7. 简单语句

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

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

7.1. 表达式语句

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

```
expression_stmt: starred_expression
```

表达式语句会对指定的表达式列表（也可能为单一表达式）进行求值。

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

7.2. 赋值语句

赋值语句用于将名称（重）绑定到特定值，以及修改属性或可变对象的成员项:

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

```
                  | slicing
                  | "*" target
```

(请参阅 原型 一节了解 *属性引用*, *抽取* 和 *切片* 的句法定义。)

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

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

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

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

对象赋值给单个目标的操作按以下方式递归地定义。

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

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

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

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

  ```
  class Cls:
      x = 3             # 类变量
  inst = Cls()
  inst.x = inst.x + 1   # 将 inst.x 改为 4 而 Cls.x 仍为 3
  ```

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

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

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

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

  对于用户自定义对象，会调用 __setitem__() 方法并附带适当的参数。

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

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

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

```
x = [0, 1]
i = 0
i, x[i] = 1, 2          # 先更新 i，再更新 x[i]
print(x)
```

参见:

PEP 3132 - 扩展的可迭代对象拆包
对 `*target` 特性的规范说明。

7.2.1. 增强赋值语句

增强赋值语句就是在单个语句中将二元运算和赋值语句合为一体:

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

（请参阅 原型 一节了解最后三种符号的句法定义。）

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

增强赋值语句如 `x += 1` 可以被改写为 `x = x + 1` 以获得类似的、但并非完全等价的效果。 在增强赋值版本中，`x` 仅会被求值一次。 而且，在可能的情况下，实际的运算是 *原地* 执行的，这意味着并不是创建一个新对象并将其赋值给目标，而是直接修改原对象。

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

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

对于属性引用类目标，针对常规赋值的 关于类和实例属性的警告 也同样适用。

7.2.2. 带标注的赋值语句

标注 赋值就是在单个语句中将变量或属性标注和可选的赋值语句合为一体:

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

与普通 赋值语句 的差别在于仅允许单个目标。

如果赋值目标由不带圆括号的单个名称组成则被视为“简单型”。 对于简单型赋值目标，如果处于类或模块作用域中，标注将被收集到一个惰性求值的 标注作用域 中。 这些标注的求值可使用类或模块的 `__annotations__` 属性，或是使用 `annotationlib` 模块中的工具。

如果赋值目标不是简单型的（即属性、下标节点或带圆括号的名称），则标注将永远不会被求值。

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

如果存在右手边，带标注的赋值会执行实际的赋值就像不存在任何标注一样。 如果不存在作为表达式目标的右手边，那么解释器会对目标求值但最后的 `__setitem__()` 或 `__setattr__()` 调用除外。

参见:

PEP 526 - 变量标注的语法

该提议增加了标注变量（也包括类变量和实例变量）类型的语法，而不再是通过注释来进行表达。

PEP 484 - 类型提示

该提议增加了 typing 模块以便为类型标注提供标准句法，可被静态分析工具和 IDE 所使用。

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

在 3.14 版本发生变更: 现在标注会在单独的 标注作用域 中被惰性求值。 如果赋值目标不是简单型的，则标注永远不会被求值。

7.3. assert 语句

assert 语句是在程序中插入调试性断言的简便方式:

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

简单形式 `assert expression` 等价于

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

扩展形式 `assert expression1, expression2` 等价于

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

这些等价形式假定 `__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则当前代码生成器不会为 `assert` 语句发出任何代码。 请注意不需要在错误信息中包括失败的表达式的源代码；它会作为栈回溯的一部分被显示。

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

7.4. pass 语句

```
pass_stmt: "pass"
```

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

```
def f(arg): pass    # 一个（目前）不做任何事的函数

class C: pass       # 一个（目前）没有任何方法的类
```

7.5. del 语句

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

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

目标列表的删除将从左至右递归地删除每一个目标。

删除名称会从局部或全局命名空间中移除该名称的绑定，具体取决于该名称是否出现在同一代码块中的 global 语句中。尝试删除未绑定的名称会引发 NameError 异常。

