示由任意索引集合所索引的对象的集合。通过下标 `a[k]` 可在映射 `a` 中选择索引为 `k` 的条目；这可以在表达式中使用，也可作为赋值或 `del` 语句的目标。内置函数 `len()` 可返回一个映射中的条

目数。

目前只有一种内生映射类型：

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

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

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

- 可调用类型
- 此类型可以被应用于函数调用操作（参见 [调用](#) 小节）：
 - 用户定义函数
 - 用户定义函数对象可通过函数定义来创建（参见 [函数定义](#) 小节）。它被调用时应附带一个参数列表，其中包含的条目应与函数所定义的形参列表一致。

特殊属性：

属性

意义

`doc`

该函数的文档字符串，没有则为 `None`；不会被子类继承。

可写

`name`

该函数的名称。

可写

`qualname`

该函数的 [qualified name](#)。

3.3 新版功能.

可写

`module`

该函数所属模块的名称，没有则为 `None`。

可写

`defaults`

由具有默认值的参数的默认参数值组成的元组，如无任何参数具有默认值则为 `None`。

可写

`code`

表示编译后的函数体的代码对象。

可写

`globals`

对存放该函数中全局变量的字典的引用 -- 函数所属模块的全局命名空间。

只读

`dict`

命名空间支持的函数属性。

可写

`closure`

`None` 或包含该函数可用变量的绑定的单元的元组。有关 `cell_contents` 属性的详情见下。

只读

`annotations`

包含参数标注的字典。字典的键是参数名，如存在返回标注则为 `'return'`。

可写

`kwdefaults`

仅包含关键字参数默认值的字典。

可写

大部分标有 "Writable" 的属性均会检查赋值的类型。

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

单元对象具有 `cell_contents` 属性。这可被用来获取以及设置单元的值。

有关函数定义的额外信息可以从其代码对象中提取；参见下文对内部类型的描述。`cell` 类型可以在 `types` 模块中访问。

- 实例方法
- 实例方法用于结合类、类实例和任何可调用对象（通常为用户定义函数）。

特殊的只读属性：`self` 为类实例对象本身，`func` 为函数对象；`doc` 为方法的文档（与 `func.doc` 作用相同）；`name` 为方法名称（与 `func.name` 作用相同）；`module` 为方法所属模块的名称，没有则为

`None`。

方法还支持获取（但不能设置）下层函数对象的任意函数属性。

用户定义方法对象可在获取一个类的属性时被创建（也可能通过该类的一个实例），如果该属性为用户定义函数对象或类方法对象。

当通过从类实例获取一个用户定义函数对象的方式创建一个实例方法对象时，类实例对象的 `self` 属性即为该实例，并会绑定方法对象。该新建方法的 `func` 属性就是原来的函数对象。

当通过从类或实例获取一个类方法对象的方式创建一个实例对象时，实例对象的 `self` 属性为该类本身，其 `func` 属性为类方法对应的下层函数对象。

当一个实例方法对象被调用时，会调用对应的下层函数（`func`），并将类实例（`self`）插入参数列表的开头。例如，当 `C` 是一个包含了 `f()` 函数定义的类，而 `x` 是 `C` 的一个实例，则调用 `x.f(1)` 就等同于调用 `C.f(x, 1)`。

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

请注意从函数对象到实例方法对象的变换会在每一次从实例获取属性时发生。在某些情况下，一种高效的优化方式是将属性赋值给一个本地变量并调用该本地变量。还要注意这样的变换只发生于用户定义函数；其他可调用对象（以及所有不可调用对象）在被获取时都不会发生变换。还有一个需要关注的要点是作为一个类实例属性的用户定义函数不会被转换为绑定方法；这样的变换 仅当 函数是类属性时才会发生。

- 生成器函数
 - 一个使用 `yield` 语句（见 [yield 语句](#) 章节）的函数或方法被称作一个 生成器函数。这样的函数在被调用时，总是返回一个可以执行函数体的迭代器对象：调用该迭代器的 `iterator.next()` 方法将会导致这个函数一直运行直到它使用 `yield` 语句提供了一个值为止。当这个函数执行 `return` 语句或者执行到末尾时，将引发 `StopIteration` 异常并且这个迭代器将到达所返回的值集合的末尾。
- 协程函数
 - 使用 `async def` 来定义的函数或方法就被称为 协程函数。这样的函数在被调用时会返回一个 `coroutine` 对象。它可能包含 `await` 表达式以及 `async with` 和 `async for` 语句。详情可参见 [协程对象](#) 一节。
- 异步生成器函数
 - 使用 `async def` 来定义并包含 `yield` 语句的函数或方法就被称为 异步生成器函数。这样的函数在被调用时会返回一个异步迭代器对象，该对象可在 `async for` 语句中用来执行函数体。

调用异步迭代器的 `aiterator.anext()` 方法将会返回一个 `awaitable`，此对象会在被等待时执行直到使用 `yield` 表达式输出一个值。当函数执行时到空的 `return` 语句或是最后一条语句时，将会引发 `StopAsyncIteration` 异常，异步迭代器也会到达要输出的值集合的末尾。

- 内置函数
 - 内置函数对象是对于 `C` 函数的外部封装。内置函数的例子包括 `len()` 和 `math.sin()`（`math` 是一个标准内置模块）。内置函数参数的数量和类型由 `C` 函数决定。特殊的只读属性：`doc` 是函数的文档字符

串，如果没有则为 `None`；`name` 是函数的名称；`self` 设定为 `None`（参见下一条目）；`module` 是函数所属模块的名称，如果没有则为 `None`。

- 内置方法

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

- 类

- 类是可调用的。此种对象通常是作为“工厂”来创建自身的实例，类也可以有重载 `new()` 的变体类型。调用的参数会传给 `new()`，而且通常也会传给 `init()` 来初始化新的实例。

- 类实例

- 任意类的实例通过在所属类中定义 `call()` 方法即能成为可调用的对象。

- 模块

- 模块是 Python 代码的基本组织单元，由 [导入系统](#) 创建，由 `import` 语句发起调用，或者通过 `importlib.import_module()` 和内置的 `import()` 等函数发起调用。模块对象具有由字典对象实现的命名空间（这是被模块中定义的函数的 `globals` 属性引用的字典）。属性引用被转换为该字典中的查找，例如 `m.x` 相当于 `m.dict["x"]`。模块对象不包含用于初始化模块的代码对象（因为初始化完成后不需要它）。

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

预定义的（可写）属性：`name` 为模块的名称；`doc` 为模块的文档字符串，如果没有则为 `None`；`annotations`（可选）为一个包含 [变量标注](#) 的字典，它是在模块体执行时获取的；`file` 是模块对应的被加载文件的路径名，如果它是加载自一个文件的话。某些类型的模块可能没有 `file` 属性，例如 C 模块是静态链接到解释器内部的；对于从一个共享库动态加载的扩展模块来说该属性为该共享库文件的路径名。

特殊的只读属性：`dict` 为以字典对象表示的模块命名空间。

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

- 自定义类
- 自定义类这种类型一般通过类定义来创建（参见 [类定义](#) 一节）。每个类都有通过一个字典对象实现的独立命名空间。类属性引用会被转化为在此字典中查找，例如 `C.x` 会被转化为 `C.dict["x"]`（不过也存在一些钩子对象以允许其他定位属性的方式）。当未在其中发现某个属性名称时，会继续在基类中查找。这种基类查找使用 C3 方法解析顺序，即使存在‘钻石形’继承结构即有多条继承路径连到一个共同祖先也能保持正确的行为。有关 Python 使用的 C3 MRO 的详情可查看配合 2.3 版发布的文档 <https://www.python.org/download/releases/2.3/mro/>。

当一个类属性引用（假设类名为 `C`）会产生一个类方法对象时，它将转化为一个 `self` 属性为 `C` 的实例方法对象。当其会产生一个静态方法对象时，它将转化为该静态方法对象所封装的对象。从类的 `dict` 所包含内容以外获取属性的其他方式请参看 [实现描述器](#) 一节。

类属性赋值会更新类的字典，但不会更新基类的字典。

类对象可被调用（见上文）以产生一个类实例（见下文）。

特殊属性：`name` 为类的名称；`module` 为类所在模块的名称；`dict` 为包含类命名空间的字典；`bases` 为包含基类的元组，按其在基类列表中的出现顺序排列；`doc` 为类的文档字符串，如果没有则为 `None`；`annotations`（可选）为一个包含 [变量标注](#) 的字典，它是在类体执行时获取的。

- 类实例
- 类实例可通过调用类对象来创建（见上文）。每个类实例都有通过一个字典对象实现的独立命名空间，属性引用会首先在此字典中查找。当未在其中发现某个属性，而实例对应的类中有该属性时，会继续在类属性中查找。如果找到的类属性为一个用户定义函数对象，它会被转化为实例方法对象，其 `self` 属性即该实例。静态方法和类方法对象也会被转化；参见上文 "Classes" 一节。要了解其他通过类实例来获取相应类属性的方式可参见 [实现描述器](#) 一节，这样得到的属性可能与实际存放于类的 `dict` 中的对象不同。如果未找到类属性，而对象对应的类具有 `getattr()` 方法，则会调用该方法来满足查找要求。

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

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

特殊属性：`dict` 为属性字典；`class` 为实例对应的类。

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

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

- 内部类型
- 某些由解释器内部使用的类型也被暴露给用户。它们的定义可能随未来解释器版本的更新而变化，为内容完整起见在此处一并介绍。
 - 代码对象
 - 代码对象表示 编译为字节的 可执行 Python 代码，或称 `bytecode`。代码对象和函数对象的区别在于函数对象包含对函数全局对象（函数所属的模块）的显式引用，而代码对象不包含上下文；而且默认参数值会存放于函数对象而不是代码对象内（因为它们表示在运行时算出的值）。与函数对象不同，代码对象不可变，也不包含对可变对象的引用（不论是直接还是间接）。

特殊的只读属性：`co_name` 给出了函数名称；`co_argcount` 为位置参数的总数量（包括有有限位置参数和带有默认值的参数）；`co_posonlyargcount` 为仅限位置参数的数量（包括带有默认值的参数）；`co_kwonlyargcount` 为仅限关键字参数的数量（包括带有默认值的参数）；`co_nlocals` 为函数使用的局部变量的数量（包括参数）；`co_varnames` 为一个包含局部变量名称的元组（参数名称排在最前面）；`co_cellvars` 为一个包含被嵌套函数所引用的局部变量名称的元组；`co_freevars` 为一个包含自由变量名称的元组；`co_code` 为一个表示字节码指令序列的字符串；`co_consts` 为一个包含字节码所使用的字面值的元组；`co_names` 为一个包含字节码所使用的名称的元组；`co_filename` 为被编码代码所在的文件名；`co_firstlineno` 为函数首行的行号；`co_lnotab` 为一个字符串，其中编码了从字节码偏移量到行号的映射（详情参见解释器的源代码）；`co_stacksize` 为要求的栈大小（包括局部变量）；`co_flags` 为一个整数，其中编码了解释器所用的多个旗标。

以下是可用于 `co_flags` 的标志位定义：如果函数使用 `arguments` 语法来接受任意数量的位置参数，则

`0x04` 位被设置；如果函数使用 `*` `keywords` 语法来接受任意数量的关键字参数，则 `0x08` 位被设置；如果函数是一个生成器，则 `0x20` 位被设置。

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

`co_flags` 中的其他位被保留为内部使用。

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

- 帧对象
- 帧对象表示执行帧。它们可能出现在回溯对象中（见下文），还会被传递给注册跟踪函数。

特殊的只读属性：`f_back` 为前一堆栈帧（指向调用者），如是最底层堆栈帧则为 `None`；`f_code` 为此帧中所执行的代码对象；`f_locals` 为用于查找本地变量的字典；`f_globals` 则用于查找全局变量；`f_builtins` 用于查找内置（固有）名称；`f_lasti` 给出精确指令（这是代码对象的字节码字符串的一个索引）。

特殊的可写属性：`f_trace`，如果不为 `None`，则是在代码执行期间调用各类事件的函数（由调试器使用）。通常每个新源码行会触发一个事件 - 这可以通过将 `f_trace_lines` 设为 `False` 来禁用。

具体的实现可能会通过将 `f_trace_opcodes` 设为 `True` 来允许按操作码请求事件。请注意如果跟踪函数引发的异常逃逸到被跟踪的函数中，这可能会导致未定义的解释器行为。

`f_lineno` 为帧的当前行号 -- 在这里写入从一个跟踪函数内部跳转的指定行（仅用于最底层的帧）。调试器可以通过写入 `f_lineno` 实现一个 Jump 命令（即设置下一语句）。

帧对象支持一个方法：

```
1. - <code>frame.</code><code>clear</code>()[(https://docs.python.org/zh-cn/3/reference/#frame.clear)
2. -
```

此方法清除该帧持有的全部对本地变量的引用。而且如果该帧属于一个生成器，生成器会被完成。这有助于打破包含帧对象的循环引用（例如当捕获一个异常并保存其回溯在之后使用）。

如果该帧当前正在执行则会引发 `RuntimeError`。

3.4 新版功能.

- 回溯对象
- 回溯对象表示一个异常的栈跟踪记录。当异常发生时隐式地创建一个回溯对象，也可能通过调用 `types.TracebackType` 显式地创建。

对于隐式地创建的回溯对象，当查找异常句柄使得执行栈展开时，会在每个展开层级的当前回溯之前插入一个回溯对象。当进入一个异常句柄时，栈跟踪将对程序启用。（参见 `try 语句` 一节。）它可作为 `sys.excinfo()` 所返回的元组的第三项，以及所捕获异常的 `_traceback` 属性被获取。

当程序不包含可用的句柄时，栈跟踪会（以良好的格式）写入标准错误流；如果解释器处于交互模式，它也可作为 `sys.last_traceback` 对用户启用。

对于显式创建的回溯对象，则由回溯对象的创建者来决定应该如何链接 `tb_next` 属性来构成完整的栈跟踪。

特殊的只读属性：`tb_frame` 指向当前层级的执行帧；`tb_lineno` 给出发生异常所在的行号；`tb_lasti` 标示具体指令。如果异常发生于没有匹配的 `except` 子句或有 `finally` 子句的 `try` 语句中，回溯对象中的行号和最后指令可能与相应帧对象中行号不同。

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

在 3.7 版更改：回溯对象现在可以使用 Python 代码显式地实例化，现有实例的 `tb_next` 属性可以被更新。

- 切片对象
- 切片对象用来表示 `getitem()` 方法得到的切片。该对象也可使用内置的 `slice()` 函数来创建。

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

切片对象支持一个方法：

```
- <code>slice.</code><code>indices</code>(_self_, _length_)[](https://docs.python.org/zh-cn/3/reference/#slice.indices)
1. cn/3/reference/#slice.indices)
2. -
```

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

- 静态方法对象
- 静态方法对象提供了一种避免上文所述将函数对象转换为方法对象的方式。静态方法对象为对任意其他对象的封装，通常用来封装用户定义方法对象。当从类或类实例获取一个静态方法对象时，实际返回的对象是封装的对象，它不会被进一步转换。静态方法对象自身不是可调用的，但它们所封装的对象通常都是可调用的。静态方法对象可通过内置的 `staticmethod()` 构造器来创建。
- 类方法对象
- 类方法对象和静态方法一样是对其他对象的封装，会改变从类或类实例获取该对象的方式。类方法对象在此类获取操作中的行为已在上文 "用户定义方法" 一节中描述。类方法对象可通过内置的 `classmethod()` 构造器来创建。

3.3. 特殊方法名称

一个类可以通过定义具有特殊名称的方法来实现由特殊语法所引发的特定操作（例如算术运算或下标与切片）。这是 Python 实现 操作符重载 的方式，允许每个类自行定义基于操作符的特定行为。例如，如果一个类定义了名为 `getitem()` 的方法，并且 `x` 为该类的一个实例，则 `x[i]` 基本就等同于 `type(x).getitem(x, i)`。除非有说明例外情况，在没有定义适当方法的情况下尝试执行一种操作将引发一个异常（通常为 `AttributeError` 或 `TypeError`）。

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

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

3.3.1. 基本定制

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

典型的实现会附带适宜的参数使用 `super().new(cls[, ...])`，通过超类的 `new()` 方法来创建一个类的新实例，然后根据需要修改新创建的实例再将其返回。

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

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

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

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

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

- `object.del(self)`
- 在实例将被销毁时调用。这还会调用终结器或析构器（不适当）。如果一个基类具有 `del()` 方法，则其所派生的类如果也有 `del()` 方法，就必须显式地调用它以确保实例基类部分的正确清除。

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

当解释器退出时不会确保为仍然存在的对象调用 `del()` 方法。

注解

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

CPython implementation detail: It is possible for a reference cycle to prevent the reference count of an object from going to zero. In this case, the cycle will be later detected and deleted by the [cyclic garbage collector](#). A common cause of reference cycles is when an exception has been caught in a local variable. The frame's locals then reference the exception, which references its own traceback, which references the locals of all frames caught in the traceback.

