Python-6,7,8

2021年9月28日　　10:01

是Python-2笔记 的第六节
https://docs.python.org/zh-cn/3/reference/expressions.html

使用扩展 BNF 标注来描述语法而不是词法分析。

6.1. 算术转换
当对下述某个算术运算符的描述中使用了“数值参数被转换为普通类型”这样的说法，这意味着内置类型的运算符实现采用了如下运作方式:
　　如果任一参数为复数，另一参数会被转换为复数；
　　否则，如果任一参数为浮点数，另一参数会被转换为浮点数；
　　否则，两者应该都为整数，不需要进行转换。

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。

6.2.1. 标识符（名称）
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。

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。
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
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

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。
。。那么能不能 1,2,3 成为一个元组， 还是必须(1,2,3) ？
。。我记得有个情况是 最后必须加一个， 加不加，区别很大的， 是go还是py？ 是什么功能？

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。

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

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

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

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

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

从 Python 3.6 开始，在 async def 函数中可以使用 async for 子句来迭代 asynchronous iterator。 在 async def 函数中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。

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:

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:

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

个推导式时，集合会根据推导式所产生的结果元素进行构建。

空集合不能用 {} 来构建；该字面值所构建的是一个空字典。
。。集合set，空集合是 set()
。。不知道有没有 map().. list()

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/数据对系列

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。 每个映射项被会加入新的字典。 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。

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。

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。

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。

圆括号在只附带一个参数的调用中可以被省略

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

6.2.9. yield 表达式

```
yield_atom       ::=  "(" yield_expression ")"
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。

```
def gen():  # defines a generator function
    yield 123

async def agen(): # defines an asynchronous generator function
    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。

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

。。runoob

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
```

```
def fab(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b      # 使用 yield
        # print b
        a, b = b, a + b
        n = n + 1

for n in fab(5):
    print n
```

yield 的作用就是把一个函数变成一个 generator，带有 yield 的函数不再是一个普通函数，Python 解释器会将其视为一个 generator，调用 fab(5) 不会执行 fab 函数，而是返回一个 iterable 对象！在 for 循环执行时，每次循环都会执行 fab 函数内部的代码，执行到 yield b 时，fab 函数就返回一个迭代值，下次迭代时，代码从 yield b 的下一条语句继续执行，而函数的本地变量看起来和上次中断执行前是完全一样的，于是函数继续执行，直到再次遇到 yield。
。。。

6.2.9.1. 生成器-迭代器的方法

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。
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。

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。

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

6.2.9.2. 示例

演示了生成器和生成器函数的行为

```
>>> def echo(value=None):
...     print("Execution starts when 'next()' is called for the first time.")
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except Exception as e:
...                 value = e
...     finally:
```

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

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。

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。

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

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。

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。

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。

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。

6.3. 原型

原型代表编程语言中最紧密绑定的操作。

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

6.3.1. 属性引用

属性引用是后面带有一个句点加一个名称的原型:

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

此原型必须求值为一个支持属性引用的类型的对象，多数对象都支持属性引用。 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

对序列（字符串、元组或列表）或映射（字典）对象的抽取操作通常就是从相应的多项集中选择一项

```
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）。

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。

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

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

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。

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):

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

一个可选项为在位置和关键字参数后加上逗号而不影响语义。

此原型必须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。

。。。

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。

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

```
>>> f(1, *(2,))
1 2
```

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。

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。
await_expr ::= "await" primary

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。
power ::= (await_expr | primary) ["**" u_expr]
。。右侧的一元先执行，然后 幂运算 然后 左侧的一元。 。。 一元不就是 正负号嘛。

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

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

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

对 0.0 进行负数幂次运算将导致 ZeroDivisionError。 对负数进行分数幂次运算将返回 complex 数值。

此运算符可使用特殊的 __pow__() 方法来自定义。

6.6. 一元算术和位运算

所有算术和位运算具有相同的优先级

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

单目的 - (负值) 运算符会产生其数字参数的负值；该运算可通过 __neg__() 特殊方法来重载。

单目的 + (正值) 运算符会原样输出其数字参数；该运算可通过 __pos__() 特殊方法来重载。

单目的 ~ (取反) 运算符会对其整数参数按位取反。 x 的按位取反被定义为 -(x+1)。 它只作用于整数或是重载了 __invert__() 特殊方法的自定义对象。

在所有三种情况下，如果参数的类型不正确，将引发 TypeError 异常。

6.7. 二元算术运算符

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

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

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

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

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

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。

This operation can be customized using the special __truediv__() and __floordiv__() methods.