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

> *在 3.2 版本发生变更:* 在之前版本中，如果一个名称作为被嵌套代码块中的自由变量出现，则将其从局部命名空间中删除是非法的。

7.6. return 语句

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

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

如果提供了表达式列表，它将被求值，否则以 None 替代。

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

当 return 将控制流传出一个带有 finally 子句的 try 语句时，该 finally 子句会先被执行然后再真正离开该函数。

在一个生成器函数中，return 语句表示生成器已完成并将导致 StopIteration 被引发。 返回值（如果有的话）会被当作一个参数用来构建 StopIteration 并成为 StopIteration.value 属性。

在一个异步生成器函数中，一个空的 return 语句表示异步生成器已完成并将导致 StopAsyncIteration 被引发。 一个非空的 return 语句在异步生成器函数中会导致语法错误。

7.7. yield 语句

```
yield_stmt: yield_expression
```

yield 语句在语义上等同于 yield 表达式。 yield 语句可用来省略在使用等效的 yield 表达式语句时所必须的圆括号。 例如，以下 yield 语句

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

等同于以下 yield 表达式语句

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

yield 表达式和语句仅在定义 generator 函数时使用，并且仅被用于生成器函数的函数体内部。 在函数定义中使用 yield 就足以使得该定义创建的是生成器函数而非普通函数。

有关 yield 语义的完整细节请参看 yield 表达式 一节。

7.8. raise 语句

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

如果没有提供表达式，则 raise 会重新引发当前正在处理的异常，它也被称为 *活动的异常*。 如果当前没有活动的异常，则会引发 RuntimeError 来提示发生了错误。

否则的话，raise 会将第一个表达式求值为异常对象。 它必须为 BaseException 的子类或实例。 如果它是一个类，当需要时会通过不带参数地实例化该类来获得异常的实例。

异常的 *类型* 为异常实例的类，*值* 为实例本身。

当有异常被引发时通常会自动创建一个回溯对象并将其关联到它的 __traceback__ 属性。 你可以创建一个异常并使用 with_traceback() 异常方法直接设置你的回溯对象（该方法将返回同一异常实例，并将回溯对象设为其参数），就像这样:

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

from 子句用于异常串连：如果给出该子句，则第二个 *表达式* 必须为另一个异常类或实例。 如果第二个表达式是一个异常实例，它将作为 __cause__ 属性（为一个可写属性）被关联到所引发的异常。 如果该表达式是一个异常类，这个类将被实例化且所生成的异常实例将作为 __cause__ 属性被关联到所引发的异常。 如果所引发的异常未被处理，则两个异常都将被打印:

```
>>> try:
...     print(1 / 0)
... except Exception as exc:
...     raise RuntimeError("Something bad happened") from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(1 / 0)
          ~~^~~
ZeroDivisionError: division by zero

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

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

当已经有一个异常在处理时如果有新的异常被引发则类似的机制会隐式地起作用。 异常可以通过使用 except 或 finally 子句或者 with 语句来处理。 之前的异常将被关联至新异常的 __context__ 属性:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened")
...
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(1 / 0)
          ~~^~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

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

异常串连可通过在 `from` 子句中指定 `None` 来显式地加以抑制：

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened") from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

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

> *在 3.3 版本发生变更:* `None` 现在允许被用作 `raise X from Y` 中的 `Y`。
>
> 增加了 `__suppress_context__` 属性向来抑制异常上下文的自动显示。

> *在 3.11 版本发生变更:* 如果活动异常的回溯在 `except` 子句中被修改，则会有后续的 `raise` 语句重新引发该异常并附带被修改的回溯。 在之前版本中，重新引发该异常则会附带它被捕获时的回溯。

7.9. break 语句

```
break_stmt: "break"
```

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

它会终结最近的外层循环，如果循环有可选的 `else` 子句，也会跳过该子句。

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

当 `break` 将控制流传出一个带有 `finally` 子句的 `try` 语句时，该 `finally` 子句会先被执行然后再真正离开该循环。

7.10. continue 语句