参见

`gc` 模块的文档。

警告

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

- `del()` 可在任意代码被执行时启用，包括来自任意线程的代码。如果 `del()` 需要接受锁或启用其他阻塞资源，可能会发生死锁，例如该资源已被为执行 `del()` 而中断的代码所获取。
- `del()` 可以在解释器关闭阶段被执行。因此，它需要访问的全局变量（包含其他模块）可能已被删除或设为 `None`。Python 会保证先删除模块中名称以单个下划线打头的全局变量再删除其他全局变量；如果已不存在其他对此类全局变量的引用，这有助于确保导入的模块在 `del()` 方法被调用时仍然可用。
- `object.repr(self)`
- 由 `repr()` 内置函数调用以输出一个对象的“官方”字符串表示。如果可能，这应类似一个有效的 Python 表达式，能被用来重建具有相同取值的对象（只要有适当的环境）。如果这不可能，则应返回形式如 `<...some useful description...>` 的字符串。返回值必须是一个字符串对象。如果一个类定义了 `repr()` 但未定义 `str()`，则在需要该类的实例的“非正式”字符串表示时也会使用 `repr()`。

此方法通常被用于调试，因此确保其表示的内容包含丰富信息且无歧义是很重要的。

- `object.str(self)`
- 通过 `str(object)` 以及内置函数 `format()` 和 `print()` 调用以生成一个对象的“非正式”或格式良好的字符串表示。返回值必须为一个 [字符串](#) 对象。

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

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

- `object.bytes(self)`
- 通过 `bytes` 调用以生成一个对象的字节串表示。这应该返回一个 `bytes` 对象。
- `object.format(self, format_spec)`
- 通过 `format()` 内置函数、扩展、[格式化字符串面值](#) 的求值以及 `str.format()` 方法调用以生成一个对象的“格式化”字符串表示。`format_spec` 参数为包含所需格式选项描述的字符串。`format_spec` 参数的解读是由实现 `format()` 的类型决定的，不过大多数类或是将格式化委托给某个内置类型，或是使用相似的格式化选项语法。

请参看 [格式规格迷你语言](#) 了解标准格式化语法的描述。

返回值必须为一个字符串对象。

在 3.4 版更改：`object` 本身的 `format` 方法如果被传入任何非空字符，将会引发一个 `TypeError`。

在 3.7 版更改：`object.format(x, '')` 现在等同于 `str(x)` 而不再是 `format(str(self), '')`。

- `object.it(self, other)`
- `object.le(self, other)`

- `object.eq(self, other)`
- `object.ne(self, other)`
- `object.gt(self, other)`
- `object.ge(self, other)`
- 以上这些被称为“富比较”方法。运算符与方法名称的对应关系如下：`x<y` 调用 `x.lt(y)`、`x<=y` 调用 `x.le(y)`、`x==y` 调用 `x.eq(y)`、`x!=y` 调用 `x.ne(y)`、`x>y` 调用 `x.gt(y)`、`x>=y` 调用 `x.ge(y)`。

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

在默认情况下 `ne()` 会委托给 `eq()` 并将结果取反，除非结果为 `NotImplemented`。比较运算符之间没有其他隐含关系，例如 `(x<y or x==y)` 为真并不意味着 `x<=y`。要根据单根运算自动生成排序操作，请参看 `functools.total_ordering()`。

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

这些方法并没有对调参数版本（在左边参数不支持该操作但右边参数支持时使用）；而是 `lt()` 和 `gt()` 互为对方的反射，`le()` 和 `ge()` 互为对方的反射，而 `eq()` 和 `ne()` 则是它们自己的反射。如果两个操作数的类型不同，且右操作数类型是左操作数类型的直接或间接子类，则优先选择右操作数的反射方法，否则优先选择左操作数的方法。虚拟子类不会被考虑。

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

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

注解

`hash()` 会从一个对象自定义的 `hash()` 方法返回值中截断为 `Pyssizet` 的大小。通常对 64 位构建为 8 字节，对 32 位构建为 4 字节。如果一个对象的 `[hash()]` (https://docs.python.org/zh-cn/3/reference/#object.__hash) 必须在不同位大小的构建上进行互操作，请确保检查全部所支持构建的宽度。做到这一点的简单方法是使用 `python -c "import sys; print(sys.hash_info.width)"`。

如果一个类没有定义 `eq()` 方法，那么也不应该定义 `hash()` 操作；如果它定义了 `eq()` 但没有定义 `hash()`，则其实例将不可被用作可哈希集的项。如果一个类定义了可变对象并实现了 `eq()` 方法，则不应该实现 `hash()`，因为可哈希集的实现要求键的哈希集是不可变的（如果对象的哈希值发生改变，它将处于错误的哈希桶中）。

用户定义类默认带有 `eq()` 和 `hash()` 方法；使用它们与任何对象（自己除外）比较必定不相等，并且 `x.hash()` 会返回一个恰当的值以确保 `x == y` 同时意味着 `x is y` 且 `hash(x) == hash(y)`。

一个类如果重载了 `eq()` 且没有定义 `hash()` 则会将其 `hash()` 隐式地设为 `None`。当一个类的 `hash()` 方法为 `None` 时，该类的实例将在一个程序尝试获取其哈希值时正确地引发 `TypeError`，并会在检测

`isinstance(obj, collections.abc.Hashable)` 时被正确地识别为不可哈希对象。

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

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

注解

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

这种做法是为了防止以下形式的拒绝服务攻击：通过仔细选择输入来利用字典插入操作在最坏情况下的执行效率即 $O(n^2)$ 复杂度。详情见 <http://www.ocert.org/advisories/ocert-2011-003.html>

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

另见 `PYTHONHASHSEED`。

在 3.3 版更改：默认启用哈希随机化。

- `object.bool(self)`
- 调用此方法以实现真值检测以及内置的 `bool()` 操作；应该返回 `False` 或 `True`。如果未定义此方法，则会查找并调用 `len()` 并在其返回非零值时视对象的逻辑值为真。如果一个类既未定义 `len()` 也未定义 `bool()` 则视其所有实例的逻辑值为真。

3.3.2. 自定义属性访问

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

- `object.getattr(self, name)`
- 当默认属性访问因引发 `AttributeError` 而失败时被调用（可能是调用 `getattrattribute()` 时由于 `name` 不是一个实例属性或 `self` 的类关系树中的属性而引发了 `AttributeError`；或者是对 `name` 特性属性调用 `get()` 时引发了 `AttributeError`）。此方法应当返回（找到的）属性值或是引发一个 `AttributeError` 异常。

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

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

注解

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

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

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

- `object.delattr(self, name)`
- 类似于 `setattr()` 但其作用为删除而非赋值。此方法应该仅在 `del obj.name` 对于该对象有意义时才被实现。
- `object.dir(self)`
- 此方法会在对相应对象调用 `dir()` 时被调用。返回值必须为一个序列。`dir()` 会把返回的序列转换为列表并对其进行排序。

3.3.2.1. 自定义模块属性访问

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

`dir` 函数应当不接受任何参数，并且返回一个表示模块中可访问名称的字符串序列。此函数如果存在，将会重载一个模块中的标准 `dir()` 查找。

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

```
1. import sys
2. from types import ModuleType
3.
4. class VerboseModule(ModuleType):
5.     def __repr__(self):
6.         return f'Verbose {self.__name__}'
7.
8.     def __setattr__(self, attr, value):
9.         print(f'Setting {attr}...')
10.        super().__setattr__(attr, value)
11.
12. sys.modules[__name__].__class__ = VerboseModule
```

注解

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

在 3.5 版更改：`class` 模块属性改为可写。

3.7 新版功能：`getattr` 和 `dir` 模块属性。

参见

- [PEP 562](#) - 模块 `getattr` 和 `dir`
- 描述用于模块的 `getattr` 和 `dir` 函数。

3.3.2.2. 实现描述器

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

- `object.get(self, instance, owner=None)`
- 调用此方法以获取所有者类的属性（类属性访问）或该类的实例的属性（实例属性访问）。可选的 `owner` 参数是所有者类而 `instance` 是被用来访问属性的实例，如果通过 `owner` 来访问属性则返回 `None`。

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

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

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

请注意，添加 `set()` 或 `delete()` 会将描述器变成“数据描述器”。更多细节请参阅 [发起调用描述器](#)。

- `object.delete(self, instance)`
- 调用此方法以删除 `instance` 指定的所有者类的实例的属性。
- `object.set_name(self, owner, name)`
- 在所有者类 `owner` 创建时被调用。描述器会被赋值给 `name`。

3.6 新版功能.

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

3.3.2.3. 发起调用描述器

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

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

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

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

- 直接调用
- 最简单但最不常见的调用方式是用户代码直接发起调用一个描述器方法：`x.get(a)`。
- 实例绑定
- 如果绑定到一个对象实例，`a.x` 会被转换为调用：`type(a).dict['x'].get(a, type(a))`。
- 类绑定
- 如果绑定到一个类，`A.x` 会被转换为调用：`A.dict['x'].get(None, A)`。
- 超绑定
- 如果 `a` 是 `super` 的一个实例，则绑定 `super(B, obj).m()` 会在 `obj.class.mro` 中搜索 `B` 的直接上级基类 `A` 然后通过以下调用发起调用描述器：`A.dict['m'].get(obj, obj.class)`。

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

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

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

3.3.2.4. slots

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

相比使用 **dict** 此方式可以显著地节省空间。属性查找速度也可得到显著的提升。

- `object.slots`
- 这个类变量可赋值为字符串、可迭代对象或由实例使用的变量名构成的字符串序列。**slots** 会为已声明的变量保留空间，并阻止自动为每个实例创建 **dict** 和 **weakref**。

3.3.2.4.1. 使用 slots 的注意事项

- 当继承自一个未定义 **slots** 的类时，实例的 **dict** 和 **weakref** 属性将总是可访问。
- 没有 **dict** 变量，实例就不能给未在 **slots** 定义中列出的新变量赋值。尝试给一个未列出的变量名赋值将引发 `AttributeError`。新变量需要动态赋值，就要将 `'dict'` 加入到 **slots** 声明的字符串序列中。
- 如果未给每个实例设置 **weakref** 变量，定义了 **slots** 的类就不支持对其实际的弱引用。如果需要弱引用支持，就要将 `'weakref'` 加入到 **slots** 声明的字符串序列中。
- **slots** 是通过为每个变量名创建描述器（[实现描述器](#)）在类层级上实现的。因此，类属性不能被用来为通过 **slots** 定义的实例变量设置默认值；否则，类属性就会覆盖描述器赋值。
- **slots** 声明的作用不只限于定义它的类。在父类中声明的 **slots** 在其子类中同样可用。不过，子类将会获得 **dict** 和 **weakref** 除非它们也定义了 **slots**（其中应该仅包含对任何 额外 名称的声明位置）。
- 如果一个类定义的位置在某个基类中也有定义，则由基类位置定义的实例变量将不可访问（除非通过直接从基类获取其描述器的方式）。这会使得程序的含义变成未定义。未来可能会添加一个防止此情况的检查。
- 非空的 **slots** 不适用于派生自“可变长度”内置类型例如 `int`、`bytes` 和 `tuple` 的派生类。
- 任何非字符串可迭代对象都可以被赋值给 **slots**。映射也可以被使用；不过，未来可能会分别赋给每个键具有特殊含义的值。
- **class** 赋值仅在两个类具有相同的 **slots** 时才会起作用。
- 带有多个父类声明位置的多重继承也是可用的，但仅允许一个父类具有由声明位置创建的属性（其他基类必须具有空的位置布局）——违反规则将引发 `TypeError`。
- 如果为 **slots** 使用了一个迭代器，则会为迭代器的每个值创建描述器。但是 **slots** 属性将为一个空迭代器。

3.3.3. 自定义类创建

当一个类继承其他类时，那个类的 `__init_subclass__` 会被调用。这样就可以编写能够改变子类行为的类。这与类装饰器有紧密的关联，但是类装饰器是影响它们所应用的特定类，而 `__init_subclass__` 则只作用于定义了该方法的类所派生的子类。

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

传入一个新类的关键字参数会被传给父类的 `__init_subclass__`。为了与其他使用 `__init_subclass__` 的类兼容，应当根据需要去掉部分关键字参数再将其余的传给基类，例如：

```
1. class Philosopher:
2.     def __init_subclass__(cls, /, default_name, **kwargs):
3.         super().__init_subclass__(**kwargs)
4.         cls.default_name = default_name
5.
6. class AustralianPhilosopher(Philosopher, default_name="Bruce"):
7.     pass
```

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

注解

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

3.6 新版功能.

3.3.3.1. 元类

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

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

```
1. class Meta(type):
2.     pass
3.
4. class MyClass(metaclass=Meta):
5.     pass
6.
7. class MySubclass(MyClass):
8.     pass
```

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

当一个类定义被执行时，将发生以下步骤：

- 解析 MRO 条目；
- 确定适当的元类；
- 准备类命名空间；
- 执行类主体；
- 创建类对象。

3.3.3.2. 解析 MRO 条目

如果在类定义中出现的基类不是 `type` 的实例，则使用 `mro_entries` 方法对其进行搜索，当找到结果时，它会以原始基类元组做参数进行调用。此方法必须返回类的元组以替代此基类被使用。元组可以为空，在此情况下原始基类将被忽略。

参见

[PEP 560](#) - 对 `typing` 模块和泛型类型的核心支持

3.3.3.3. 确定适当的元类

为一个类定义确定适当的元类是根据以下规则：

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

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

`TypeError` 。

3.3.3.4. 准备类命名空间

一旦适当的元类被确定，则类命名空间将会准备好。如果元类具有 `prepare` 属性，它会以 `namespace = metaclass.prepare(name, bases, **kws)` 的形式被调用（其中如果有附加的关键字参数，应来自类定义）。

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

参见

- [PEP 3115](#) - Python 3000 中的元类
- 引入 `prepare` 命名空间钩子

3.3.3.5. 执行类主体

类主体会以（类似于） `exec(body, globals(), namespace)` 的形式被执行。普通调用与 `exec()` 的关键区别在于

当类定义发生于函数内部时，词法作用域允许类主体（包括任何方法）引用来自当前和外部作用域的名称。

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

3.3.3.6. 创建类对象

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

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

CPython implementation detail: 在 CPython 3.6 及之后的版本中，`class` 单元会作为类命名空间中的 cell is passed to the metaclass as a `classcell` 条目被传给元类。如果存在，它必须被向上传播给 `type.new` 调用，以便能正确地初始化该类。如果不这样做，在 Python 3.8 中将引发 `RuntimeError`。

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

- 首先，`type.new` 将收集类命名空间中所有定义了 `set_name()` 方法的描述器；
- 接下来，所有这些 `set_name` 方法将使用所定义的类和特定描述器所赋的名称进行调用；
- 最后，将在新类根据方法解析顺序所确定的直接父类上调用 `init_subclass()` 钩子。

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

当通过 `type.new` 创建一个新类时，提供以作为命名空间形参的对象会被复制到一个新的有序映射并丢弃原对象。这个新副本包装于一个只读代理中，后者则成为类对象的 `dict` 属性。

参见

- [PEP 3135](#) - 新的超类型
- 描述隐式的 `class` 闭包引用

3.3.3.7. 元类的作用

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

3.3.4. 自定义实例及子类检查

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

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

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

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

参见

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

3.3.5. 模拟泛型类型

通过定义一个特殊方法，可以实现由 [PEP 484](#) 所规定的泛型类语法（例如 `List[int]`）：

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

此方法的查找会基于对象自身，并且当定义于类体内部时，此方法将隐式地成为类方法。请注意，此机制主要是被保留用于静态类型提示，不鼓励在其他场合使用。

参见

[PEP 560](#) - 对 `typing` 模块和泛型类型的核心支持

3.3.6. 模拟可调用对象

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

3.3.7. 模拟容器类型

可以定义下列方法来实现容器对象。容器通常属于序列（如列表或元组）或映射（如字典），但也存在其他形式的容器。前几个方法集被用于模拟序列或是模拟映射；两者的不同之处在于序列允许的键应为整数 k 且 $0 \leq k < N$ 其中 N 是序列或定义指定区间的项的切片对象的长度。此外还建议让映射提供 `keys()` , `values()` , `items()` , `get()` , `clear()` , `setdefault()` , `pop()` , `popitem()` , `copy()` 以及 `update()` 等方法，它们的行为应与 Python 标准字典对象的相应方法类似。此外 `collections.abc` 模块提供了一个 `MutableMapping` 抽象基类以便根据由 `getitem()` , `setitem()` , `delitem()` , 和 `keys()` 组成的基本集来创建所需的方法。可变序列还应像 Python 标准列表对象那样提供 `append()` , `count()` , `index()` , `extend()` , `insert()` , `pop()` , `remove()` , `reverse()` 和 `sort()` 等方法。最后，序列类型还应通过定义下文描述的 `add()` , `radd()` , `iadd()` , `mul()` , `rmul()` 和 `imul()` 等方法来实现加法（指拼接）和乘法（指重复）；它们不应定义其他数值运算符。此外还建议映射和序列都实现 `contains()` 方法以允许高效地使用 `in` 运算符；对于映射，`in` 应该搜索映射的键；对于序列，则应搜索其中的值。另外还建议映射和序列都实现 `iter()` 方法以允许高效地迭代容器中的条目；对于映射，`iter()` 应当迭代对象的键；对于序列，则应当迭代其中的值。

- `object. len (self)`
- 调用此方法以实现内置函数 `len()` 。应该返回对象的长度，以一个 ≥ 0 的整数表示。此外，如果一个对象未定义 `bool()` 方法而其 `len()` 方法返回值为零，则在布尔运算中会被视为假值。

CPython implementation detail: 在 CPython 中，要求长度最大为 `sys.maxsize` 。如果长度大于 `sys.maxsize` 则某些特性（例如 `len()` ）可能会引发 `OverflowError` 。要通过真值检测来防止引发 `OverflowError` ，对象必须定义 `bool()` 方法。

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

3.4 新版功能.

注解

切片是通过下述三个专门方法完成的。以下形式的调用

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

会为转写为

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

其他形式以此类推。略去的切片项总是以 `None` 补全。

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

`IndexError` 异常。对于映射类型，如果 `key` 找不到（不在容器中）则应引发 `KeyError` 异常。

注解

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

- `object.setitem(self, key, value)`
- 调用此方法以实现向 `self[key]` 赋值。注意事项与 `getitem()` 相同。为对象实现此方法应该仅限于需要映射允许基于键修改值或添加键，或是序列允许元素被替换时。不正确的 `key` 值所引发的异常应与 `getitem()` 方法的情况相同。
- `object.delitem(self, key)`
- 调用此方法以实现 `self[key]` 的删除。注意事项与 `getitem()` 相同。为对象实现此方法应该仅限于需要映射允许移除键，或是序列允许移除元素时。不正确的 `key` 值所引发的异常应与 `getitem()` 方法的情况相同。
- `object.missing(self, key)`
- 此方法由 `dict.getitem()` 在找不到字典中的键时调用以实现 `dict` 子类的 `self[key]`。
- `object.iter(self)`
- 此方法在需要为容器创建迭代器时被调用。此方法应该返回一个新的迭代器对象，它能够逐个迭代容器中的所有对象。对于映射，它应该逐个迭代容器中的键。

迭代器对象也需要实现此方法；它们需要返回对象自身。有关迭代器对象的详情请参看 [迭代器类型](#) 一节。

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

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

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

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

对于未定义 `contains()` 的对象，成员检测将首先尝试通过 `iter()` 进行迭代，然后再使用 `getitem()` 的旧式序列迭代协议，参看 [语言参考中的相应部分](#)。

3.3.8. 模拟数字类型

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

- `object.add(self, other)`
- `object.sub(self, other)`
- `object.mul(self, other)`
- `object.matmul(self, other)`
- `object.truediv(self, other)`
- `object.floordiv(self, other)`
- `object.mod(self, other)`
- `object.divmod(self, other)`
- `object.pow(self, other[, modulo])`
- `object.lshift(self, other)`
- `object.rshift(self, other)`
- `object.and(self, other)`
- `object.xor(self, other)`
- `object.or(self, other)`
- 调用这些方法来实现二进制算术运算（`+`，`-`，`*`，`@`，`/`，`//`，`%`，`divmod()`，`pow()`，`*`，`<<`，`>>`，`&`，`^`，`|`）。例如，求表达式 `x + y` 的值，其中 `x` 是具有 `add()` 方法的类的一个实例，则会调用 `x.add(y)`。`divmod()` 方法应该等价于使用 `floordiv()` 和 `mod()`，它不应该被关联到 `truediv()`。请注意如果要支持三元版本的内置 `pow()` 函数，则 `pow()` 的定义应该接受可选的第三个参数。

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

- `object.radd(self, other)`
- `object.rsub(self, other)`
- `object.rmul(self, other)`
- `object.rmatmul(self, other)`
- `object.rtruediv(self, other)`
- `object.rfloordiv(self, other)`
- `object.rmod(self, other)`
- `object.rdivmod(self, other)`
- `object.rpow(self, other)`
- `object.rlshift(self, other)`
- `object.rrshift(self, other)`
- `object.rand(self, other)`
- `object.rxor(self, other)`
- `object.ror(self, other)`
- 调用这些方法来实现具有反射（交换）操作数的二进制算术运算（`+`，`-`，`*`，`@`，`/`，`//`，`%`，`divmod()`，`pow()`，`*`，`<<`，`>>`，`&`，`^`，`|`）。这些成员函数仅会在左操作数不支持相应运算³且两个操作数类型不同时被调用。⁴例如，求表达式 `x - y` 的值，其中 `y` 是具有 `rsub()` 方法的类的一个实例，则当 `x.sub(y)` 返回 `NotImplemented` 时会调用 `y.rsub(x)`。