运算符 % (模) 将输出第一个参数除以第二个参数的余数。 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

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)

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）

取余 运算可使用特殊的 __mod__() 方法来自定义。

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

运算符 + (addition) 将输出其参数的和。 两个参数或者必须都为数字，或者都为相同类型的序列。 在前一种情况下，两个数字将被转换为相同类型然后相加。 在后一种情况下，将执行序列拼接操作。
此运算可使用特殊的 __add__() 和 __radd__() 方法来自定义。

运算符 - (减) 将输出其参数的差。 两个数字参数将先被转换为相同类型。
此运算可使用特殊的 __sub__() 方法来自定义。

6.8. 移位运算
移位运算的优先级低于算术运算
```
shift_expr ::=  a_expr | shift_expr ("<<" | ">>") a_expr
```

这些运算符接受整数参数。 它们会将第一个参数左移或右移第二个参数所指定的比特位数。
此运算可使用特殊的 __lshift__() 和 __rshift__() 方法来自定义。

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

6.9. 二元位运算
三种位运算具有各不相同的优先级:
```
and_expr ::=  shift_expr | and_expr "&" shift_expr
xor_expr ::=  and_expr | xor_expr "^" and_expr
or_expr  ::=  xor_expr | or_expr "|" xor_expr
```

& 运算符会对其参数执行按位 AND，参数必须都为整数或者其中之一必须为重载了 __and__() 或 __rand__() 特殊方法的自定义对象。

^ 运算符会对其参数执行按位 XOR (异 OR)，参数必须都为整数或者其中之一必须为重载了 __xor__() 或 __rxor__() 特殊方法的自定义对象。

| 运算符会对其参数执行按位 (合并) OR，参数必须都为整数或者其中之一必须为重载了 __or__() 或 __ror__() 特殊方法的自定义对象。

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

```
comp_operator ::=  "<" | ">" | "==" | ">=" | "<=" | "!="
                   | "is" ["not"] | ["not"] "in"
```

。。C中 比较的 优先级 是什么。。。C中 a<b<c 是什么？(是把第一个bool转成int？)。。传统算术法则是什么？是指 a<b && b<c ？

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

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

正式的说法是这样：如果 a, b, c, ..., y, z 为表达式而 op1, op2, ..., opN 为比较运算符，则 a op1 b op2 c ... y opN z 就等价于 a op1 b and b op2 c and ... y opN z，不同点在于每个表达式最多只被求值一次。

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

6.10.1. 值比较

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

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

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。

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。

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。

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。

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

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）。

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。

6.11. 布尔运算

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

在执行布尔运算的情况下，或是当表达式被用于流程控制语句时，以下值会被解析为假值：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）。

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。
。。短路

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 ''。）
。。。。直接三目了啊。 简化版的三目。。
。。py应该没有三目， go呢？也没有，想起来了，go需要自己func下的。。
。。有三目，没有?: 用 x if C else y

6.12. 赋值表达式

```
assignment_expression ::=  [identifier ":="] expression
```

赋值表达式（有时又被叫做“命名表达式”或“海象表达式”）将一个 expression 赋值给一个 identifier，同时还返回 expression 的值。

一个常见用例是在处理匹配的正则表达式的时候:

```
if matching := pattern.search(data):
    do_something(matching)
```

或者是在处理分块的文件流的时候:

```
while chunk := file.read(9000):
    process(chunk)
```

。。和普通等于号有什么区别?
。。感觉这里说的是 变量是一个 表达式。 而不是表达式的值。

6.13. 条件表达式

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

条件表达式（有时称为“三元运算符”）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。

6.14. lambda 表达式

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

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

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

请参阅 函数定义(这个是8.6) 了解有关参数列表的句法。 请注意通过 lambda 表达式创建的函数不能包含语句或标注。

6.15. 表达式列表

```
expression_list    ::=  expression ("," expression)* [","]
starred_list       ::=  starred_item ("," starred_item)* [","]
starred_expression ::=  expression | (starred_item ",")* [starred_item]
starred_item       ::=  assignment_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。
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: ()。）

6.16. 求值顺序

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

6.17. 运算符优先级

相同单元格内的运算符从左至右分组（除了幂运算是从右至左分组）。

运算符	描述
(expressions...),[expressions...], {key:	绑定或加圆括号的表达式，列表显示，

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

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。

```
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
```

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。

```
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
```

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

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

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。

。。。。。。。凭感觉吧。。

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         # i is updated, then x[i] is updated
print(x)
```