```
continue_stmt: "continue"
```

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

当 `continue` 将控制流传出一个带有 `finally` 子句的 `try` 语句时，该 `finally` 子句会先被执行然后再真正开始循环的下一个轮次。

7.11. `import` 语句

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

基本的 import 语句（不带 `from` 子句）会分两步执行:

1. 查找一个模块，如果有必要还会加载并初始化模块。
2. 在局部命名空间中为 `import` 语句发生位置所处的作用域定义一个或多个名称。

当语句包含多个子句（由逗号分隔）时这两个步骤将对每个子句分别执行，如同这些子句被分成独立的 import 语句一样。

第一个步骤，即查找和加载模块的细节在 导入系统 一节中有更详细的描述，其中也描述了可被导入的多种类型的包和模块，以及可用于定制导入系统的所有钩子对象。 请注意如果这一步失败，则可能说明模块无法找到，*或者* 是在初始化模块，包括执行模块代码期间发生了错误。

如果成功获取到请求的模块，则可以通过以下三种方式一之在局部命名空间中使用它:

- 模块名后使用 `as` 时，直接把 `as` 后的名称与导入模块绑定。
- 如果没有指定其他名称，且被导入的模块为最高层级模块，则模块的名称将被绑定到局部命名空间作为对所导入模块的引用。
- 如果被导入的模块 *不是* 最高层级模块，则包含该模块的最高层级包的名称将被绑定到局部命名空间作为对该最高层级包的引用。 所导入的模块必须使用其完整限定名称来访问而不能直接访问。

`from` 形式使用的过程略微繁复一些:

1. 查找 `from` 子句中指定的模块，如有必要还会加载并初始化模块；
2. 对于 `import` 子句中指定的每个标识符:
    1. 检查被导入模块是否有该名称的属性
    2. 如果没有，尝试导入具有该名称的子模块，然后再次检查被导入模块是否有该属性
    3. 如果未找到该属性，则引发 `ImportError`。
    4. 否则的话，将对该值的引用存入局部命名空间，如果有 `as` 子句则使用其指定的名称，否则使用该属性的名称

示例:

```
import foo                  # foo 被导入并且被局部绑定
import foo.bar.baz          # foo, foo.bar 和 foo.bar.baz 被导入，foo 被局部绑定
import foo.bar.baz as fbb   # foo, foo.bar 和 foo.bar.baz 被导入，foo.bar.baz 被绑定
from foo.bar import baz     # foo, foo.bar 和 foo.bar.baz 被导入，foo.bar.baz 被绑定
from foo import attr        # foo 被导入并且 foo.attr 被绑定为 attr
```

如果标识符列表改为一个星号 (`'*'`)，则在模块中定义的全部公有名称都将按 `import` 语句所在的作用域被绑定到局部命名空间。

一个模块所定义的 *公有名称* 是由在模块的命名空间中检测一个名为 `__all__` 的变量来确定的；如果有定义，它必须是一个字符串列表，其中的项为该模块所定义或导入的名称。 在 `__all__` 中所给出的名称都会被视为公有并且应当存在。 如果 `__all__` 没有被定义，则公有名称的集合将包含在模块的命名空间中找到的所有不以下划线字符 (`'_'`) 打头的名称。 `__all__` 应当包括整个公有 API。 它的目标是避免意外地导出不属于 API 的一部分的项（例如在模块内部被导入和使用的库模块）。

通配符形式的导入 --- `from module import *` --- 仅在模块层级上被允许。 尝试在类或函数定义中使用它将引发 `SyntaxError`。

当指定要导入哪个模块时，你不必指定模块的绝对名称。 当一个模块或包被包含在另一个包之中时，可以在同一个最高层级包中进行相对导入，而不必提及包名称。 通过在 `from` 之后指定的模块或包中使用前缀点号，你可以在不指定确切名称的情况下指明在当前包层级结构中要上溯多少级。 一个前缀点号表示是执行导入的模块所在的当前包，两个点号表示上溯一个包层级。 三个点号表示上溯两级，依此类推。 因此如果你执行 `from . import mod` 时所处位置为 `pkg` 包内的一个模块，则最终你将导入 `pkg.mod`。 如果你执行 `from ..subpkg2 import mod` 时所处位置为 `pkg.subpkg1` 则你将导入 `pkg.subpkg2.mod`。 有关相对导入的规范说明包含在 包相对导入 一节中。

`importlib.import_module()` 被提供用来为动态地确定要导入模块的应用提供支持。

引发一个 审计事件 `import` 并附带参数 `module`, `filename`, `sys.path`, `sys.meta_path`, `sys.path_hooks`。

7.11.1. future 语句

future 语句 是一种针对编译器的指令，指明某个特定模块应当使用在特定的未来某个 Python 发行版中成为标准特性的语法或语义。

future 语句的目的是使得向在语言中引入了不兼容改变的 Python 未来版本的迁移更为容易。 它允许基于每个模块在某种新特性成为标准之前的发行版中使用该特性。

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

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

- 模块的文档字符串（如果存在），
- 注释，

- 空行，以及
- 其他 future 语句。

唯一需要使用 future 语句的特性是 `annotations` (参见 **PEP 563**)。

future 语句所启用的所有历史特性仍然为 Python 3 所认可。 其中包括 `absolute_import`, `division`, `generators`, `generator_stop`, `unicode_literals`, `print_function`, `nested_scopes` 和 `with_statement`。 它们都已成为冗余项，因为它们总是为已启用状态，保留它们只是为了向后兼容。

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

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

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

相应的运行时语义取决于 future 语句所启用的指定特性。

请注意以下语句没有任何特别之处:

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

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

在默认情况下，通过对内置函数 `exec()` 和 `compile()` 的调用编译的代码如果出现于一个包含有 future 语句的模块 `M` 之中，就会使用该 future 语句所关联的语法和语义。 此行为可以通过传给 `compile()` 的可选参数来控制 --- 请参阅该函数的文档了解详情。

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

参见:

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

7.12. global 语句

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

`global` 语句将使其所列出的标识符被解读为全局变量。 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。

global 语句将应用于整个当前作用域（模块、函数体或类定义）。 如果一个变量在本作用域的 global 声明之前被使用或赋值则会引发 SyntaxError。

在模块层级上，所有变量都是全局变量，因此 global 语句将没有用处。 不过，变量仍然不能在其 global 声明之前被使用或赋值。 此项要求在交互式提示符 (REPL) 中被取消。

程序员注意事项: global 是对解析器的指令。 它仅对与 global 语句同时被解析的代码起作用。 特别地，包含在提供给内置 exec() 函数字符串或代码对象中的 global 语句并不会影响 *包含* 该函数调用的代码块，而包含在这种字符串中的代码也不会受到包含该函数调用的代码中的 global 语句影响。 这同样适用于 eval() 和 compile() 函数。

7.13. nonlocal 语句

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

当一个函数或类的定义嵌套（被包围）在其他函数的定义中时，其非局部作用域就是包围它的函数的局部作用域 。 nonlocal 语句会使其所列出的标识符指向之前在非局部作用域中绑定的名称。 它允许封装的代码重新绑定这样的非局部标识符。 如果一个名称在多个非局部作用域中都被绑定，则会使用最近的绑定。 如果一个名称在任何非局部作用域中都未被绑定，或者不存在非局部作用域，则会引发 SyntaxError。

nonlocal 语句将应用于函数或类语句体的整个作用域。 如果一个变量在本作用域的 nonlocal 声明之前被使用或赋值则会引发 SyntaxError。

参见:

PEP 3104 - 访问外层作用域中的名称
有关 nonlocal 语句的规范说明。

程序员注意事项: nonlocal 是对解析器的指令并且仅会在与其一同被解析的代码上应用。 参见 global 语句的相关注意事项。

7.14. type 语句

```
type_stmt: 'type' identifier [type_params] "=" expression
```

type 语句声明一个类型别名，即 typing.TypeAliasType 的实例。

例如，以下语句创建了一个类型别名:

```
type Point = tuple[float, float]
```

此代码大致等价于:

```
annotation-def VALUE_OF_Point():
    return tuple[float, float]
Point = typing.TypeAliasType("Point", VALUE_OF_Point())
```

`annotation-def` 指定一个 标注作用域，其行为很像是一个函数，但有几个小差别。

类型别名的值是在标注作用域中被求值的。 当创建类型别名时它不会被求值，只有当通过该类型别名的 `__value__` 属性访问时它才会被求值 (参见 惰性求值)。 这允许类型别名引用尚未被定义的名称。

类型别名可以通过在名称之后添加 类型形参列表 来泛型化。 请参阅 泛型类型别名 了解详情。

`type` 是一个 软关键字。

> *Added in version 3.12.*

参见:

PEP 695 - 类型形参语法
引入了 `type` 语句和用于泛型类和函数的语法。