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

注解

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

- `object.iadd(self, other)`
- `object.isub(self, other)`
- `object.imul(self, other)`
- `object.imatmul(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)`
- 调用这些方法来实现扩展算术赋值 (`+=`, `-=`, `=`, `@=`, `/=`, `//=`, `%=`, `*=`, `<<=`, `>>=`, `&=`, `^=`, `|=`)。这些方法应该尝试进行自身操作 (修改 `self`) 并返回结果 (结果应该但并非必须为 `self`)。如果某个方法未被定义，相应的扩展算术赋值将回退到普通方法。例如，如果 `x` 是具有 `iadd()` 方法的类的一个实例，则 `x += y` 就等价于 `x = x.iadd(y)`。否则就如 `x + y` 的求值一样选择 `x.add(y)` 和 `y.radd(x)`。在某些情况下，扩展赋值可导致未预期的错误 (参见 [为什么 `a_tuple\[i\] += \['item'\]` 会在执行加法时引发异常?](#))，但此行为实际上是数据模型的一个组成部分。
- `object.neg(self)`
- `object.pos(self)`
- `object.abs(self)`
- `object.invert(self)`
- 调用此方法以实现一元算术运算 (`-`, `+`, `abs()` 和 `~`)。
- `object.complex(self)`
- `object.int(self)`
- `object.float(self)`
- 调用这些方法以实现内置函数 `complex()`, `int()` 和 `float()`。应当返回一个相应类型的值。
- `object.index(self)`
- 调用此方法以实现 `operator.index()` 以及 Python 需要无损地将数字对象转换为整数对象的场合 (例如切片或是内置的 `bin()`, `hex()` 和 `oct()` 函数)。存在此方法表明数字对象属于整数类型。必须返回一个整数。

如果未定义 `int()`, `float()` 和 `complex()` 则相应的内置函数 `int()`, `float()` 和 `complex()` 将

回退为 `index()` 。

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

如果未定义 `int()` 则内置函数 `int()` 会回退到 `trunc()` 。

3.3.9. with 语句上下文管理器

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

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

要了解上下文管理器的更多信息，请参阅 [上下文管理器类型](#)。

- `object. enter (self)`
 - 进入与此对象相关的运行时上下文。 `with` 语句将会绑定这个方法的返回值到 `as` 子句中指定的目标，如果有的话。
- `object. exit (self, exc_type, exc_value, traceback)`
 - 退出关联到此对象的运行时上下文。 各个参数描述了导致上下文退出的异常。 如果上下文是无异常地退出的，三个参数都将为 `None`。

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

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

参见

- [PEP 343](#) - "with" 语句
- Python `with` 语句的规范描述、背景和示例。

3.3.10. 特殊方法查找

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

```
1. >>> class C:
2. ...     pass
3. ...
4. >>> c = C()
5. >>> c.__len__ = lambda: 5
6. >>> len(c)
7. Traceback (most recent call last):
8.   File "<stdin>", line 1, in <module>
9. TypeError: object of type 'C' has no len()
```

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

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

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

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

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

```
1. >>> class Meta(type):
2. ...     def __getattr__(*args):
3. ...         print("Metaclass getattr invoked")
4. ...         return type.__getattr__(*args)
5. ...
6. >>> class C(object, metaclass=Meta):
7. ...     def __len__(self):
8. ...         return 10
9. ...     def __getattr__(*args):
10. ...         print("Class getattr invoked")
11. ...         return object.__getattr__(*args)
12. ...
```

```
13. >>> c = C()
14. >>> c.__len__()           # Explicit lookup via instance
15. Class getattribute invoked
16. 10
17. >>> type(c).__len__(c)    # Explicit lookup via type
18. Metaclass getattribute invoked
19. 10
20. >>> len(c)                # Implicit lookup
21. 10
```

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

3.4. 协程

- [3.4.1. 可等待对象](#)
- [3.4.2. 协程对象](#)
- [3.4.3. 异步迭代器](#)
- [3.4.4. 异步上下文管理器](#)

3.4.1. 可等待对象

`awaitable` 对象主要实现了 `await()` 方法。从 `async def` 函数返回的 `Coroutine` 对象即属于可等待对象。

注解

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

- `object. await (self)`
- 必须返回一个 `iterator`。应当被用来实现 `awaitable` 对象。例如，`asyncio.Future` 实现了此方法以与 `await` 表达式相兼容。

3.5 新版功能.

参见

PEP 492 了解有关可等待对象的详细信息。

3.4.2. 协程对象

`Coroutine` 对象属于 `awaitable` 对象。协程的执行可通过调用 `await()` 并迭代其结果来进行控制。当协程结束执行并返回时，迭代器会引发 `StopIteration`，该异常的 `value` 属性将指向返回值。如果协程引发了异常，它会被迭代器所传播。协程不应该直接引发未处理的 `StopIteration` 异常。

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

在 3.5.2 版更改：等待一个协程超过一次将引发 `RuntimeError`。

- `coroutine.send(value)`
- 开始或恢复协程的执行。如果 `value` 为 `None`，则这相当于前往 `await()` 所返回迭代器的下一项。如果 `value` 不为 `None`，此方法将委托给导致协程挂起的迭代器的 `send()` 方法。其结果（返回值，`StopIteration` 或是其他异常）将与上述对 `await()` 返回值进行迭代的结果相同。
- `coroutine.throw(type[, value[, traceback]])`
- 在协程内引发指定的异常。此方法将委托给导致协程挂起的迭代器的 `throw()` 方法，如果存在该方法。否则的话，异常会在挂起点被引发。其结果（返回值，`StopIteration` 或是其他异常）将与上述对 `await()` 返回值进行迭代的结果相同。如果异常未在协程内被捕获，则将回传给调用者。
- `coroutine.close()`
- 此方法会使得协程清理自身并退出。如果协程被挂起，此方法会先委托给导致协程挂起的迭代器的 `close()` 方法，如果存在该方法。然后它会在挂起点引发 `GeneratorExit`，使得协程立即清理自身。最后，协程会被标记为已结束执行，即使它根本未被启动。

当协程对象将要被销毁时，会使用以上处理过程来自动关闭。

3.4.3. 异步迭代器

异步迭代器 可以在其 `anext` 方法中调用异步代码。

异步迭代器可在 `async for` 语句中使用。

- `object. aiter (self)`
- 必须返回一个 异步迭代器 对象。
- `object. anext (self)`
- 必须返回一个 可迭代对象 输出迭代器的下一结果值。 当迭代结束时应该引发 `StopAsyncIteration` 错误。

异步可迭代对象的一个示例：

```

1. class Reader:
2.     async def readline(self):
3.         ...
4.
5.     def __aiter__(self):
6.         return self
7.
8.     async def __anext__(self):
9.         val = await self.readline()
10.        if val == b'':
11.            raise StopAsyncIteration
12.        return val

```

3.5 新版功能.

在 3.7 版更改：在 Python 3.7 之前，`aiter` 可以返回一个 可迭代对象 并解析为 异步迭代器。

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

3.4.4. 异步上下文管理器

异步上下文管理器 是 上下文管理器 的一种，它能够在其 `aenter` 和 `aexit` 方法中暂停执行。

异步上下文管理器可在 `async with` 语句中使用。

- `object. aenter (self)`
- 在语义上类似于 `enter()`，仅有的区别是它必须返回一个 可等待对象。
- `object. aexit (self, exc_type, exc_value, traceback)`
- 在语义上类似于 `exit()`，仅有的区别是它必须返回一个 可等待对象。

异步上下文管理器类的一个示例：

```
1. class AsyncContextManager:
2.     async def __aenter__(self):
3.         await log('entering context')
4.
5.     async def __aexit__(self, exc_type, exc, tb):
6.         await log('exiting context')
```

3.5 新版功能.

脚注

- [1](#)
- 在某些情况下 有可能 基于可控的条件改变一个对象的类型。 但这通常不是个好主意，因为如果处理不当会导致一些非常怪异的行为。
- [2](#)
- `hash()`，`iter()`，`reversed()` 以及 `contains()` 方法对此有特殊处理；其他方法仍会引发 `TypeError`，但可能依靠 `None` 属于不可调用对象的行为来做到这一点。
- [3](#)
- 这里的“不支持”是指该类无此方法，或方法返回 `NotImplemented`。 如果你想强制回退到右操作数的反射方法，请不要设置方法为 `None` — 那会造成显式地 阻塞 此种回退的相反效果。
- [4](#)
- 对于相同类型的操作数，如果非反射方法（例如 `add()`）失败则会认为相应运算不被支持，这就是反射方法未被调用的原因。

4. 执行模型

- 4.1. 程序的结构
- 4.2. 命名与绑定
- 4.3. 异常

4.1. 程序的结构

Python 程序是由代码块构成的。 代码块 是被作为一个单元来执行的一段 Python 程序文本。 以下几个都是代码块：模块、函数体和类定义。 交互式输入的每条命令都是一个代码块。 一个脚本文件（作为标准输入发送给解释器或是作为命令行参数发送给解释器的文件）也是一个代码块。 一条脚本命令（通过 `-c` 选项在解释器命令行中指定的命令）也是一个代码块。 传递给内置函数 `eval()` 和 `exec()` 的字符串参数也是代码块。

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

4.2. 命名与绑定

4.2.1. 名称的绑定

名称 用于指代对象。 名称是通过名称绑定操作来引入的。

以下构造会绑定名称：传给函数的正式形参， `import` 语句，类与函数定义（这会在定义的代码块中绑定类或函数名称）以及发生以标识符为目标的赋值， `for` 循环的开头，或 `with` 语句和 `except` 子句的 `as` 之后。 `import` 语句的 `from ... import *` 形式会绑定在被导入模块中定义的所有名称，那些以下划线开头的除外。 这种形式仅在模块层级上使用。

`del` 语句的目标也被视作一种绑定（虽然其实际语义为解除名称绑定）。

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

如果名称绑定在一个代码块中，则为该代码块的局部变量，除非声明为 `nonlocal` 或 `global`。 如果名称绑定在模块层级，则为全局变量。（模块代码块的变量既为局部变量又为全局变量。） 如果变量在一个代码块中被使用但不是在其中定义，则为 自由变量。

每个在程序文本中出现的名称是指由以下名称解析规则所建立的对该名称的 绑定。

4.2.2. 名称的解析

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

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

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

如果一个代码块内的任何位置发生名称绑定操作，则代码块内所有对该名称的使用会被认为是对当前代码块的引用。 当一个名称在其被绑定前就在代码块内被使用时则会导致错误。 这个一个很微妙的规则。 Python 缺少声明语法，并允许名称绑定操作发生于代码块内的任何位置。 一个代码块的局部变量可通过在整个代码块文本中扫描名称绑定操作来确定。

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

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

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

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

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

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

4.2.3. 内置命名空间和受限的执行

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

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

4.2.4. 与动态特性的交互

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

```
1. i = 10
2. def f():
3.     print(i)
4. i = 42
5. f()
```

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

4.3. 异常

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

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

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

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

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

注解

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

另请参看 [try 语句](#) 小节中对 `try` 语句的描述以及 [raise 语句](#) 小节中对 `raise` 语句的描述。

脚注

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

5. 导入系统

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

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

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

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

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

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

5.1. importlib

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

5.2. 包

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

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

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

所有模块都有自己的名字。 子包名与其父包名以点号分隔，与 Python 的标准属性访问语法一致。 例如你可能看到一个名为 `sys` 的模块，以及一个名为 `email` 的包，这个包又有一个名为 `email.mime` 的子包和该子包中的名为 `email.mime.text` 的子包。

5.2.1. 常规包

Python 定义了两类包，**常规包** 和 **命名空间包**。 常规包是传统的包类型，它们在 Python 3.2 及之前就已存在。 常规包通常以一个包含 `__init__.py` 文件的目录形式实现。 当一个常规包被导入时，这个 `__init__.py` 文件会隐式地被执行，它所定义的对象会被绑定到该包命名空间中的名称。 `__init__.py` 文件可以包含与任何其他模块中所包含的 Python 代码相似的代码，Python 将在模块被导入时为其添加额外的属性。

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

```
1. parent/  
2.     __init__.py  
3.     one/  
4.         __init__.py  
5.     two/  
6.         __init__.py  
7.     three/  
8.         __init__.py
```

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

5.2.2. 命名空间包

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

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

命名空间包没有 `parent/__init__.py` 文件。 实际上，在导入搜索期间可能找到多个 `parent` 目录，每个都由不同

的部分所提供。 因此 `parent/one` 的物理位置不一定与 `parent/two` 相邻。 在这种情况下，Python 将为顶级的 `parent` 包创建一个命名空间包，无论是它本身还是它的某个子包被导入。

另请参阅 [PEP 420](#) 了解对命名空间包的规格描述。

5.3. 搜索

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

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

5.3.1. 模块缓存

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

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

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

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

5.3.2. 查找器和加载器

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

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

导入机制是可扩展的，因此可以加入新的查找器以扩展模块搜索的范围和作用域。

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

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

在 3.4 版更改：在之前的 Python 版本中，查找器会直接返回 [加载器](#)，现在它们则返回模块规格说明，其中包含 [加载器](#)。加载器仍然在导入期间被使用，但负担的任务有所减少。

5.3.3. 导入钩子

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

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

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

5.3.4. 元路径

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

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

元路径查找器的 `find_spec()` 方法调用带有两到三个参数。 第一个是被导入模块的完整限定名称，例如 `foo.bar.baz`。 第二个参数是供模块搜索使用的路径条目。 对于最高层级模块，第二个参数为 `None`，但对于子模块或子包，第二个参数为父包 `path` 属性的值。 如果相应的 `path` 属性无法访问，将引发 `ModuleNotFoundError`。 第三个参数是一个将被作为稍后加载目标的现有模块对象。 导入系统仅会在重加载期间传入一个目标模块。

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

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

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

在 3.4 版更改：元路径查找器的 `find_spec()` 方法替代了 `find_module()`，后者现已弃用，它将继续可用但不会再做改变，导入机制仅会在查找器未实现 `find_spec()` 时尝试使用它。

5.4. 加载

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

```

1. module = None
2. if spec.loader is not None and hasattr(spec.loader, 'create_module'):
3.     # It is assumed 'exec_module' will also be defined on the loader.
4.     module = spec.loader.create_module(spec)
5. if module is None:
6.     module = ModuleType(spec.name)
7. # The import-related module attributes get set here:
8. _init_module_attrs(spec, module)
9.
10. if spec.loader is None:
11.     # unsupported
12.     raise ImportError
13. if spec.origin is None and spec.submodule_search_locations is not None:
14.     # namespace package
15.     sys.modules[spec.name] = module
16. elif not hasattr(spec.loader, 'exec_module'):
17.     module = spec.loader.load_module(spec.name)
18.     # Set __loader__ and __package__ if missing.
19. else:
20.     sys.modules[spec.name] = module
21.     try:
22.         spec.loader.exec_module(module)
23.     except BaseException:
24.         try:
25.             del sys.modules[spec.name]
26.         except KeyError:
27.             pass
28.         raise
29. return sys.modules[spec.name]
```

请注意以下细节：

- 如果在 `sys.modules` 中存在指定名称的模块对象，导入操作会已经将其返回。
- 在加载器执行模块代码之前，该模块将存在于 `sys.modules` 中。这一点很关键，因为该模块代码可能（直接或间接地）导入其自身；预先将其添加到 `sys.modules` 可防止在最坏情况下的无限递归和最好情况下的多次加载。
- 如果加载失败，则该模块 — 只限加载失败的模块 — 将从 `sys.modules` 中移除。任何已存在于 `sys.modules` 缓存的模块，以及任何作为附带影响被成功加载的模块仍会保留在缓存中。这与重新加载不同，后者会把即使加载失败的模块也保留在 `sys.modules` 中。
- 在模块创建完成但还未执行之前，导入机制会设置导入相关模块属性（在上面的示例伪代码中为 “_init_module_attrs”），详情参见 [后续部分](#)。
- 模块执行是加载的关键时刻，在此期间将填充模块的命名空间。执行会完全委托给加载器，由加载器决定要填充的内容和方式。
- 在加载过程中创建并传递给 `exec_module()` 的模块并不一定就是在导入结束时返回的模块 [2](#)。

在 3.4 版更改：导入系统已经接管了加载器建立样板的责任。 这些在以前是由

`importlib.abc.Loader.load_module()` 方法来执行的。

5.4.1. 加载器

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

加载器必须满足下列要求：

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

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

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

3.4 新版功能：加载器的 `create_module()` 方法。

在 3.4 版更改：`load_module()` 方法被 `exec_module()` 所替代，导入机制会对加载的所有样板责任作出假定。

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

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

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

在 3.5 版更改：当 `exec_module()` 已定义但 `create_module()` 未定义时将引发 `DeprecationWarning`。

在 3.6 版更改：当 `exec_module()` 已定义但 `create_module()` 未定义时将引发 `ImportError`。

5.4.2. 子模块

当使用任意机制（例如 `importlib` API，`import` 及 `import-from` 语句或者内置的 `import()`）加载一个子模块时，父模块的命名空间中会添加一个对子模块对象的绑定。 例如，如果包 `spam` 有一个子模块 `foo`，

则在导入 `spam.foo` 之后，`spam` 将具有一个 绑定到相应子模块的 `foo` 属性。 假如现在有如下的目录结构：

```
1. spam/
2.     __init__.py
3.     foo.py
4.     bar.py
```

并且 `spam/init.py` 中有如下几行内容：

```
1. from .foo import Foo
2. from .bar import Bar
```

则执行如下代码将在 `spam` 模块中添加对 `foo` 和 `bar` 的名称绑定：

```
1. >>> import spam
2. >>> spam.foo
3. <module 'spam.foo' from '/tmp/imports/spam/foo.py'>
4. >>> spam.bar
5. <module 'spam.bar' from '/tmp/imports/spam/bar.py'>
```

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

5.4.3. 模块规格说明

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

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

模块的规格说明会作为模块对象的 `spec` 属性对外公开。 有关模块规格的详细内容请参阅 `ModuleSpec` 。

3.4 新版功能.

5.4.4. 导入相关的模块属性

导入机制会在加载期间会根据模块的规格说明填充每个模块对象的这些属性，并在加载器执行模块之前完成。

- `name`
- `name` 属性必须被设为模块的完整限定名称。 此名称被用来在导入系统中唯一地标识模块。
- `loader`

- `loader` 属性必须被设为导入系统在加载模块时使用的加载器对象。这主要是用于自省，但也可用于额外的加载器专用功能，例如获取关联到加载器的数据。
- `package`
- 模块的 `package` 属性必须设定。其取值必须为一个字符串，但可以与 `name` 取相同的值。当模块是包时，其 `package` 值应该设为其 `name` 值。当模块不是包时，对于最高层级模块 `package` 应该设为空字符串，对于子模块则应该设为其父包名。更多详情可参阅 [PEP 366](#)。

该属性取代 `name` 被用来为主模块计算显式相对导入，相关定义见 [PEP 366](#)。预期它与 `spec.parent` 具有相同的值。

在 3.6 版更改：`package` 预期与 `spec.parent` 具有相同的值。

- `spec`
- `spec` 属性必须设为在导入模块时要使用的模块规格说明。对 `spec` 的正确设定将同时作用于 [解释器启动期间初始化的模块](#)。唯一的例外是 `main`，其中的 `spec` 会 [在某些情况下](#) 设为 `None`。

当 `package` 未定义时，`spec.parent` 会被用作回退项。

3.4 新版功能.