7.2.1. 增强赋值语句

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

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。

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。

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。 标注绝不会在函数作用域内被求值和保存。
如果存在右手边，带标注的赋值会在对标注求值之前（如果适用）执行实际的赋值。 如果用作表达式目标的右手边不存在，则解释器会对目标求值，但最后的 __setitem__() 或 __setattr__() 调用除外。

7.3. assert 语句

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

简单形式 assert expression 等价于

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

扩展形式 assert expression1, expression2 等价于

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

以上等价形式假定 __debug__ 和 AssertionError 指向具有指定名称的内置变量。

在当前实现中，内置变量 __debug__ 在正常情况下为 True，在请求优化时为 False (对应命令行选项为 -O)。
请注意不必在错误信息中包含失败表达式的源代码；它会被作为栈追踪的一部分被显示。
赋值给 __debug__ 是非法的。 该内置变量的值会在解释器启动时确定。

7.4. pass 语句
pass 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

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。
目标列表的删除将从左至右递归地删除每一个目标。

7.6. return 语句
return_stmt ::= "return" [expression_list]

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 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。

7.8. raise 语句
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。

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:

```
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 必须为另一个异常类或实例。 如果第二个表达式是一个异常实例，它将作为可写的 __cause__ 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出来:

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:

异常串连可通过在 from 子句中指定 None 来显式地加以抑制:

7.9. break 语句

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

继续执行最近的外层循环的下一个轮次

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:

- 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。

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

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

如果标识符列表改为一个星号 ('*')，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。
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。

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。

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。
它允许基于每个模块在某种新特性成为标准之前的发行版中使用该特性。

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

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

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

唯一需要使用 future 语句的特性是 标注 (参见 PEP 563)。

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。

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。

7.12. global 语句

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

global 语句是作用于整个当前代码块的声明。 它意味着所列出的标识符将被解读为全局变量。 要给全局变量赋值不可能不用到 global 关键字，不过自由变量也可以指向全局变量而不必声明为全局变量。
在 global 语句中列出的名称不得在同一代码块内该 global 语句之前的位置中使用。
在 global 语句中列出的名称不能被定义为形式参数，也不能在 for 循环的控制目标、class 定义、函数定义、import 语句或变量标注中定义。

7.13. nonlocal 语句

```
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。

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。

这种情形下分号的绑定比冒号更紧密，因此在以下示例中，所有 print() 调用或者都不执行，或者都执行:

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

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

8.1. if 语句

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

对表达式逐个求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。值直至找到一个真值在子句体中选择唯一匹配的一个

8.2. while 语句

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

这将重复地检验表达式，并且如果其值为真就执行第一个子句体；如果表达式值为假（这可能在第一次检验时就发生）则如果 else 子句体存在就会被执行并终止循环。
第一个子句体中的 break 语句在执行时将终止循环且不执行 else 子句体。 第一个子句体中的 continue 语句在执行时将跳过子句体中的剩余部分并返回检验表达式。

8.3. for 语句

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

```
for_stmt ::=  "for" target_list "in" expression_list ":" suite
              ["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。

or 循环会对目标列表中的变量进行赋值。 这将覆盖之前对这些变量的所有赋值，包括在 for 循环体中的赋值:

```
for i in range(10):
    print(i)
    i = 5             # this will not affect the for-loop
                      # because i will be overwritten with the next
                      # index in the range
```

目标列表中的名称在循环结束时不会被删除，但如果序列为空，则它们根本不会被循环所赋值。

。。没有 for i=0;i<10;i++ 这种， 只有 for i in range(10)

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。
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。
对整个序列使用切片来创建一个临时副本

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

8.4. try 语句

为一组语句指定异常处理器和/或清理代码

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

except 子句指定一个或多个异常处理程序。
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。

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。

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

被转写为
```
except E as N:
    try:
        foo
    finally:
        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。

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

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 finally 子句引发了另一个异常，被保存的异常会被设为新异常的上下文。 如果 finally 子句执行了 return, break 或 continue 语句，则被保存的异常会被丢弃:
```
>>> def f():
...     try:
...         1/0
...     finally:
...         return 42
...
>>> f()
42
```

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

```
...
>>> foo()
'finally'
```

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。

```
with_stmt ::=  "with" with_item ("," with_item)* ":" suite
with_item ::=  expression ["as" target]
```

带有一个“项目”的 with 语句的执行过程如下:

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 __enter__() 以便后续使用。

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。

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。

```
with EXPRESSION as TARGET:
    SUITE
在语义上等价于:
manager = (EXPRESSION)
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)
hit_except = False
try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise
finally:
```

```
    if not hit_except:
        exit(manager, None, None, None)
```

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

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

在语义上等价于:

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

8.6. 函数定义

```
funcdef                    ::=  [decorators] "def" funcname "(" [parameter_list]
")"
                                ["->" expression] ":" suite
decorators                 ::=  decorator+
decorator                  ::=  "@" assignment_expression NEWLINE
parameter_list             ::=  defparameter ("," defparameter)* "," "/" [","
[parameter_list_no_posonly]]
                                  | parameter_list_no_posonly
parameter_list_no_posonly ::=  defparameter ("," defparameter)* [","
[parameter_list_starargs]]
                                | parameter_list_starargs
parameter_list_starargs    ::=  "*" [parameter] ("," defparameter)* ["," ["**"
parameter [","]]]
                                | "**" parameter [","]
parameter                  ::=  identifier [":" expression]
defparameter               ::=  parameter ["=" expression]
funcname                   ::=  identifier
```

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

函数定义并不会执行函数体；只有当函数被调用时才会执行此操作

一个函数定义可以被一个或多个 decorator 表达式所包装。 当函数被定义时将在包含该函数定义的作用域中对装饰器表达式求值。 求值结果必须是一个可调用对象，它会以该函数对象作为唯一参数被发起调用。 其返回值将被绑定到函数名称而非函数对象。 多个装饰器会以嵌套方式被应用。

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

大致等价于

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

不同之处在于原始函数并不会被临时绑定到名称 func。

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

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

如果存在 "*identifier" 这样的形式，它会被初始化为一个元组来接收任何额外的位置参数，默认为一个空元组。
如果存在 "**identifier" 这样的形式，它会被初始化为一个新的有序映射来接收任何额外的关键字参数，默认为一个相同类型的空映射。
在 "*" 或 "*identifier" 之后的形参都是仅限关键字形参因而只能通过关键字参数传入。
在 "/" 之前的形参都是仅限位置形参因而只能通过位置参数传入。
在 3.8 版更改: 可以使用 / 函数形参语法来标示仅限位置形参。 请参阅 PEP 570 了解详情。

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。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。

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。

8.7. 类定义

类定义就是对类对象的定义

```
classdef    ::=  [decorators] "class" classname [inheritance] ":" suite
inheritance ::=  "(" [argument_list] ")"
```

```
classname   ::=  identifier
```

类定义是一条可执行语句。 其中继承列表通常给出基类的列表（进阶用法请参见 元类），列表中的每一项都应当被求值为一个允许子类的类对象。 没有继承列表的类默认继承自基类 object；

```
class Foo:
    pass
```

等价于

```
class Foo(object):
    pass
```

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，：

```
@f1(arg)
@f2
class Foo: pass
```

大致等价于

```
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

装饰器表达式的求值规则与函数装饰器相同。 结果随后会被绑定到类名称。

在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。

8.8. 协程

3.5 新版功能.

8.8.1. 协程函数定义

```
async_funcdef ::=  [decorators] "async" "def" funcname "(" [parameter_list] ")"
                   ["->" 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。

```
async def func(param1, param2):
    do_stuff()
```

```
    await some_coroutine()
```

8.8.2. async for 语句

```
async_for_stmt ::=  "async" for_stmt
```

asynchronous iterable 提供了 __aiter__ 方法，该方法会直接返回 asynchronous iterator，它可以在其 __anext__ 方法中调用异步代码。

async for 语句允许方便地对异步可迭代对象进行迭代

```
async for TARGET in ITER:
    SUITE
else:
    SUITE2
```

在语义上等价于:

```
iter = (ITER)
iter = type(iter).__aiter__(iter)
running = True
while running:
    try:
        TARGET = await type(iter).__anext__(iter)
    except StopAsyncIteration:
        running = False
    else:
        SUITE
else:
    SUITE2
```

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

```
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。

```
async with EXPRESSION as TARGET:
    SUITE
```

在语义上等价于:

```
manager = (EXPRESSION)
aenter = type(manager).__aenter__
aexit = type(manager).__aexit__
value = await aenter(manager)
hit_except = False
try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not await aexit(manager, *sys.exc_info()):
```

```
        raise
finally:
    if not hit_except:
        await aexit(manager, None, None, None)
```

在协程函数体之外使用 async with 语句将引发 SyntaxError。