在 3.6 版更改：当 `package` 未定义时，`spec.parent` 会被用作回退项。

- `path`
- 如果模块为包（不论是正规包还是命名空间包），则必须设置模块对象的 `path` 属性。属性值必须为可迭代对象，但如果 `path` 没有进一步的用处则可以为空。如果 `path` 不为空，则在迭代时它应该产生字符串。有关 `path` 语义的更多细节将在 [下文](#) 中给出。

不是包的模块不应该具有 `path` 属性。

- `file`
- `cached`
- `file` 是可选项。如果设置，此属性的值必须为字符串。导入系统可以选择在其没有语法意义时不设置 `file`（例如从数据库加载的模块）。

如果设置了 `file`，则也可以再设置 `cached` 属性，后者取值为编译版本代码（例如字节码文件）所在的路径。设置此属性不要求文件已存在；该路径可以简单地指向应该存放编译文件的位置（参见 [PEP 3147](#)）。

当未设置 `file` 时也可以设置 `cached`。但是，那样的场景很不典型。最终，加载器会使用 `file` 和/或 `cached`。因此如果一个加载器可以从缓存加载模块但是不能从文件加载，那种非典型场景就是适当的。

5.4.5. module.path

根据定义，如果一个模块具有 `path` 属性，它就是包。

包的 `path` 属性会在导入其子包期间被使用。在导入机制内部，它的功能与 `sys.path` 基本相同，即在导入期间提供一个模块搜索位置列表。但是，`path` 通常会比 `sys.path` 受到更多限制。

`path` 必须是由字符串组成的可迭代对象，但它也可以为空。作用于 `sys.path` 的规则同样适用于包的

`path`，并且 `sys.path_hooks`（见下文）会在遍历包的 `path` 时被查询。

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

5.4.6. 模块的 repr

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

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

以下是所使用的确切规则：

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

在 3.4 版更改：`__loader__.__module__.__repr__()` 已弃用，导入机制现在使用模块规格说明来生成模块 `__repr__`。

为了向后兼容 Python 3.3，如果加载器定义了 `__module__.__repr__()` 方法，则会在尝试上述两种方式之前先调用该方法来生成模块 `__repr__`。但请注意此方法已弃用。

5.4.7. 已缓存字节码的失效

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

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

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

5.5. 基于路径的查找器

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

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

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

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

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

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

`sys.meta_path` 遍历。

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

5.5.1. 路径条目查找器

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

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

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

`sys.path` 包含一个提供模块和包搜索位置的字符串列表。它初始化自 `PYTHONPATH` 环境变量以及多种其他特定安装和实现的默认设置。`sys.path` 条目可指定的名称有文件系统中的目录、`zip` 文件和其他可用于搜索模块的潜在“位置” (参见 `site` 模块)，例如 URL 或数据库查询等。在 `sys.path` 中只能出现字符串和字节串；所有其他数据类型都会被忽略。字节串条目使用的编码由单独的 `路径条目查找器` 来确定。

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

基于路径的查找器会迭代搜索路径中的每个条目，并且每次都查找与路径条目对应的 `path entry finder`

(`PathEntryFinder`)。因为这种操作可能很耗费资源（例如搜索会有 `stat()` 调用的开销），基于路径的查找器会维持一个缓存来将路径条目映射到路径条目查找器。这个缓存放于 `sys.path_importer_cache`（尽管如此命名，但这个缓存实际存放的是查找器对象而非仅限于 `importer` 对象）。通过这种方式，对特定 `path entry` 位置的 `path entry finder` 的高耗费搜索只需进行一次。用户代码可以自由地从 `sys.path_importer_cache` 移除缓存条目，以强制基于路径的查找器再次执行路径条目搜索 3。

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

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

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

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

5.5.2. 路径条目查找器协议

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

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

为了向导入机制提示该规格说明代表一个命名空间的 `portion`。路径条目查找器会将规格说明中的 "loader" 设为 `None` 并将 "submodule_search_locations" 设为一个包含该部分的列表。

在 3.4 版更改：`find_spec()` 替代了 `find_loader()` 和 `find_module()`，后两者现在都已弃用，但会在 `find_spec()` 未定义时被使用。

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

`find_loader()` 接受一个参数，即被导入模块的完整限定名称。`find_loader()` 会返回一个二元组，其中第一项为加载器，第二项为一个命名空间 `portion`。当第一项（即加载器）为 `None` 时，这意味着路径条目查找器虽然没有指定名称模块的加载器，但它知道该路径条目为指定名称模块提供了一个命名空间部分。这几乎总是表明一种情况，即 Python 被要求导入一个并不以文件系统中的实体形式存在的命名空间包。当一个路径条目查找器返回的加载器为 `None` 时，该二元组返回值的第二项必须为一个序列，不过它也可以为空。

如果 `find_loader()` 所返回加载器的值不为 `None`，则该部分会被忽略，而该加载器会自基于路径的查找器返

回，终止对路径条目的搜索。

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

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

5.6. 替换标准导入系统

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

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

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

5.7. 包相对导入

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

```
1. package/  
2.     __init__.py  
3.     subpackage1/  
4.         __init__.py  
5.         moduleX.py  
6.         moduleY.py  
7.     subpackage2/  
8.         __init__.py  
9.         moduleZ.py  
10.    moduleA.py
```

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

```
1. from .moduleY import spam  
2. from .moduleY import spam as ham  
3. from . import moduleY  
4. from ..subpackage1 import moduleY  
5. from ..subpackage2.moduleZ import eggs  
6. from ..moduleA import foo
```

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

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

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

5.8. 有关 main 的特殊事项

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

5.8.1. main.spec

根据 `main` 被初始化的方式，`main.spec` 会被设置相应值或是 `None`。

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

在 [其余的情况](#) 下 `main.spec` 会被设为 `None`，因为用于填充 `main` 的代码不直接与可导入的模块相对应：

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

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

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

5.9. 开放问题项

XXX 最好是能增加一个图表。

XXX * (`import_machinery.rst`) 是否要专门增加一节来说明模块和包的属性，也许可以扩展或移植数据模型参考页中的相关条目？

XXX 库手册中的 `runpy` 和 `pkgutil` 等等应该都在页面顶端增加指向新的导入系统章节的“另请参阅”链接。

XXX 是否要增加关于初始化 `main` 的不同方式的更多解释？

XXX 增加更多有关 `main` 怪异/坑人特性的信息（例如直接从 [PEP 395](#) 复制）。

5.10. 参考文献

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

原始的 `sys.meta_path` 规格说明见 [PEP 302](#)，后续的扩展说明见 [PEP 420](#)。

[PEP 420](#) 为 Python 3.3 引入了 [命名空间包](#)。[PEP 420](#) 还引入了 `find_loader()` 协议作为 `find_module()` 的替代。

[PEP 366](#) 描述了新增的 `package` 属性，用于在模块中的显式相对导入。

[PEP 328](#) 引入了绝对和显式相对导入，并初次提出了 `name` 语义，最终由 [PEP 366](#) 为 `package` 加入规范描述。

[PEP 338](#) 定义了将模块作为脚本执行。

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

脚注

- [1](#)
- 参见 `types.ModuleType`。
- [2](#)
- `importlib` 实现避免直接使用返回值。而是通过在 `sys.modules` 中查找模块名称来获取模块对象。这种方式的间接影响是被导入的模块可能在 `sys.modules` 中替换其自身。这属于具体实现的特定行为，不保证能在其他 Python 实现中起作用。
- [3](#)
- 在遗留代码中，有可能在 `sys.path_importer_cache` 中找到 `imp.NullImporter` 的实例。建议将这些代码修改为使用 `None` 代替。详情参见 [Porting Python code](#)。

6. 表达式

本章将解释 Python 中组成表达式的各种元素的含义。

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

```
1. name ::= othername
```

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

6.1. 算术转换

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

- 如果任一参数为复数，另一参数会被转换为复数；
- 否则，如果任一参数为浮点数，另一参数会被转换为浮点数；
- 否则，两者应该都为整数，不需要进行转换。

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

6.2. 原子

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

```
1. atom ::= identifier | literal | enclosure
2. enclosure ::= parenth_form | list_display | dict_display | set_display
3.                | generator_expression | yield_atom
```

6.2.1. 标识符（名称）

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

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

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

6.2.2. 字面值

Python 支持字符串和字节串字面值，以及几种数字字面值：

```
1. literal ::= stringliteral | bytesliteral
2.                | integer | floatnumber | imagnumber
```

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

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

6.2.3. 带圆括号的形式

带圆括号的形式是包含在圆括号中的可选表达式列表。

```
1. parenth_form ::= "(" [starred_expression] ")"
```

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

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

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

6.2.4. 列表、集合与字典的显示

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

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

推导式的常用句法元素为：

```
1. comprehension ::= expression comp_for
2. comp_for      ::= ["async"] "for" target_list "in" or_test [comp_iter]
3. comp_iter     ::= comp_for | comp_if
4. comp_if      ::= "if" expression_nocond [comp_iter]
```

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

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

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

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

从 Python 3.6 开始，在 `async def` 函数中可以使用 `async for` 子句来迭代 `asynchronous iterator`。 在 `async def` 函数中构建推导式可以通过在打头的表达式后加上 `for` 或 `async for` 子句，也可能包含额外的 `for` 或 `async for` 子句，还可能使用 `await` 表达式。 如果一个推导式包含 `async for` 子句或者 `await` 表达式，则被称为 异步推导式。 异步推导式可以暂停执行它所在的协程函数。 另请参阅 [PEP 530](#)。

3.6 新版功能：引入了异步推导式。

在 3.8 版更改： `yield` 和 `yield from` 在隐式嵌套的作用域中已被禁用。

6.2.5. 列表显示

列表显示是一个用方括号括起来的可能为空的表达式系列：

```
1. list_display ::= "[" [starred_list | comprehension] "]"
```

列表显示会产生一个新的列表对象，其内容通过一系列表达式或一个推导式来指定。当提供由逗号分隔的一系列表达式时，其元素会从左至右被求值并按此顺序放入列表对象。当提供一个推导式时，列表会根据推导式所产生的结果元素进行构建。

6.2.6. 集合显示

集合显示是用花括号标明的，与字典显示的区别在于没有冒号分隔的键和值：

```
1. set_display ::= "{" (starred_list | comprehension) "}"
```

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

空集合不能用 `{}` 来构建；该面值所构建的是一个空字典。

6.2.7. 字典显示

字典显示是一个用花括号括起来的可能为空的键/数据对序列：

```
1. dict_display      ::= "{" [key_datum_list | dict_comprehension] "}"
2. key_datum_list    ::= key_datum ("," key_datum)* ["," ]
3. key_datum         ::= expression ":" expression | "***" or_expr
4. dict_comprehension ::= expression ":" expression comp_for
```

字典显示会产生一个新的字典对象。

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

双星号 `**` 表示字典拆包。它的操作数必须是一个 [mapping](#)。每个映射项被加入新的字典。后续的值会替代先前的键/数据对和先前的字典拆包所设置的值。

3.5 新版功能：拆包到字典显示，最初由 [PEP 448](#) 提出。

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

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

在 3.8 版更改：在 Python 3.8 之前的字典推导式中，并没有定义好键和值的求值顺序。在 CPython 中，值会先于键被求值。根据 [PEP 572](#) 的提议，从 3.8 开始，键会先于值被求值。

6.2.8. 生成器表达式

生成器表达式是用圆括号括起来的紧凑形式生成器标注。

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

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

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

圆括号在只附带一个参数的调用中可以被省略。详情参见 [调用](#) 一节。

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

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

3.6 新版功能：引入了异步生成器表达式。

在 3.7 版更改：在 Python 3.7 之前，异步生成器表达式只能在 `async def` 协和中出现。从 3.7 开始，任何函数都可以使用异步生成器表达式。

在 3.8 版更改：`yield` 和 `yield from` 在隐式嵌套的作用域中已被禁用。

6.2.9. yield 表达式

```
1. yield_atom ::= "(" yield_expression ")"
2. yield_expression ::= "yield" [expression_list | "from" expression]
```

`yield` 表达式在定义 `generator` 函数或是 `asynchronous generator` 的时候才会用到。因此只能在函数定义的内部使用 `yield` 表达式。在一个函数体内使用 `yield` 表达式会使这个函数变成一个生成器，并且在一个 `async def` 定义的函数体内使用 `yield` 表达式会让协程函数变成异步的生成器。比如说：

```
1. def gen(): # defines a generator function
2.     yield 123
3.
4. async def agen(): # defines an asynchronous generator function
5.     yield 123
```

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

在 3.8 版更改：禁止在实现推导式和生成器表达式的隐式嵌套作用域中使用 `yield` 表达式。

下面是对生成器函数的描述，异步生成器函数会在 [异步生成器函数](#) 一节中单独介绍。

当一个生成器函数被调用的时候，它返回一个迭代器，称为生成器。然后这个生成器来控制生成器函数的执行。当这个生成器的某一个方法被调用的时候，生成器函数开始执行。这时会一直执行到第一个 `yield` 表达式，在此执行再次被挂起，给生成器的调用者返回 `expression_list` 的值。挂起后，我们说所有局部状态都被保留下来，包括局部变量的当前绑定，指令指针，内部求值栈和任何异常处理的状态。通过调用生成器的某一个方法，生成器函数继续执行。此时函数的运行就和 `yield` 表达式只是一个外部函数调用的情况完全一致。恢复后 `yield` 表达式的值取决于调用的哪个方法来恢复执行。如果用的是 `next()`（通常通过语言内置的 `for` 或是 `next()` 来调用）那么结果就是 `None`。否则，如果用 `send()`，那么结果就是传递给 `send` 方法的值。

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

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

当使用 `yield from <expr>` 时，它会将所提供的表达式视为一个子迭代器。这个子迭代器产生的所有值都直接被传递给当前生成器方法的调用者。通过 `send()` 传入的任何值以及通过 `throw()` 传入的任何异常如果有适当的方法则会被传给下层迭代器。如果不是这种情况，那么 `send()` 将引发 `AttributeError` 或 `TypeError`，而 `throw()` 将立即引发所传入的异常。

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

在 3.3 版更改：添加 `yield from <expr>` 以委托控制流给一个子迭代器。

当 `yield` 表达式是赋值语句右侧的唯一表达式时，括号可以省略。

参见

- [PEP 255](#) - 简单生成器
- 在 Python 中加入生成器和 `yield` 语句的提议。
- [PEP 342](#) - 通过增强型生成器实现协程
- 增强生成器 API 和语法的提议，使其可以被用作简单的协程。
- [PEP 380](#) - 委托给子生成器的语法
- 引入 `yield from` 语法以方便地委托给子生成器的提议。
- [PEP 525](#) - 异步生成器
- 通过给协程函数加入生成器功能对 [PEP 492](#) 进行扩展的提议。

6.2.9.1. 生成器-迭代器的方法

这个子小节描述了生成器迭代器的方法。它们可被用于控制生成器函数的执行。

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

- `generator. next ()`
- 开始一个生成器函数的执行或是从上次执行的 `yield` 表达式位置恢复执行。 当一个生成器函数通过 `next()` 方法恢复执行时，当前的 `yield` 表达式总是取值为 `None`。随后会继续执行到下一个 `yield` 表达式，其 `expression_list` 的值会返回给 `next()` 的调用者。如果生成器没有产生下一个值就退出，则将引发 `StopIteration` 异常。

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

- `generator. send (value)`
- 恢复执行并向生成器函数“发送”一个值。 `value` 参数将成为当前 `yield` 表达式的结果。 `send()` 方法会返回生成器所产生的下一个值，或者如果生成器没有产生下一个值就退出则会引发 `StopIteration`。当调用 `send()` 来启动生成器时，它必须以 `None` 作为调用参数，因为这时没有可以接收值的 `yield` 表达式。
- `generator. throw (type[, value[, traceback]])`
- 在生成器暂停的位置引发 `type` 类型的异常，并返回该生成器函数所产生的下一个值。如果生成器没有产生下一个值就退出，则将引发 `StopIteration` 异常。如果生成器函数没有捕获传入的异常，或引发了另一个异常，则该异常会被传播给调用者。
- `generator. close ()`
- 在生成器函数暂停的位置引发 `GeneratorExit`。如果之后生成器函数正常退出、关闭或引发 `GeneratorExit`（由于未捕获该异常）则关闭并返回其调用者。如果生成器产生了一个值，关闭会引发 `RuntimeError`。如果生成器引发任何其他异常，它会被传播给调用者。如果生成器已经由于异常或正常退出则 `close()` 不会做任何事。

6.2.9.2. 示例

这里是一个简单的例子，演示了生成器和生成器函数的行为：

```
1. >>> def echo(value=None):
2. ...     print("Execution starts when 'next()' is called for the first time.")
3. ...     try:
4. ...         while True:
5. ...             try:
6. ...                 value = (yield value)
7. ...             except Exception as e:
8. ...                 value = e
9. ...         finally:
10. ...             print("Don't forget to clean up when 'close()' is called.")
11. ...
12. >>> generator = echo(1)
13. >>> print(next(generator))
14. Execution starts when 'next()' is called for the first time.
15. 1
```

```

16. >>> print(next(generator))
17. None
18. >>> print(generator.send(2))
19. 2
20. >>> generator.throw(TypeError, "spam")
21. TypeError('spam',)
22. >>> generator.close()
23. Don't forget to clean up when 'close()' is called.

```

对于 `yield from` 的例子，参见“Python 有什么新变化”中的 [PEP 380：委托给子生成器的语法](#)。

6.2.9.3. 异步生成器函数

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

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

调用异步生成器的方法之一将返回 `awaitable` 对象，执行会在此对象被等待时启动。到那时，执行将前往第一个 `yield` 表达式，在那里它会再次暂停，将 `expression_list` 的值返回给等待中的协程。与生成器一样，挂起意味着局部的所有状态会被保留，包括局部变量的当前绑定、指令的指针、内部求值的堆栈以及任何异常处理的状态。当执行在等待异步生成器的方法返回下一个对象后恢复时，该函数可以从原状态继续进行，就仿佛 `yield` 表达式只是另一个外部调用。恢复执行之后 `yield` 表达式的值取决于恢复执行所用的方法。如果使用 `anext()` 则结果为 `None`。否则的话，如果使用 `asend()` 则结果将是传递给该方法的价值。

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

为了能处理最终化，事件循环应该定义一个 `终结器` 函数，它接受一个异步生成器-迭代器且可能调用 `aclose()` 并执行协程。这个 `终结器` 可能通过调用 `sys.set_asyncgen_hooks()` 来注册。当首次迭代时，异步生成器-迭代器将保存已注册的 `终结器` 以便在最终化时调用。有关 `For a reference example of a 终结器` 方法的参考示例请查看 `Lib/asyncio/base_events.py` 中实现的 `asyncio.Loop.shutdown_asyncgens`。

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

6.2.9.4. 异步生成器-迭代器方法

这个子小节描述了异步生成器迭代器的方法，它们可被用于控制生成器函数的执行。

- `coroutine agen. anext ()`
- 返回一个可等待对象，它在运行时会开始执行该异步生成器或是从上次执行的 `yield` 表达式位置恢复执行。当一个异步生成器函数通过 `anext()` 方法恢复执行时，当前的 `yield` 表达式所返回的可等待对象总是取值为 `None`，它在运行时将继续执行到下一个 `yield` 表达式。该 `yield` 表达式的 `expression_list` 的值会是完成的协程所引发的 `StopIteration` 异常的值。如果异步生成器没有产生下一个值就退出，则该可等待

对象将引发 `StopAsyncIteration` 异常，提示该异步迭代操作已完成。

此方法通常是通过 `async for` 循环隐式地调用。

- `coroutine agen. asend (value)`
 - 返回一个可等待对象，它在运行时会恢复该异步生成器的执行。与生成器的 `send()` 方法一样，此方法会“发送”一个值给异步生成器函数，其 `value` 参数会成为当前 `yield` 表达式的结果值。`asend()` 方法所返回的可等待对象将返回生成器产生的下一个值，其值为所引发的 `StopIteration`，或者如果异步生成器没有产生下一个值就退出则引发 `StopAsyncIteration`。当调用 `asend()` 来启动异步生成器时，它必须以 `None` 作为调用参数，因为这时没有可以接收值的 `yield` 表达式。
- `coroutine agen. athrow (type[, value[, traceback]])`
 - 返回一个可等待对象，它会在异步生成器暂停的位置引发 `type` 类型的异常，并返回该生成器函数所产生的下一个值，其值为所引发的 `StopIteration` 异常。如果异步生成器没有产生下一个值就退出，则将由该可等待对象引发 `StopAsyncIteration` 异步。如果生成器函数没有捕获传入的异常，或引发了另一个异常，则当可等待对象运行时该异常会被传播给可等待对象的调用者。
- `coroutine agen. aclose ()`
 - 返回一个可等待对象，它会在运行时向异步生成器函数暂停的位置抛入一个 `GeneratorExit`。如果该异步生成器函数正常退出、关闭或引发 `GeneratorExit`（由于未捕获该异常）则返回的可等待对象将引发 `StopIteration` 异常。后续调用异步生成器所返回的任何其他可等待对象将引发 `StopAsyncIteration` 异常。如果异步生成器产生了一个值，该可等待对象会引发 `RuntimeError`。如果异步生成器引发任何其他异常，它会被传播给可等待对象的调用者。如果异步生成器已经由于异常或正常退出则后续调用 `aclose()` 将返回一个不会做任何事的可等待对象。

6.3. 原型

原型代表编程语言中最紧密绑定的操作。 它们的句法如下：

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

6.3.1. 属性引用

属性引用是后面带有一个句点加一个名称的原型：

```
1. attributeref ::= primary "." identifier
```

此原型必须求值为一个支持属性引用的类型的对象，多数对象都支持属性引用。 随后该对象会被要求产生以指定标识符为名称的属性。 这个产生过程可通过重载 `getattr()` 方法来自定义。 如果这个属性不可用，则将引发 `AttributeError` 异常。 否则的话，所产生对象的类型和值会根据该对象来确定。 对同一属性引用的多次求值可能产生不同的对象。

6.3.2. 抽取

抽取就是在序列（字符串、元组或列表）或映射（字典）对象中选择一项：

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

此原型必须求值为一个支持抽取操作的对象（例如列表或字典）。 用户定义的对象可通过定义 `getitem()` 方法来支持抽取操作。

对于内置对象，有两种类型的对象支持抽取操作：

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

如果原型为序列，表达式列表必须求值为一个整数或一个切片（详情见下节）。

正式句法规则并没有在序列中设置负标号的特殊保留条款；但是，内置序列所提供的 `getitem()` 方法都可通过在索引中添加序列长度来解析负标号（这样 `x[-1]` 会选出 `x` 中的最后一项）。 结果值必须为一个小于序列中项数的非负整数，抽取操作会选出标号为该值的项（从零开始数）。 由于对负标号和切片的支持存在于对象的 `getitem()` 方法，重载此方法的子类需要显式地添加这种支持。

字符串的项是字符。 字符不是单独的数据类型而是仅有一个字符的字符串。

6.3.3. 切片

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

```

1. slicing      ::= primary "[" slice_list "]"
2. slice_list  ::= slice_item ("," slice_item)* ["," ]
3. slice_item  ::= expression | proper_slice
4. proper_slice ::= [lower_bound] ":" [upper_bound] [ ":" [stride] ]
5. lower_bound ::= expression
6. upper_bound ::= expression
7. stride      ::= expression

```

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

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

6.3.4. 调用

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

```

1. call          ::= primary "(" [argument_list ["," ] | comprehension] ")"
2. argument_list ::= positional_arguments ["," starred_and_keywords]
3.               ["," keywords_arguments]
4.               | starred_and_keywords ["," keywords_arguments]
5.               | keywords_arguments
6. positional_arguments ::= ["*"] expression ("," ["*"] expression)*
7. starred_and_keywords ::= ("*" expression | keyword_item)
8.               ("*" expression | "," keyword_item)*
9. keywords_arguments  ::= (keyword_item | "*" expression)
10.               ("*" keyword_item | "," "*" expression)*
11. keyword_item       ::= identifier "=" expression

```

一个可选项为在位置和关键字参数后加上逗号而不影响语义。

此原型必须求值为一个可调用对象（用户定义的函数，内置函数，内置对象的方法，类对象，类实例的方法以及任何具有 `call()` 方法的对象都是可调用对象）。所有参数表达式将在尝试调用前被求值。请参阅 [函数定义](#) 一节了解正式的 [parameter](#) 列表句法。

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

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

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

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

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

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

```
1. >>> def f(a, b):
2. ...     print(a, b)
3. ...
4. >>> f(b=1, *(2,))
5. 2 1
6. >>> f(a=1, *(2,))
7. Traceback (most recent call last):
8.   File "<stdin>", line 1, in <module>
9.   TypeError: f() got multiple values for keyword argument 'a'
10. >>> f(1, *(2,))
11. 1 2
```

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

如果函数调用中出现了 `**expression` 句法，`expression` 必须求值为一个 `mapping`，其内容会被当作是额外的关键字参数。如果一个关键字已存在（作为显式关键字参数，或来自另一个拆包），则将引发 `TypeError` 异常。

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

在 3.5 版更改：函数调用接受任意数量的 `*` 和 `**` 拆包，位置参数可能跟在可迭代对象拆包（`*`）之后，而关键字参数可能跟在字典拆包（`**`）之后。由 [PEP 448**] (<https://www.python.org/dev/peps/pep-0448>) 发起最初提议。

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

如果类型为--

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

- 具体结果依赖于解释器；有关内置函数和方法的描述参见 [内置函数](#)。
- 类对象：
- 返回该类的一个新实例。
- 类实例方法：
- 调用相应的用户自定义函数，向其传入的参数列表会比调用的参数列表多一项：该实例将成为第一个参数。
- 类实例：
- 该类必须定义有 `call()` 方法；作用效果将等价于调用该方法。

6.4. await 表达式

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

```
1. await_expr ::= "await" primary
```

3.5 新版功能.

6.5. 幂运算符

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

```
1. power ::= (await_expr | primary) ["**" u_expr]
```

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

`-1**2` 结果将为 `-1`。

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

对于 `int` 类型的操作数，结果将具有与操作数相同的类型，除非第二个参数为负数；在那种情况下，所有参数会被转换为 `float` 类型并输出 `float` 类型的结果。 例如，`102` 返回 `100`，而 `10` `-2` 返回 `0.01`。

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

6.6. 一元算术和位运算

所有算术和位运算具有相同的优先级：

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

一元运算符 `-`（负）会产生其数值参数的负值。

一元运算符 `+`（正）会产生与其数值参数相同的值。

一元运算符 `~`（取反）的结果是对其整数参数按位取反。`x` 的按位取反被定义为 `-(x+1)`。它只作用于整数。

在所有三种情况下，如果参数的类型不正确，将引发 `TypeError` 异常。

6.7. 二元算术运算符

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

```
1. m_expr ::= u_expr | m_expr "*" u_expr | m_expr "@" m_expr |
2.          m_expr "/" u_expr | m_expr "/" u_expr |
3.          m_expr "%" u_expr
4. a_expr ::= m_expr | a_expr "+" m_expr | a_expr "-" m_expr
```

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

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

3.5 新版功能.

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

运算符 `%`（模）将输出第一个参数除以第二个参数的余数。 两个数字参数将先被转换为相同类型。 右参数为零将引发 `ZeroDivisionError` 异常。 参数可以为浮点数，例如 `3.14%0.7` 等于 `0.34`（因为 `3.14` 等于 `4*0.7 + 0.34`）。 模运算符的结果的正负总是与第二个操作数一致（或是为零）；结果的绝对值一定小于第二个操作数的绝对值 `1`。

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

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

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

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

运算符 `-`（减）将输出其参数的差。 两个数字参数将先被转换为相同类型。

6.8. 移位运算

移位运算的优先级低于算术运算：

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

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


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


6.9. 二元位运算

三种位运算具有各不相同的优先级：

1. **and_expr** ::= shift_expr | and_expr "&" shift_expr
2. **xor_expr** ::= and_expr | xor_expr "^" and_expr
3. **or_expr** ::= xor_expr | or_expr "|" xor_expr

运算符  对两个参数进行按位 AND（与）运算，两个参数必须为整数。

运算符  对两个参数进行按位 XOR（异或）运算，两个参数必须为整数。

运算符  对两个参数进行按位 OR（或）运算，两个参数必须为整数。

6.10. 比较运算

与 C 不同，Python 中所有比较运算的优先级相同，低于任何算术、移位或位运算。另一个与 C 不同之处在于

`a < b < c` 这样的表达式会按传统算术法则来解读：

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

比较运算将输出布尔值：`True` 或 `False`。

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

正式的说法是这样：如果 `a, b, c, ..., y, z` 为表达式而 `op1, op2, ..., opN` 为比较运算符，则 `a op1 b op2 c ... y opN z` 就等价于 `a op1 b and b op2 c and ... y opN z`，后者的不同之处只是每个表达式最多只被求值一次。

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

6.10.1. 值比较

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

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

由于所有类型都是 `object` 的（直接或间接）子类型，它们都从 `object` 继承了默认的比较行为。类型可以通过实现丰富比较方法例如 `lt()` 来定义自己的比较行为，详情参见 [基本定制](#)。

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

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

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

以下列表描述了最主要内置类型的比较行为。

- 内置数值类型（[数字类型](#) -- `int`，`float`，`complex`）以及标准库类型 `fractions.Fraction` 和 `decimal.Decimal` 可进行类型内部和跨类型的比较，例外限制是复数不支持次序比较。在类型相关的限制以内，它们会按数学（算法）规则正确进行比较且不会有精度损失。

非数字值 `float('NaN')` 和 `decimal.Decimal('NaN')` 属于特例。任何数字与非数字值的比较均返回假值。还有

一个反直觉的结果是非数字值不等于其自身。例如，如果 `x = float('NaN')` 则 `3 < x` , `x < 3` , `x == x` , `x != x` 均为假值。此行为是符合 IEEE 754 标准的。

- `None` 和 `NotImplemented` 都是单例对象。PEP 8 建议单例对象的比较应当总是通过 `is` 或 `is not` 而不是等于运算符来进行。
- 二进制码序列（`bytes` 或 `bytearray` 的实例）可进行类型内部和跨类型的比较。它们使用其元素的数字值按字典顺序进行比较。
- 字符串（`str` 的实例）使用其字符的 Unicode 码位数字值（内置函数 `ord()` 的结果）按字典顺序进行比较。3

字符串和二进制码序列不能直接比较。

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

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

内置多项集间的字典序比较规则如下：

- 两个多项集若要相等，它们必须为相同类型、相同长度，并且每对相应的元素都必须相等（例如，`[1,2] == (1,2)` 为假值，因为类型不同）。
- 对于支持次序比较的多项集，排序与其第一个不相等元素的排序相同（例如 `[1,2,x] <= [1,2,y]` 的值与 `x <= y` 相同）。如果对应元素不存在，较短的多项集排序在前（例如 `[1,2] < [1,2,3]` 为真值）。
- 两个映射（`dict` 的实例）若要相等，必须当且仅当它们具有相同的（键，值）对。键和值的一致性比较强制规定自反射性。

次序比较（`<` , `>` , `<=` 和 `>=`）将引发 `TypeError` 。

- 集合（`set` 或 `frozenset` 的实例）可进行类型内部和跨类型的比较。

它们将比较运算符定义为子集和超集检测。这类关系没有定义完全排序（例如 `{1,2}` 和 `{2,3}` 两个集合不相等，即不为彼此的子集，也不为彼此的超集。相应地，集合不适宜作为依赖于完全排序的函数的参数（例如如果给出一个集合列表作为 `min()` , `max()` 和 `sorted()` 的输入将产生未定义的结果）。

集合的比较强制规定其元素的自反射性。

- 大多数其他内置类型没有实现比较方法，因此它们会继承默认的比较行为。

在可能的情况下，用户定义类在定制其比较行为时应当遵循一些一致性规则：

- 相等比较应该是自反射的。换句话说，相同的对象比较时应该相等：

```
x is y 意味着 x == y
```

- 比较应该是对称的。换句话说，下列表达式应该有相同的结果：

`x == y` 和 `y == x`

`x != y` 和 `y != x`

`x < y` 和 `y > x`

`x <= y` 和 `y >= x`

- 比较应该是可传递的。 下列（简要的）例子显示了这一点：

`x > y and y > z` 意味着 `x > z`

`x < y and y <= z` 意味着 `x < z`

- 反向比较应该导致布尔值取反。 换句话说，下列表达式应该有相同的结果：

`x == y` 和 `not x != y`

`x < y` 和 `not x >= y` （对于完全排序）

`x > y` 和 `not x <= y` （对于完全排序）

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

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

Python 并不强制要求这些一致性规则。 实际上，非数字值就是一个不遵循这些规则的例子。

6.10.2. 成员检测运算

运算符 `in` 和 `not in` 用于成员检测。 如果 `x` 是 `s` 的成员则 `x in s` 求值为 `True`，否则为 `False`。 `x not in s` 返回 `x in s` 取反后的值。 所有内置序列和集合类型以及字典都支持此运算，对于字典来说 `in` 检测其是否有给定的键。 对于 `list`, `tuple`, `set`, `frozenset`, `dict` 或 `collections.deque` 这样的容器类型，表达式 `x in y` 等价于 `any(x is e or x == e for e in y)`。

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

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

对于未定义 `contains()` 但定义了 `iter()` 的用户自定义类来说，如果在对 `y` 进行迭代时产生了值 `z` 使得表达式 `x is z or x == z` 为真，则 `x in y` 为 `True`。 如果在迭代期间引发了异常，则等同于 `in` 引发了该异常。

最后将会尝试旧式的迭代协议：如果一个类定义了 `getitem()`，则当且仅当存在非负整数索引号 `i` 使得 `x is y[i] or x == y[i]` 并且没有更小的索引号引发 `IndexError` 异常时 `x in y` 为 `True`。（如果引发了任何其他异常，则等同于 `in` 引发了该异常）。

运算符 `not in` 被定义为具有与 `in` 相反的逻辑值。

6.10.3. 标识号比较

运算符 `is` 和 `is not` 用于检测对象的标识号：当且仅当 `x` 和 `y` 是同一对象时 `x is y` 为真。 一个对象的标识号可使用 `id()` 函数来确定。 `x is not y` 会产生相反的逻辑值。 4

6.11. 布尔运算

```
1. or_test ::= and_test | or_test "or" and_test
2. and_test ::= not_test | and_test "and" not_test
3. not_test ::= comparison | "not" not_test
```

在执行布尔运算的情况下，或是当表达式被用于流程控制语句时，以下值会被解析为假值：`False`，`None`，所有类型的数字零，以及空字符串和空容器（包括字符串、元组、列表、字典、集合与冻结集合）。所有其他值都会被解析为真值。用户自定义对象可通过提供 `bool()` 方法来定制其逻辑值。

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

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

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

请注意 `and` 和 `or` 都不限制其返回的值和类型必须为 `False` 和 `True`，而是返回最终求值的参数。此行为是有必要的，例如假设 `s` 为一个当其为空时应被替换为某个默认值的字符串，表达式 `s or 'foo'` 将产生希望的值。由于 `not` 必须创建一个新值，不论其参数为何种类型它都会返回一个布尔值（例如，`not 'foo'` 结果为 `False` 而非 `''`。）

6.12. 条件表达式

```
1. conditional_expression ::= or_test ["if" or_test "else" expression]
2. expression            ::= conditional_expression | lambda_expr
3. expression_nocond     ::= or_test | lambda_expr_nocond
```

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

表达式 `x if C else y` 首先是对条件 `C` 而非 `x` 求值。如果 `C` 为真，`x` 将被求值并返回其值；否则将对 `y` 求值并返回其值。

请参阅 [PEP 308](#) 了解有关条件表达式的详情。

6.13. lambda 表达式

```
1. lambda_expr      ::= "lambda" [parameter_list] ":" expression
2. lambda_expr_nocond ::= "lambda" [parameter_list] ":" expression_nocond
```

lambda 表达式（有时称为 lambda 构型）被用于创建匿名函数。表达式 `lambda parameters: expression` 会产生一个函数对象。该未命名对象的行为类似于用以下方式定义的函数：

```
1. def <lambda>(parameters):
2.     return expression
```

请参阅 [函数定义](#) 了解有关参数列表的句法。 请注意通过 lambda 表达式创建的函数不能包含语句或标注。

6.14. 表达式列表

```
1. expression_list    ::= expression ("," expression)* [","]
2. starred_list       ::= starred_item ("," starred_item)* [","]
3. starred_expression ::= expression | (starred_item ",")* [starred_item]
4. starred_item       ::= expression | "*" or_expr
```

除了作为列表或集合显示的一部分，包含至少一个逗号的表达式列表将生成一个元组。元组的长度就是列表中表达式的数量。表达式将从左至右被求值。

一个星号 `*` 表示 可迭代拆包。其操作数必须为一个 `iterable`。该可迭代对象将被拆解为迭代项的序列，并被包含于在拆包位置上新建的元组、列表或集合之中。

3.5 新版功能：表达式列表中的可迭代对象拆包，最初由 [PEP 448](#) 提出。

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

6.15. 求值顺序

Python 按从左至右的顺序对表达式求值。 但注意在对赋值操作求值时，右侧会先于左侧被求值。

在以下几行中，表达式将按其后缀的算术优先顺序被求值。：

```
1. expr1, expr2, expr3, expr4
2. (expr1, expr2, expr3, expr4)
3. {expr1: expr2, expr3: expr4}
4. expr1 + expr2 * (expr3 - expr4)
5. expr1(expr2, expr3, *expr4, **expr5)
6. expr3, expr4 = expr1, expr2
```

6.16. 运算符优先级

下表对 Python 中运算符的优先顺序进行了总结，从最低优先级（最后绑定）到最高优先级（最先绑定）。 相同单元格内的运算符具有相同优先级。 除非句法显式地给出，否则运算符均指二元运算。 相同单元格内的运算符均从左至右分组（除了幂运算是从右至左分组）。

请注意比较、成员检测和标识号检测均为相同优先级，并具有如 [比较运算](#) 一节所描述的从左至右串连特性。

运算符	描述
<code>:=</code>	赋值表达式
<code>lambda</code>	lambda 表达式
<code>if</code> — <code>else</code>	条件表达式
<code>or</code>	布尔逻辑或 OR
<code>and</code>	布尔逻辑与 AND
<code>not</code> <code>x</code>	布尔逻辑非 NOT
<code>in</code> , <code>not in</code> , <code>is</code> , <code>is not</code> , <code><</code> , <code><=</code> , <code>></code> , <code>>=</code> , <code>!=</code> , <code>==</code>	比较运算，包括成员检测和标识号检测
<code> </code>	按位或 OR
<code>^</code>	按位异或 XOR
<code>&</code>	按位与 AND
<code><<</code> , <code>>></code>	移位
<code>+</code> , <code>-</code>	加和减
<code>,</code> , <code>@</code> , <code>/</code> , <code>//</code> , <code>%</code>	乘，矩阵乘，除，整除，取余 5
<code>+x</code> , <code>-x</code> , <code>~x</code>	正，负，按位非 NOT
<code>*</code>	乘方 6
<code>await</code> <code>x</code>	await 表达式
<code>x[index]</code> , <code>x[index:index]</code> , <code>x(arguments...)</code> , <code>x.attribute</code>	抽取，切片，调用，属性引用
<code>(expressions...)</code> , <code>[expressions...]</code> , <code>{key: value...}</code> , <code>{expressions...}</code>	绑定或加圆括号的表达式，列表显示，字典显示，集合显示

脚注

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

- 如果 x 恰好非常接近于 y 的整数倍，则由于舍入的存在 `x//y` 可能会比 `(x-x%y)//y` 大。在这种情况下，Python 会返回后一个结果，以便保持令 `divmod(x,y)[0] * y + x % y` 尽量接近 `x`。
- 3
- Unicode 标准明确区分 码位（例如 U+0041）和 抽象字符（例如 "大写拉丁字母 A"）。虽然 Unicode 中的大多数抽象字符都只用一个码位来代表，但也存在一些抽象字符可使用由多个码位组成的序列来表示。例如，抽象字符 "带有下加符的大写拉丁字母 C" 可以用 U+00C7 码位上的单个 预设字符 来表示，也可以用一个 U+0043 码位上的 基础字符（大写拉丁字母 C）加上一个 U+0327 码位上的 组合字符（组合下加符）组成的序列来表示。

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

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

- 4
- 由于存在自动垃圾收集、空闲列表以及描述器的动态特性，你可能会注意到在特定情况下使用 `is` 运算符会出现看似不正常的行为，例如涉及到实例方法或常量之间的比较时就是如此。更多信息请查看有关它们的文档。
- 5
- `%` 运算符也被用于字符串格式化；在此场合下会使用同样的优先级。
- 6
- 幂运算符 `**` 绑定的紧密程度低于在其右侧的算术或按位一元运算符，也就是说 `2 ** -1` 为 `0.5`。

7. 简单语句

简单语句由一个单独的逻辑行构成。 多条简单语句可以存在于同一行内并以分号分隔。 简单语句的句法为：

```
1. simple_stmt ::= expression_stmt
2.                | assert_stmt
3.                | assignment_stmt
4.                | augmented_assignment_stmt
5.                | annotated_assignment_stmt
6.                | pass_stmt
7.                | del_stmt
8.                | return_stmt
9.                | yield_stmt
10.               | raise_stmt
11.               | break_stmt
12.               | continue_stmt
13.               | import_stmt
14.               | future_stmt
15.               | global_stmt
16.               | nonlocal_stmt
```

7.1. 表达式语句

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

```
1. expression_stmt ::= starred_expression
```

表达式语句会对指定的表达式列表（也可能为单一表达式）进行求值。

在交互模式下，如果结果值不为 `None`，它会通过内置的 `repr()` 函数转换为一个字符串，该结果字符串将以单独一行的形式写入标准输出（例外情况是如果结果为 `None`，则该过程调用不产生任何输出。）

7.2. 赋值语句

赋值语句用于将名称（重）绑定到特定值，以及修改属性或可变对象的成员项：

```

1. assignment_stmt ::= (target_list "=") + (starred_expression | yield_expression)
2. target_list     ::= target ("," target)* [","]
3. target          ::= identifier
4.                  | "(" [target_list] ")"
5.                  | "[" [target_list] "]"
6.                  | attributeref
7.                  | subscription
8.                  | slicing
9.                  | "*" target

```

（请参阅 [原型](#) 一节了解 属性引用、抽取 和 切片 的句法定义。）

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

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

对象赋值的目标对象可以包含于圆括号或方括号内，具体操作按以下方式递归地定义。

- 如果目标列表为后面不带逗号、可以包含于圆括号内的单一目标，则将对象赋值给该目标。
- 否则：该对象必须为具有与目标列表相同项数的可迭代对象，这些项将按从左至右的顺序被赋值给对应的目标。
 - 如果目标列表包含一个带有星号前缀的目标，这称为“加星”目标：则该对象至少必须为与目标列表项数减一相同项数的可迭代对象。该可迭代对象前面的项将按从左至右的顺序被赋值给加星目标之前的目标。该可迭代对象末尾的项将被赋值给加星目标之后的目标。然后该可迭代对象中剩余项的列表将被赋值给加星目标（该列表可以为空）。
 - 否则：该对象必须为具有与目标列表相同项数的可迭代对象，这些项将按从左至右的顺序被赋值给对应的目标。

对象赋值给单个目标的操作按以下方式递归地定义。

- 如果目标为标识符（名称）：
 - 如果该名称未出现于当前代码块的 `global` 或 `nonlocal` 语句中：该名称将被绑定到当前局部命名空间的对象。
 - 否则：该名称将被分别绑定到全局命名空间或由 `nonlocal` 所确定的外层命名空间的对象。

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

- 如果该对象为属性引用：引用中的原型表达式会被求值。 它应该产生一个具有可赋值属性的对象；否则将引发 `TypeError`。 该对象会被要求将可赋值对象赋值给指定的属性；如果它无法执行赋值，则会引发异常（通常为 `AttributeError` 但并不强制要求）。

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

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

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

- 如果目标为一个抽取项：引用中的原型表达式会被求值。 它应当产生一个可变序列对象（例如列表）或一个映射对象（例如字典）。 接下来，该抽取表达式会被求值。

如果原型为一个可变序列对象（例如列表），抽取应产生一个整数。 如其为负值，则再加上序列长度。 结果值必须为一个小于序列长度的非负整数，序列将把被赋值对象赋值给该整数指定索引号的项。 如果索引超出范围，将会引发 `IndexError`（给被抽取序列赋值不能向列表添加新项）。

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

对于用户定义对象，会调用 `setitem()` 方法并附带适当的参数。

- 如果目标为一个切片：引用中的原型表达式会被求值。 它应当产生一个可变序列对象（例如列表）。 被赋值对象应当是一个相同类型的序列对象。 接下来，下界与上界表达式如果存在的话将被求值；默认值分别为零和序列长度。 上下边界值应当为整数。 如果某一边界为负值，则会加上序列长度。 求出的边界会被裁剪至介于零和序列长度的开区间中。 最后，将要求序列对象以被赋值序列的项替换该切片。 切片的长度可能与被赋值序列的长度不同，这会在目标序列允许的情况下改变目标序列的长度。

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

虽然赋值的定义意味着左手边与右手边的重叠是“同时”进行的（例如 `a, b = b, a` 会交换两个变量的值），但在赋值给变量的多项集 之内 的重叠是从左至右进行的，这有时会令人混淆。 例如，以下程序将会打印出 `[0, 2]`：

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

参见

- [PEP 3132](#) - 扩展的可迭代对象拆包

- 对 `*target` 特性的规范说明。

7.2.1. 增强赋值语句

增强赋值语句就是在单个语句中将二元运算和赋值语句合为一体：

```
1. augmented_assignment_stmt ::= augtarget augop (expression_list | yield_expression)
2. augtarget                 ::= identifier | attributeref | subscription | slicing
3. augop                     ::= "+=" | "-=" | "*=" | "@=" | "/=" | "//=" | "%=" | "**="
4.                           | ">>=" | "<<=" | "&=" | "^=" | "|="
```

（请参阅 [原型](#) 一节了解最后三种符号的句法定义。）

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

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

不同于普通赋值，增强赋值会在对右边求值之前对左边求值。例如，`a[i] += f(x)` 首先查找 `a[i]`，然后对 `f(x)` 求值并执行加法操作，最后将结果写回到 `a[i]`。

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

对于属性引用类目标，针对常规赋值的 [关于类和实例属性的警告](#) 也同样适用。

7.2.2. 带标注的赋值语句

标注 赋值就是在单个语句中将变量或属性标注和可选的赋值语句合为一体：

```
1. annotated_assignment_stmt ::= augtarget ":" expression
2.                           ["=" (starred_expression | yield_expression)]
```

与普通 [赋值语句](#) 的差别在于仅允许单个目标。

对于将简单名称作为赋值目标的情况，如果是在类或模块作用域中，标注会被求值并存入一个特殊的类或模块属性 `annotations` 中，这是一个将变量名称（如为私有会被移除）映射到被求值标注的字典。此属性为可写并且在类或模块体开始执行时如果静态地发现标注就会自动创建。

对于将表达式作为赋值目标的情况，如果是在类或模块作用域中，标注会被求值，但不会保存。

如果一个名称在函数作用域内被标注，则该名称为该作用域的局部变量。标注绝不在函数作用域内被求值和保存。

如果存在右边，带标注的赋值会在对标注求值之前（如果适用）执行实际的赋值。如果用作表达式目标的右边不存在，则解释器会对目标求值，但最后的 `setitem()` 或 `setattr()` 调用除外。

参见

- [PEP 526](#) - 变量标注的语法
- 该提议增加了标注变量（也包括类变量和实例变量）类型的语法，而不再是通过注释来进行表达。
- [PEP 484](#) - 类型提示
- 该提议增加了 `typing` 模块以便为类型标注提供标准句法，可被静态分析工具和 IDE 所使用。

在 3.8 版更改：现在带有标注的赋值允许在右边以同样的表达式作为常规赋值。 之前，某些表达式（例如未加圆括号的元组表达式）会导致语法错误。

7.3. assert 语句

assert 语句是在程序中插入调试性断言的简便方式：

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

简单形式 `assert expression` 等价于

```
1. if __debug__:  
2.     if not expression: raise AssertionError
```

扩展形式 `assert expression1, expression2` 等价于

```
1. if __debug__:  
2.     if not expression1: raise AssertionError(expression2)
```

以上等价形式假定 `debug` 和 `AssertionError` 指向具有指定名称的内置变量。在当前实现中，内置变量 `debug` 在正常情况下为 `True`，在请求优化时为 `False`（对应命令行选项为 `-O`）。如果在编译时请求优化，当前代码生成器不会为 `assert` 语句发出任何代码。请注意不必在错误信息中包含失败表达式的源代码；它会被作为栈追踪的一部分被显示。

赋值给 `debug` 是非法的。该内置变量的值会在解释器启动时确定。

7.4. pass 语句

```
1. pass_stmt ::= "pass"
```

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

```
1. def f(arg): pass    # a function that does nothing (yet)
2.
3. class C: pass       # a class with no methods (yet)
```

7.5. del 语句

```
1. del_stmt ::= "del" target_list
```

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

目标列表的删除将从左至右递归地删除每一个目标。

名称的删除将从局部或全局命名空间中移除该名称的绑定，具体作用域的确定是看该名称是否有在同一代码块的

`global` 语句中出现。 如果该名称未被绑定，将会引发 `NameError` 。

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

在 3.2 版更改：在之前版本中，如果一个名称作为被嵌套代码块中的自由变量出现，则将其从局部命名空间中删除是非法的。

7.6. return 语句

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

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

如果提供了表达式列表，它将被求值，否则以 **None** 替代。

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

当 **return** 将控制流传出一个带有 **finally** 子句的 **try** 语句时，该 **finally** 子句会先被执行然后再真正离开该函数。

在一个生成器函数中，**return** 语句表示生成器已完成并将导致 **StopIteration** 被引发。返回值（如果有的话）会被当作一个参数用来构建 **StopIteration** 并成为 **StopIteration.value** 属性。

在一个异步生成器函数中，一个空的 **return** 语句表示异步生成器已完成并将导致 **StopAsyncIteration** 被引发。一个非空的 **return** 语句在异步生成器函数中会导致语法错误。

7.7. yield 语句

```
1. yield_stmt ::= yield_expression
```

`yield` 语句在语义上等同于 `yield 表达式`。 `yield` 语句可用来省略在使用等效的 `yield` 表达式语句时所必须的圆括号。 例如，以下 `yield` 语句

```
1. yield <expr>
2. yield from <expr>
```

等同于以下 `yield` 表达式语句

```
1. (yield <expr>)
2. (yield from <expr>)
```

`yield` 表达式和语句仅在定义 `generator` 函数时使用，并且仅被用于生成器函数的函数体内部。 在函数定义中使用 `yield` 就足以使得该定义创建的是生成器函数而非普通函数。

有关 `yield` 语义的完整细节请参看 `yield 表达式` 一节。

7.8. raise 语句

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

如果不带表达式，`raise` 会重新引发当前作用域内最后一个激活的异常。如果当前作用域内没有激活的异常，将会引发 `RuntimeError` 来提示错误。

否则的话，`raise` 会将第一个表达式求值为异常对象。它必须为 `BaseException` 的子类或实例。如果它是一个类，当需要时会通过不带参数地实例化该类来获得异常的实例。

异常的 `类型` 为异常实例的类，`值` 为实例本身。

当异常被引发时通常会自动创建一个回溯对象并将其关联到可写的 `traceback` 属性。你可以创建一个异常并同时使用 `with_traceback()` 异常方法（该方法将返回同一异常实例，并将回溯对象设为其参数）设置自己的回溯，就像这样：

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

`from` 子句用于异常串连：如果有该子句，则第二个 `表达式` 必须为另一个异常或实例，它将作为可写的 `cause` 属性被关联到所引发的异常。如果引发的异常未被处理，两个异常都将被打印出来：

```
1. >>> try:
2. ...     print(1 / 0)
3. ... except Exception as exc:
4. ...     raise RuntimeError("Something bad happened") from exc
5. ...
6. Traceback (most recent call last):
7.   File "<stdin>", line 2, in <module>
8. ZeroDivisionError: division by zero
9.
10. The above exception was the direct cause of the following exception:
11.
12. Traceback (most recent call last):
13.   File "<stdin>", line 4, in <module>
14. RuntimeError: Something bad happened
```

如果一个异常在异常处理器或 `finally` `clause`：中被引发，类似的机制会隐式地发挥作用，之前的异常将被关联到新异常的 `context` 属性：

```
1. >>> try:
2. ...     print(1 / 0)
3. ... except:
4. ...     raise RuntimeError("Something bad happened")
5. ...
6. Traceback (most recent call last):
7.   File "<stdin>", line 2, in <module>
8. ZeroDivisionError: division by zero
```

```

9.
10. During handling of the above exception, another exception occurred:
11.
12. Traceback (most recent call last):
13.   File "<stdin>", line 4, in <module>
14. RuntimeError: Something bad happened

```

异常串连可通过在 `from` 子句中指定 `None` 来显式地加以抑制：

```

1. >>> try:
2.     ...     print(1 / 0)
3. ... except:
4.     ...     raise RuntimeError("Something bad happened") from None
5. ...
6. Traceback (most recent call last):
7.   File "<stdin>", line 4, in <module>
8. RuntimeError: Something bad happened

```

有关异常的更多信息可在 [异常](#) 一节查看，有关处理异常的信息可在 [try 语句](#) 一节查看。

在 3.3 版更改：`None` 现在允许被用作 `raise X from Y` 中的 `Y`。

3.3 新版功能：使用 `suppress_context` 属性来抑制异常上下文的自动显示。

7.9. break 语句

```
1. break_stmt ::= "break"
```

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

它会终结最近的外层循环，如果循环有可选的 `else` 子句，也会跳过该子句。

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

当 `break` 将控制流传出一个带有 `finally` 子句的 `try` 语句时，该 `finally` 子句会先被执行然后再真正离开该循环。

7.10. continue 语句

```
1. continue_stmt ::= "continue"
```

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

当 `continue` 将控制流传出一个带有 `finally` 子句的 `try` 语句时，该 `finally` 子句会先被执行然后再真正开始循环的下一个轮次。

7.11. import 语句

```

1. import_stmt      ::= "import" module ["as" identifier] ("," module ["as" identifier])*
2.                  | "from" relative_module "import" identifier ["as" identifier]
3.                  ("," identifier ["as" identifier])*
4.                  | "from" relative_module "import" "(" identifier ["as" identifier]
5.                  ("," identifier ["as" identifier])* [","] ")"
6.                  | "from" module "import" "*"
7. module           ::= (identifier ".")* identifier
8. relative_module ::= "."* module | "."+
```

基本的 import 语句（不带 `from` 子句）会分两步执行：

- 查找一个模块，如果有必要还会加载并初始化模块。
- 在局部命名空间中为 `import` 语句发生位置所处的作用域定义一个或多个名称。

当语句包含多个子句（由逗号分隔）时这两个步骤将对每个子句分别执行，如同这些子句被分成独立的 import 语句一样。

第一个步骤即查找和加载模块的详情 [导入系统](#) 一节中有更详细的描述，其中也描述了可被导入的多种类型的包和模块，以及可用于定制导入系统的所有钩子对象。 请注意这一步如果失败，则可能说明模块无法找到，或者是在初始化模块，包括执行模块代码期间发生了错误。

如果成功获取到请求的模块，则可以通过以下三种方式一之在局部命名空间中使用它：

- 如果模块名称之后带有 `as`，则跟在 `as` 之后的名称将直接绑定到所导入的模块。
- 如果没有指定其他名称，且被导入的模块为最高层级模块，则模块的名称将被绑定到局部命名空间作为对所导入模块的引用。
- 如果被导入的模块 不是 最高层级模块，则包含该模块的最高层级包的名称将被绑定到局部命名空间作为对该最高层级包的引用。 所导入的模块必须使用其完整限定名称来访问而不能直接访问。

`from` 形式使用的过程略微繁复一些：

- 查找 `from` 子句中指定的模块，如有必要还会加载并初始化模块；
- 对于 `import` 子句中指定的每个标识符：
 - 检查被导入模块是否有该名称的属性
 - 如果没有，尝试导入具有该名称的子模块，然后再次检查被导入模块是否有该属性
 - 如果未找到该属性，则引发 `ImportError`。
 - 否则的话，将对该值的引用存入局部命名空间，如果有 `as` 子句则使用其指定的名称，否则使用该属性的名称

例如：

```

1. import foo                # foo imported and bound locally
2. import foo.bar.baz        # foo.bar.baz imported, foo bound locally
3. import foo.bar.baz as fbb # foo.bar.baz imported and bound as fbb
4. from foo.bar import baz    # foo.bar.baz imported and bound as baz
5. from foo import attr       # foo imported and foo.attr bound as attr

```

如果标识符列表改为一个星号 (`'*'`)，则在模块中定义的全部公有名称都将按 `import` 语句所在的作用域被绑定到局部命名空间。

一个模块所定义的 公有名称 是由在模块的命名空间中检测一个名为 `__all__` 的变量来确定的；如果有定义，它必须是一个字符串列表，其中的项为该模块所定义或导入的名称。在 `__all__` 中所给出的名称都会被视为公有并且应当存在。如果 `__all__` 没有被定义，则公有名称的集合将包含在模块的命名空间中找到的所有不以下划线字符 (`'_'`) 打头的名称。`__all__` 应当包括整个公有 API。它的目标是避免意外地导出不属于 API 的一部分的项（例如在模块内部被导入和使用的库模块）。

通配符形式的导入 `-- from module import *` 仅在模块层级上被允许。尝试在类或函数定义中使用它将引发 `SyntaxError`。

当指定要导入哪个模块时，你不必指定模块的绝对名称。当一个模块或包被包含在另一个包之中时，可以在同一个最高层级包中进行相对导入，而不必提及包名称。通过在 `from` 之后指定的模块或包中使用前缀点号，你可以在不指定确切名称的情况下指明在当前包层级结构中要上溯多少级。一个前缀点号表示是执行导入的模块所在的当前包，两个点号表示上溯一个包层级。三个点号表示上溯两级，依此类推。因此如果你执行 `from . import mod` 时所处位置为 `pkg` 包内的一个模块，则最终你将导入 `pkg.mod`。如果你执行 `from ..subpkg2 import mod` 时所处位置为 `pkg.subpkg1` 则你将导入 `pkg.subpkg2.mod`。有关相对导入的规范说明包含在 [包相对导入](#) 一节中。

`importlib.import_module()` 被提供用来为动态地确定要导入模块的应用提供支持。

引发一个 [审核事件](#) `import` 附带参数 `module` , `filename` , `sys.path` , `sys.meta_path` , `sys.path_hooks` 。

7.11.1. future 语句

`future` 语句 是一种针对编译器的指令，指明某个特定模块应当使用在特定的未来某个 Python 发行版中成为标准特性的语法或语义。

`future` 语句的目的是使得向在语言中引入了不兼容改变的 Python 未来版本的迁移更为容易。它允许基于每个模块在某种新特性成为标准之前的发行版中使用该特性。

```

1. future_stmt ::= "from" "__future__" "import" feature ["as" identifier]
2.              ("," feature ["as" identifier])*
3.              | "from" "__future__" "import" "(" feature ["as" identifier]
4.              ("," feature ["as" identifier])* [","] ")"
5. feature      ::= identifier

```

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

- 模块的文档字符串（如果存在），
- 注释，

- 空行，以及
- 其他 `future` 语句。

在 Python 3.7 中唯一需要使用 `future` 语句的特性是 `标注`。

`future` 语句所启用的所有历史特性仍然为 Python 3 所认可。其中包括 `absolute_import`，`division`，`generators`，`generator_stop`，`unicode_literals`，`print_function`，`nested_scopes` 和 `with_statement`。它们都已成为冗余项，因为它们总是为已启用状态，保留它们只是为了向后兼容。

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

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

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

相应的运行时语义取决于 `future` 语句所启用的指定特性。

请注意以下语句没有任何特别之处：

```
1. import __future__ [as name]
```

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

在默认情况下，通过对 `Code compiled by calls to the` 内置函数 `exec()` 和 `compile()` 的调用所编译的代码如果出现于一个包含有 `future` 语句的模块 `M` 之中，就会使用 `future` 语句所关联的语法和语义。此行为可以通过 `compile()` 的可选参数加以控制 -- 请参阅该函数的文档以了解详情。

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

参见

- [PEP 236](#) - 回到 `future`
- 有关 `future` 机制的最初提议。

7.12. global 语句

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

`global` 语句是作用于整个当前代码块的声明。它意味着所列出的标识符将被解读为全局变量。要给全局变量赋值不可能不用到 `global` 关键字，不过自由变量也可以指向全局变量而不必声明为全局变量。

在 `global` 语句中列出的名称不得在同一代码块内该 `global` 语句之前的位置中使用。

在 `global` 语句中列出的名称不得被定义为正式形参，也不得出现于 `for` 循环的控制目标、`class` 定义、函数定义、`import` 语句或变量标注之中。

CPython implementation detail: 当前的实现并未强制要求所有的上述限制，但程序不应当滥用这样的自由，因为未来的实现可能会改为强制要求，并静默地改变程序的含义。

程序员注意事项：`global` 是对解析器的指令。它仅对与 `global` 语句同时被解析的代码起作用。特别地，包含在提供给内置 `exec()` 函数字符串或代码对象中的 `global` 语句并不会影响包含该函数调用的代码块，而包含在这种字符串中的代码也不会受到包含该函数调用的代码中的 `global` 语句影响。这同样适用于 `eval()` 和 `compile()` 函数。

7.13. nonlocal 语句

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

`nonlocal` 语句会使得所列出的名称指向之前在最近的包含作用域中绑定的除全局变量以外的变量。 这种功能很重要，因为绑定的默认行为是先搜索局部命名空间。 这个语句允许被封装的代码重新绑定局部作用域以外且非全局（模块）作用域当中的变量。

与 `global` 语句中列出的名称不同，`nonlocal` 语句中列出的名称必须指向之前存在于包含作用域之中的绑定（在这个应当用来创建新绑定的作用域不能被无歧义地确定）。

`nonlocal` 语句中列出的名称不得与之前存在于局部作用域中的绑定相冲突。

参见

- [PEP 3104](#) - 访问外层作用域中的名称
- 有关 `nonlocal` 语句的规范说明。

8. 复合语句

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

`if`，`while` 和 `for` 语句用来实现传统的控制流程构造。`try` 语句为一组语句指定异常处理和/和清理代码，而 `with` 语句允许在一个代码块周围执行初始化和终结化代码。函数和类定义在语法上也属于复合语句。

一条复合语句由一个或多个‘子句’组成。一个子句则包含一个句头和一个‘句体’。特定复合语句的子句头都处于相同的缩进层级。每个子句头以一个作为唯一标识的关键字开始并以一个冒号结束。子句体是由一个子句控制的一组语句。子句体可以是在子句头的冒号之后与其同处一行的一条或由分号分隔的多条简单语句，或者也可以是在其之后缩进的一行或多行语句。只有后一种形式的子句体才能包含嵌套的复合语句；以下形式是不合法的，这主要是因为无法分清某个后续的 `else` 子句应该属于哪个 `if` 子句：

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

还要注意的是在这种情形下分号的绑定比冒号更紧密，因此在以下示例中，所有 `print()` 调用或者都不执行，或者都执行：

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

总结：

```
1. compound_stmt ::= if_stmt
2.                  | while_stmt
3.                  | for_stmt
4.                  | try_stmt
5.                  | with_stmt
6.                  | funcdef
7.                  | classdef
8.                  | async_with_stmt
9.                  | async_for_stmt
10.                 | async_funcdef
11. suite          ::= stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
12. statement      ::= stmt_list NEWLINE | compound_stmt
13. stmt_list      ::= simple_stmt (";" simple_stmt)* [";"]
```

请注意语句总是以 `NEWLINE` 结束，之后可能跟随一个 `DEDENT`。还要注意可选的后续子句总是以一个不能作为语句开头的关键字作为开头，因此不会产生歧义（‘悬空的 `else`’问题在 Python 中是通过要求嵌套的 `if` 语句必须缩进来解决的）。

为了保证清晰，以下各节中语法规则采用将每个子句都放在单独行中的格式。

8.1. if 语句

`if` 语句用于有条件的执行：

```
1. if_stmt ::= "if" expression ":" suite
2.           ("elif" expression ":" suite)*
3.           ["else" ":" suite]
```

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

8.2. while 语句

`while` 语句用于在表达式保持为真的情况下重复地执行：

```
1. while_stmt ::= "while" expression ":" suite
2.             ["else" ":" suite]
```

这将重复地检验表达式，并且如果其值为真就执行第一个子句体；如果表达式值为假（这可能在第一次检验时就发生）则如果 `else` 子句体存在就会被执行并终止循环。

第一个子句体中的 `break` 语句在执行时将终止循环且不执行 `else` 子句体。第一个子句体中的 `continue` 语句在执行时将跳过子句体中的剩余部分并返回检验表达式。

8.3. for 语句

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

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

表达式列表会被求值一次；它应该产生一个可迭代对象。系统将为 `expression_list` 的结果创建一个迭代器，然后将为迭代器所提供的每一项执行一次子句体，具体次序与迭代器的返回顺序一致。每一项会按标准赋值规则（参见[赋值语句](#)）被依次赋值给目标列表，然后子句体将被执行。当所有项被耗尽时（这会在序列为空或迭代器引发 `StopIteration` 异常时立刻发生），`else` 子句的子句体如果存在将会被执行，并终止循环。

第一个子句体中的 `break` 语句在执行时将终止循环且不执行 `else` 子句体。第一个子句体中的 `continue` 语句在执行时将跳过子句体中的剩余部分并转往下一项继续执行，或者在没有下一项时转往 `else` 子句执行。

`for` 循环会对目标列表中的变量进行赋值。这将覆盖之前对这些变量的所有赋值，包括在 `for` 循环体中的赋值：

```
1. for i in range(10):
2.     print(i)
3.     i = 5           # this will not affect the for-loop
4.                   # because i will be overwritten with the next
5.                   # index in the range
```

目标列表中的名称在循环结束时不会被删除，但如果序列为空，则它们根本不会被循环所赋值。提示：内置函数 `range()` 会返回一个可迭代的整数序列，适用于模拟 Pascal 中的 `for i := a to b do` 这种效果；例如 `list(range(3))` 会返回列表 `[0, 1, 2]`。

注解

当序列在循环中被修改时会有一个微妙的问题（这只可能发生于可变序列例如列表中）。会有一个内部计数器被用来跟踪下一个要使用的项，每次迭代都会使计数器递增。当计数器值达到序列长度时循环就会终止。这意味着如果语句体从序列中删除了当前（或之前）的一项，下一项就会被跳过（因为其标号将变成已被处理的当前项的标号）。类似地，如果语句体在序列当前项的前面插入一个新项，当前项会在循环的下一轮中再次被处理。这会导致麻烦的程序错误，避免此问题的办法是对整个序列使用切片来创建一个临时副本，例如

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

8.4. try 语句

try 语句可为一组语句指定异常处理器和/或清理代码：

```
1. try_stmt ::= try1_stmt | try2_stmt
2. try1_stmt ::= "try" ":" suite
3.             ("except" [expression ["as" identifier]] ":" suite)+
4.             ["else" ":" suite]
5.             ["finally" ":" suite]
6. try2_stmt ::= "try" ":" suite
7.             "finally" ":" suite
```

except 子句指定一个或多个异常处理器。当 **try** 子句中未发生异常时，没有异常处理器会被执行。当 **try** 子句中发生异常时，将启动对异常处理器的搜索。此搜索会依次检查 **except** 子句，直至找到与该异常相匹配的子句。如果存在无表达式的 **except** 子句，它必须是最后一个；它将匹配任何异常。对于带有表达式的 **except** 子句，该表达式会被求值，如果结果对象与发生的异常“兼容”则该子句将匹配该异常。一个对象如果是异常对象所属的类或基类，或者是包含有兼容该异常的项的元组则两者就是兼容的。

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

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

当找到一个匹配的 **except** 子句时，该异常将被赋值给该 **except** 子句在 **as** 关键字之后指定的目标，如果存在此关键字的话，并且该 **except** 子句体将被执行。所有 **except** 子句都必须有可执行的子句体。当到达子句体的末尾时，通常会转向整个 **try** 语句之后继续执行。（这意味着如果对于同一异常存在有嵌套的两个处理器，而异常发生于内层处理器的 **try** 子句中，则外层处理器将不会处理该异常。）

当使用 **as** 将目标赋值为一个异常时，它将在 **except** 子句结束时被清除。这就相当于

```
1. except E as N:
2.     foo
```

被转写为

```
1. except E as N:
2.     try:
3.         foo
4.     finally:
5.         del N
```

这意味着异常必须赋值给一个不同的名称才能在 **except** 子句之后引用它。异常会被清除是因为在附加了回溯信息的情况下，它们会形成堆栈帧的循环引用，使得所有局部变量保持存活直到发生下一次垃圾回收。

在一个 **except** 子句体被执行之前，有关异常的详细信息存放在 **sys** 模块中，可通过 **sys.exc_info()** 来访问。**sys.exc_info()** 返回一个 3 元组，由异常类、异常实例和回溯对象组成（参见 [标准类型层级结构](#) 一节），用于在程序中标识异常发生点。当从处理异常的函数返回时 **sys.exc_info()** 的值会恢复为（调用前的）原

值。

如果控制流离开 `try` 子句体时没有引发异常，并且没有执行 `return`，`continue` 或 `break` 语句，可选的 `else` 子句将被执行。`else` 语句中的异常不会由之前的 `except` 子句处理。

如果存在 `finally`，它将指定一个‘清理’处理程序。`try` 子句会被执行，包括任何 `except` 和 `else` 子句。如果在这些子句中发生任何未处理的异常，该异常会被临时保存。`finally` 子句将被执行。如果存在被保存的异常，它会在 `finally` 子句的末尾被重新引发。如果 `finally` 子句引发了另一个异常，被保存的异常会被设为新异常的上下文。如果 `finally` 子句执行了 `return`，`break` 或 `continue` 语句，则被保存的异常会被丢弃：

```
1. >>> def f():
2. ...     try:
3. ...         1/0
4. ...     finally:
5. ...         return 42
6. ...
7. >>> f()
8. 42
```

在 `finally` 子句执行期间，程序不能获取异常信息。

当 `return`，`break` 或 `continue` 语句在一个 `try ... finally` 语句的 `try` 子语句体中被执行时，`finally` 子语句也会‘在离开时’被执行。

函数的返回值是由最后被执行的 `return` 语句所决定的。由于 `finally` 子句总是被执行，因此在 `finally` 子句中被执行的 `return` 语句总是最后被执行的：

```
1. >>> def foo():
2. ...     try:
3. ...         return 'try'
4. ...     finally:
5. ...         return 'finally'
6. ...
7. >>> foo()
8. 'finally'
```

有关异常的更多信息可以在 [异常](#) 一节找到，有关使用 `raise` 语句生成异常的信息可以在 [raise 语句](#) 一节找到。

在 3.8 版更改：在 Python 3.8 之前，`continue` 语句不允许在 `finally` 子句中使用，这是因为具体实现存在一个问题。

8.5. with 语句

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

```
1. with_stmt ::= "with" with_item ("," with_item)* ":" suite
2. with_item ::= expression ["as" target]
```

带有一个“项目”的 `with` 语句的执行过程如下：

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

注解

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

- 执行语句体。
- 发起调用上下文管理器的 `exit()` 方法。如果语句体的退出是由异常导致的，则其类型、值和回溯信息将被作为参数传递给 `exit()`。否则的话，将提供三个 `None` 参数。

如果语句体的退出是由异常导致的，并且来自 `exit()` 方法的返回值为假，则该异常会被重新引发。如果返回值为真，则该异常会被抑制，并会继续执行 `with` 语句之后的语句。

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

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

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

等价于

```
1. with A() as a:
2.     with B() as b:
3.         suite
```

在 3.1 版更改：支持多个上下文表达式。

参见

- [PEP 343](#) - "with" 语句
- Python `with` 语句的规范描述、背景和示例。

8.6. 函数定义

函数定义就是对用户自定义函数的定义（参见 [标准类型层级结构](#) 一节）：

```

1. funcdef ::= [decorators] "def" funcname "(" [parameter_list] ")"
2.          ["->" expression] ":" suite
3. decorators ::= decorator+
4. decorator ::= "@" dotted_name "(" [argument_list [","]] ")" NEWLINE
5. dotted_name ::= identifier ( "." identifier)*
6. parameter_list ::= defparameter ( "," defparameter)* [ "/" ["," [parameter_list_no_posonly]]
7.                  | parameter_list_no_posonly
8. parameter_list_no_posonly ::= defparameter ( "," defparameter)* [ "," [parameter_list_starargs]]
9.                  | parameter_list_starargs
10. parameter_list_starargs ::= "*" [parameter] ( "," defparameter)* [ "," ["**" parameter [","]]]
11.                  | "**" parameter [" ,"]
12. parameter ::= identifier [ ":" expression]
13. defparameter ::= parameter ["=" expression]
14. funcname ::= identifier

```

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

函数定义并不会执行函数体；只有当函数被调用时才会执行此操作。 2

一个函数定义可以被一个或多个 [decorator](#) 表达式所包装。当函数被定义时将在包含该函数定义的作用域中对装饰器表达式求值。求值结果必须是一个可调对象，它会以该函数对象作为唯一参数被发起调用。其返回值将被绑定到函数名称而非函数对象。多个装饰器会以嵌套方式被应用。例如以下代码

```
1. @f1(arg)@f2def func(): pass
```

大致等价于

```

1. def func(): pass
2. func = f1(arg)(f2(func))

```

不同之处在于原始函数并不会被临时绑定到名称 `func`。

当一个或多个 [形参](#) 具有 [形参](#) `=` 表达式 这样的形式时，该函数就被称为具有“默认形参值”。对于一个具有默认值的形参，其对应的 [argument](#) 可以在调用中被省略，在此情况下会用形参的默认值来替代。如果一个形参具有默认值，后续所有在 `" * "` 之前的形参也必须具有默认值 -- 这个句法限制并未在语法中明确表达。

默认形参值会在执行函数定义时按从左至右的顺序被求值。这意味着当函数被定义时将对表达式求值一次，相同的“预计算”值将在每次调用时被使用。这一点在默认形参为可变对象，例如列表或字典的时候尤其需要重点理解：如果函数修改了该对象（例如向列表添加了一项），则实际上默认值也会被修改。这通常不是人们所预期的。绕过此问题的一个方法是使用 `None` 作为默认值，并在函数体中显式地对其进行测试，例如：

```
1. def whats_on_the_telly(penguin=None):
```

```

2.     if penguin is None:
3.         penguin = []
4.     penguin.append("property of the zoo")
5.     return penguin

```

函数调用的语义在 [调用](#) 一节中有更详细的描述。函数调用总是会给形参列表中列出的所有形参赋值，或用位置参数，或用关键字参数，或用默认值。如果存在 " `identifier` " 这样的形式，它会被初始化为一个元组来接收任何额外的位置参数，默认为空元组。如果存在 " `**identifier` " 这样的形式，它会被初始化为一个新的有序映射来接收任何额外的关键字参数，默认为一个相同类型的空映射。在 " `"` " 或 " `*identifier` " 之后的形参都是仅关键字形参，只能通过关键字参数传入值。

形参可以带有 [标注](#)，其形式为在形参名称后加上 " `: expression` "。任何形参都可以带有标注，甚至 `identifier` 或 `* identifier` 这样的形参也可以。函数可以带有“返回”标注，其形式为在形参列表后加上 " `-> expression` "。这些标注可以是任何有效的 Python 表达式。标注的存在不会改变函数的语义。标注值可以作为函数对象的 `annotations` 属性中以对应形参名称为键的字典值被访问。如果使用了 `annotations` `import from future` 的方式，则标注会在运行时保存为字符串以启用延迟求值特性。否则，它们会在执行函数定义时被求值。在这种情况下，标注的求值顺序可能与它们在源代码中出现的顺序不同。

创建匿名函数（未绑定到一个名称的函数）以便立即在表达式中使用也是可能的。这需要使用 `lambda` 表达式，具体描述见 [lambda 表达式](#) 一节。请注意 `lambda` 只是简单函数定义的一种简化写法；在 " `def` " 语句中定义的函数也可以像用 `lambda` 表达式定义的函数一样被传递或赋值给其他名称。" `def` " 形式实际上更为强大，因为它允许执行多条语句和使用标注。

程序员注意事项：函数属于一类对象。在一个函数内部执行的 " `def` " 语句会定义一个局部函数并可被返回或传递。在嵌套函数中使用的自由变量可以访问包含该 `def` 语句的函数的局部变量。详情参见 [命名与绑定](#) 一节。

参见

- [PEP 3107](#) - 函数标注
- 最初的函数标注规范说明。
- [PEP 484](#) - 类型提示
- 标注的标准含意定义：类型提示。
- [PEP 526](#) - 变量标注的语法
- 变量声明的类型提示功能，包括类变量和实例变量
- [PEP 563](#) - 延迟的标注求值
- 支持在运行时通过以字符串形式保存标注而非不是即求值来实现标注内部的向前引用。

8.7. 类定义

类定义就是对类对象的定义（参见 [标准类型层级结构](#) 一节）：

```
1. classdef ::= [decorators] "class" classname [inheritance] ":" suite
2. inheritance ::= "(" [argument_list] ")"
3. classname ::= identifier
```

类定义是一条可执行语句。其中继承列表通常给出基类的列表（进阶用法请参见 [元类](#)），列表中的每一项都应当被求值为一个允许子类的类对象。没有继承列表的类默认继承自基类 `object`；因此，：

```
1. class Foo:
2.     pass
```

等价于

```
1. class Foo(object):
2.     pass
```

随后类体将在一个新的执行帧（参见 [命名与绑定](#)）中被执行，使用新创建的局部命名空间和原有的全局命名空间。（通常，类体主要包含函数定义。）当类体结束执行时，其执行帧将被丢弃而其局部命名空间会被保存。³ 一个类对象随后会被创建，其基类使用给定的继承列表，属性字典使用保存的局部命名空间。类名称将在原有的全局命名空间中绑定到该类对象。

在类体内定义的属性的顺序保存在新类的 `dict` 中。请注意此顺序的可靠性只限于类刚被创建时，并且只适用于使用定义语法所定义的类。

类的创建可使用 [元类](#) 进行重度定制。

类也可以被装饰：就像装饰函数一样，：

```
1. @f1(arg)@f2class Foo: pass
```

大致等价于

```
1. class Foo: pass
2. Foo = f1(arg)(f2(Foo))
```

装饰器表达式的求值规则与函数装饰器相同。结果随后会被绑定到类名称。

程序员注意事项：在类定义内定义的变量是类属性；它们将被类实例所共享。实例属性可通过 `self.name = value` 在方法中设定。类和实例属性均可通过 "`self.name`" 表示法来访问，当通过此方式访问时实例属性会隐藏同名的类属性。类属性可被用作实例属性的默认值，但在此场景下使用可变值可能导致未预期的结果。可以使用 [描述器](#) 来创建具有不同实现细节的实例变量。

参见

- [PEP 3115](#) - Python 3000 中的元类
- 将元类声明修改为当前语法的提议，以及关于如何构建带有元类的类的语义描述。
- [PEP 3129](#) - 类装饰器
- 增加类装饰器的提议。 函数和方法装饰器是在 [PEP 318](#) 中被引入的。

8.8. 协程

3.5 新版功能.

8.8.1. 协程函数定义

```
1. async_funcdef ::= [decorators] "async" "def" funcname "(" [parameter_list] ")"
2.                               ["->" expression] ":" suite
```

Python 协程可以在多个位置上挂起和恢复执行（参见 [coroutine](#)）。在协程函数体内部，`await` 和 `async` 标识符已成为保留关键字；`await` 表达式，`async for` 以及 `async with` 只能在协程函数体中使用。

使用 `async def` 语法定义的函数总是为协程函数，即使它们不包含 `await` 或 `async` 关键字。

在协程函数体中使用 `yield from` 表达式将引发 `SyntaxError`。

协程函数的例子：

```
1. async def func(param1, param2):
2.     do_stuff()
3.     await some_coroutine()
```

8.8.2. async for 语句

```
1. async_for_stmt ::= "async" for_stmt
```

[asynchronous iterable](#) 能够在其 `iter` 实现中调用异步代码，而 [asynchronous iterator](#) 可以在其 `next` 方法中调用异步代码。

`async for` 语句允许方便地对异步迭代器进行迭代。

以下代码：

```
1. async for TARGET in ITER:
2.     BLOCK
3. else:
4.     BLOCK2
```

在语义上等价于：

```
1. iter = (ITER)
2. iter = type(iter).__aiter__(iter)
3. running = True
4. while running:
5.     try:
6.         TARGET = await type(iter).__anext__(iter)
```

```

7.     except StopAsyncIteration:
8.         running = False
9.     else:
10.        BLOCK
11. else:
12.     BLOCK2

```

另请参阅 `aiter()` 和 `anext()` 了解详情。

在协程函数体之外使用 `async for` 语句将引发 `SyntaxError`。

8.8.3. async with 语句

```
1. async_with_stmt ::= "async" with_stmt
```

`asynchronous context manager` 是一种 `context manager`，能够在其 `enter` 和 `exit` 方法中暂停执行。

以下代码：

```

1. async with EXPR as VAR:
2.     BLOCK

```

在语义上等价于：

```

1. mgr = (EXPR)
2. aexit = type(mgr).__aexit__
3. aenter = type(mgr).__aenter__(mgr)
4.
5. VAR = await aenter
6. try:
7.     BLOCK
8. except:
9.     if not await aexit(mgr, *sys.exc_info()):
10.         raise
11. else:
12.     await aexit(mgr, None, None, None)

```

另请参阅 `aenter()` 和 `aexit()` 了解详情。

在协程函数体之外使用 `async with` 语句将引发 `SyntaxError`。

参见

- [PEP 492](#) - 使用 `async` 和 `await` 语法实现协程
- 将协程作为 Python 中的一个正式的单独概念，并增加相应的支持语法。

脚注

- [1](#)

- 异常会被传播给发起调用栈，除非存在一个 `finally` 子句正好引发了另一个异常。新引发的异常将导致旧异常的丢失。
- 2
- 作为函数体的第一条语句出现的字符串字面值会被转换为函数的 `doc` 属性，也就是该函数的 `docstring`。
- 3
- 作为类体的第一条语句出现的字符串字面值会被转换为命名空间的 `doc` 条目，也就是该类的 `docstring`。

9. 最高层级组件

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

- [9.1. 完整的 Python 程序](#)
- [9.2. 文件输入](#)
- [9.3. 交互式输入](#)
- [9.4. 表达式输入](#)

9.1. 完整的 Python 程序

虽然语言规范描述不必规定如何发起调用语言解释器，但对完整的 Python 程序加以说明还是很有用的。一个完整的 Python 程序会在最小初始化环境中被执行：所有内置和标准模块均为可用，但均处于未初始化状态，只有 `sys`（各种系统服务），`builtins`（内置函数、异常以及 `None`）和 `main` 除外。最后一个模块用于为完整程序的执行提供局部和全局命名空间。

适用于一个完整 Python 程序的语法即下节所描述的文件输入。

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

一个完整程序可通过三种形式被传递给解释器：使用 `-c` 字符串 命令行选项，使用一个文件作为第一个命令行参数，或者使用标准输入。如果文件或标准输入是一个 tty 设置，解释器会进入交互模式；否则的话，它会将文件当作一个完整程序来执行。

9.2. 文件输入

所有从非交互式文件读取的输入都具有相同的形式：

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

此语法用于下列几种情况：

- 解析一个完整 Python 程序时（从文件或字符串）；
- 解析一个模块时；
- 解析一个传递给 `exec()` 函数的字符串时；

9.3. 交互式输入

交互模式下的输入使用以下语法进行解析：

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

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

9.4. 表达式输入

`eval()` 被用于表达式输入。 它会忽略开头的空白。 传递给 `eval()` 的字符串参数必须具有以下形式：

```
1. eval_input ::= expression_list NEWLINE*
```

10. 完整的语法规范

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

```

1. # Grammar for Python
2.
3. # NOTE WELL: You should also follow all the steps listed at
4. # https://devguide.python.org/grammar/
5.
6. # Start symbols for the grammar:
7. #     single_input is a single interactive statement;
8. #     file_input is a module or sequence of commands read from an input file;
9. #     eval_input is the input for the eval() functions.
10. #     func_type_input is a PEP 484 Python 2 function type comment
11. # NB: compound_stmt in single_input is followed by extra NEWLINE!
12. # NB: due to the way TYPE_COMMENT is tokenized it will always be followed by a NEWLINE
13. single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
14. file_input: (NEWLINE | stmt)* ENDMARKER
15. eval_input: testlist NEWLINE* ENDMARKER
16.
17. decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
18. decorators: decorator+
19. decorated: decorators (classdef | funcdef | async_funcdef)
20.
21. async_funcdef: ASYNC funcdef
22. funcdef: 'def' NAME parameters ['->' test] ':' [TYPE_COMMENT] func_body_suite
23.
24. parameters: '(' [typedarglist] ')'
25.
26. # The following definition for typedarglist is equivalent to this set of rules:
27. #
28. #     arguments = argument (',' [TYPE_COMMENT] argument)*
29. #     argument = tfpdef ['=' test]
30. #     kwargs = '*' tfpdef [','] [TYPE_COMMENT]
31. #     args = '*' [tfpdef]
32. #     kwnonly_kwargs = (',' [TYPE_COMMENT] argument)* (TYPE_COMMENT | [',' [TYPE_COMMENT] [kwargs]])
33. #     args_kwnonly_kwargs = args kwnonly_kwargs | kwargs
34. #     poskeyword_args_kwnonly_kwargs = arguments ( TYPE_COMMENT | [',' [TYPE_COMMENT] [args_kwnonly_kwargs]])
35. #     typedarglist_no_posonly = poskeyword_args_kwnonly_kwargs | args_kwnonly_kwargs
36. #     typedarglist = (arguments ',' [TYPE_COMMENT] '/' [',' [[TYPE_COMMENT] typedarglist_no_posonly]])|
37. #                     (typedarglist_no_posonly)"
38. # It needs to be fully expanded to allow our LL(1) parser to work on it.
39.
40. typedarglist: (
41.     (tfpdef ['=' test] (',' [TYPE_COMMENT] tfpdef ['=' test])* ',' [TYPE_COMMENT] '/' [',' [ [TYPE_COMMENT]
42.         tfpdef ['=' test] (
43.             ',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',' [TYPE_COMMENT] [
44.                 '*' [tfpdef] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',' [TYPE_COMMENT] ['**' tfpdef
45.                     [','] [TYPE_COMMENT]]))
46.                 | '*' tfpdef [','] [TYPE_COMMENT]]))
47.         | '*' [tfpdef] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',' [TYPE_COMMENT] ['**' tfpdef

```

```

    [' ' [TYPE_COMMENT]]])
46. | '*' tfpdef [' ' [TYPE_COMMENT]] )
47. | (tfpdef ['=' test] (' ' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [' ' [TYPE_COMMENT] [
48.   '*' [tfpdef] (' ' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [' ' [TYPE_COMMENT] ['*' tfpdef [' '
    [TYPE_COMMENT]]])
49. | '*' tfpdef [' ' [TYPE_COMMENT]]])
50. | '*' [tfpdef] (' ' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [' ' [TYPE_COMMENT] ['*' tfpdef
    [' ' [TYPE_COMMENT]]])
51. | '*' tfpdef [' ' [TYPE_COMMENT])
52. )
53. tfpdef: NAME [':' test]
54.
55. # The following definition for vararglist is equivalent to this set of rules:
56. #
57. #   arguments = argument (' ' argument )*
58. #   argument = vfpdef ['=' test]
59. #   kwargs = '*' vfpdef [' ' ]
60. #   args = '*' [vfpdef]
61. #   kwonly_kwargs = (' ' argument )* [' ' [kwargs]]
62. #   args_kwonly_kwargs = args kwonly_kwargs | kwargs
63. #   poskeyword_args_kwonly_kwargs = arguments [' ' [args_kwonly_kwargs]]
64. #   vararglist_no_posonly = poskeyword_args_kwonly_kwargs | args_kwonly_kwargs
65. #   vararglist = arguments ' ' '/' [' ' [(vararglist_no_posonly)] | (vararglist_no_posonly)
66. #
67. # It needs to be fully expanded to allow our LL(1) parser to work on it.
68.
69. vararglist: vfpdef ['=' test ]([' ' vfpdef ['=' test])* ' ' '/' [' ' [ (vfpdef ['=' test] (' ' vfpdef ['='
    test])* [' ' [
70.   '*' [vfpdef] (' ' vfpdef ['=' test])* [' ' ['*' vfpdef [' ' ]]]
71.   | '*' vfpdef [' ' ]]]
72.   | '*' [vfpdef] (' ' vfpdef ['=' test])* [' ' ['*' vfpdef [' ' ]]]
73.   | '*' vfpdef [' ' ] ] ] | (vfpdef ['=' test] (' ' vfpdef ['=' test])* [' ' [
74.   '*' [vfpdef] (' ' vfpdef ['=' test])* [' ' ['*' vfpdef [' ' ]]]
75.   | '*' vfpdef [' ' ]]]
76.   | '*' [vfpdef] (' ' vfpdef ['=' test])* [' ' ['*' vfpdef [' ' ]]]
77.   | '*' vfpdef [' ' ]
78. )
79. vfpdef: NAME
80.
81. stmt: simple_stmt | compound_stmt
82. simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
83. small_stmt: (expr_stmt | del_stmt | pass_stmt | flow_stmt |
84.   import_stmt | global_stmt | nonlocal_stmt | assert_stmt)
85. expr_stmt: testlist_star_expr (annassign | augassign (yield_expr|testlist) |
86.   [('=' (yield_expr|testlist_star_expr))+ [TYPE_COMMENT]] )
87. annassign: ':' test ['=' (yield_expr|testlist_star_expr)]
88. testlist_star_expr: (test|star_expr) (' ' (test|star_expr))* [' ' ]
89. augassign: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' | '&=' | '|=' | '^=' |
90.   '<<=' | '>>=' | '**=' | '//=')
91. # For normal and annotated assignments, additional restrictions enforced by the interpreter
92. del_stmt: 'del' exprlist
93. pass_stmt: 'pass'
94. flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
95. break_stmt: 'break'
96. continue_stmt: 'continue'

```

```

97. return_stmt: 'return' [testlist_star_expr]
98. yield_stmt: yield_expr
99. raise_stmt: 'raise' [test ['from' test]]
100. import_stmt: import_name | import_from
101. import_name: 'import' dotted_as_names
102. # note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
103. import_from: ('from' (('.' | '...')* dotted_name | ('.' | '...')+
104.             'import' ('*' | '(' import_as_names ')' | import_as_names))
105. import_as_name: NAME ['as' NAME]
106. dotted_as_name: dotted_name ['as' NAME]
107. import_as_names: import_as_name (',' import_as_name)* [',' ]
108. dotted_as_names: dotted_as_name (',' dotted_as_name)*
109. dotted_name: NAME ( '.' NAME)*
110. global_stmt: 'global' NAME (',' NAME)*
111. nonlocal_stmt: 'nonlocal' NAME (',' NAME)*
112. assert_stmt: 'assert' test [',' test]
113.
114. compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef | classdef | decorated |
    async_stmt
115. async_stmt: ASYNC (funcdef | with_stmt | for_stmt)
116. if_stmt: 'if' namedexpr_test ':' suite ('elif' namedexpr_test ':' suite)* ['else' ':' suite]
117. while_stmt: 'while' namedexpr_test ':' suite ['else' ':' suite]
118. for_stmt: 'for' exprlist 'in' testlist ':' [TYPE_COMMENT] suite ['else' ':' suite]
119. try_stmt: ('try' ':' suite
120.           ((except_clause ':' suite)+
121.            ['else' ':' suite]
122.            ['finally' ':' suite] |
123.            'finally' ':' suite))
124. with_stmt: 'with' with_item (',' with_item)* ':' [TYPE_COMMENT] suite
125. with_item: test ['as' expr]
126. # NB compile.c makes sure that the default except clause is last
127. except_clause: 'except' [test ['as' NAME]]
128. suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT
129.
130. namedexpr_test: test [':=' test]
131. test: or_test ['if' or_test 'else' test] | lambdef
132. test_nocond: or_test | lambdef_nocond
133. lambdef: 'lambda' [varargslist] ':' test
134. lambdef_nocond: 'lambda' [varargslist] ':' test_nocond
135. or_test: and_test ('or' and_test)*
136. and_test: not_test ('and' not_test)*
137. not_test: 'not' not_test | comparison
138. comparison: expr (comp_op expr)*
139. # <> isn't actually a valid comparison operator in Python. It's here for the
140. # sake of a __future__ import described in PEP 401 (which really works :-))
141. comp_op: '<' | '>' | '==' | '>=' | '<=' | '<>' | '!=' | 'in' | 'not in' | 'is' | 'is not'
142. star_expr: '*' expr
143. expr: xor_expr ('|' xor_expr)*
144. xor_expr: and_expr ('^' and_expr)*
145. and_expr: shift_expr ('&' shift_expr)*
146. shift_expr: arith_expr (('<<' | '>>') arith_expr)*
147. arith_expr: term (('+' | '-') term)*
148. term: factor (('*' | '@' | '/' | '%' | '//') factor)*
149. factor: ('+' | '-' | '~') factor | power

```

```

150. power: atom_expr ['**' factor]
151. atom_expr: [AWAIT] atom trailer*
152. atom: '(' [yield_expr|testlist_comp] ')' |
153.        '[' [testlist_comp] ']' |
154.        '{' [dictorsetmaker] '}' |
155.        NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False')
156. testlist_comp: (namedexpr_test|star_expr) ( comp_for | (',' (namedexpr_test|star_expr))* [','] )
157. trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
158. subscriptlist: subscript (',' subscript)* [',']
159. subscript: test | [test] ':' [test] [sliceop]
160. sliceop: ':' [test]
161. exprlist: (expr|star_expr) (',' (expr|star_expr))* [',']
162. testlist: test (',' test)* [',']
163. dictorsetmaker: ( ((test ':' test | '**' expr)
164.                    (comp_for | (',' (test ':' test | '**' expr))* [','])) |
165.                    ((test | star_expr)
166.                     (comp_for | (',' (test | star_expr))* [',']))) )
167.
168. classdef: 'class' NAME ['(' [arglist] ')'] ':' suite
169.
170. arglist: argument (',' argument)* [',']
171.
172. # The reason that keywords are test nodes instead of NAME is that using NAME
173. # results in an ambiguity. ast.c makes sure it's a NAME.
174. # "test '=' test" is really "keyword '=' test", but we have no such token.
175. # These need to be in a single rule to avoid grammar that is ambiguous
176. # to our LL(1) parser. Even though 'test' includes '*expr' in star_expr,
177. # we explicitly match '*' here, too, to give it proper precedence.
178. # Illegal combinations and orderings are blocked in ast.c:
179. # multiple (test comp_for) arguments are blocked; keyword unpackings
180. # that precede iterable unpackings are blocked; etc.
181. argument: ( test [comp_for] |
182.            test ':' test |
183.            test '=' test |
184.            '**' test |
185.            '*' test )
186.
187. comp_iter: comp_for | comp_if
188. sync_comp_for: 'for' exprlist 'in' or_test [comp_iter]
189. comp_for: [ASYNC] sync_comp_for
190. comp_if: 'if' test_nocond [comp_iter]
191.
192. # not used in grammar, but may appear in "node" passed from Parser to Compiler
193. encoding_decl: NAME
194.
195. yield_expr: 'yield' [yield_arg]
196. yield_arg: 'from' test | testlist_star_expr
197.
198. # the TYPE_COMMENT in suites is only parsed for funcdefs,
199. # but can't go elsewhere due to ambiguity
200. func_body_suite: simple_stmt | NEWLINE [TYPE_COMMENT NEWLINE] INDENT stmt+ DEDENT
201.
202. func_type_input: func_type NEWLINE* ENDMARKER

```

```
203. func_type: '(' [typelist] ')' '->' test
204. # typelist is a modified typedargslist (see above)
205. typelist: (test (',' test)* [',' 
206.             ['*' [test] (',' test)* [',' '*' test] | '*' test]]
207.           | '*' [test] (',' test)* [',' '*' test] | '*' test)
```