6. 表达式

本章将解释 Python 中组成表达式的各种元素的的含义。

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

```
name: othername
```

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

6.1. 算术转换

如果在下面某个算术运算符的描述中使用了“数字参数被转换为普通实数类型”这样的说法，则意味着针对内置类型的运算符实现的作用方式如下：

- 如果两个参数均为复数，则不会执行任何转换；
- 如果任一参数为复数或浮点数，另一参数将被转换为浮点数；
- 否则，两者应该都为整数，不需要进行转换。

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

6.2. 原子

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

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

6.2.1. 标识符（名称）

作为原子出现的标识符叫做名称。 请参看 名称（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。

6.2.1.1. 私有名称的 mangling

当类定义中出现的标识符，以两个或更多下划线开头，并且不以两个或更多下划线结尾，就称它为类的 *private name* 。

参见: 类规范说明。

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

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

- 用作被分配或读取的变量的名字的，或者用作被访问的属性的名字的。

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

- 导入的模块的名称，例如 `import __spam` 中的 `__spam`。 若模块属于一个包（即它的名称中有点号），这个名称 *不会* 被 mangle，比如 `import __foo.bar` 中的 `__foo` 不会被 mangle。

- 导入的成员的名称，比如 `from spam import __f` 中的 `__f`。

转换规则的定义如下：

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

6.2.2. 字面值

Python 支持字符串和字节串字面值，以及几种数字字面值:

```
literal: strings | NUMBER
```

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 字面量 小节。有关 `string` 的详细信息，请参阅 字符串字面值合并 小节。

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

6.2.2.1. 字符串字面值合并

允许多个相邻的字符串或字节字面量（以空白符分隔）可以使用不同的引号约定，其含义与将这些字面量拼接后的结果相同:

```
>>> "hello" 'world'
"helloworld"
```

形式上：

```
strings: ( STRING | fstring)+ | tstring+
```

此特性是在语法层面定义的，因此仅适用于字面量。若要在运行时拼接字符串表达式，可以使用 '+' 运算符:

```
>>> greeting = "Hello"
>>> space = " "
>>> name = "Blaise"
>>> print(greeting + space + name)   # 不是: print(greeting space name)
Hello Blaise
```

字面量拼接可以自由混合原始字符串、三引号字符串和格式化字符串字面量。例如:

```
>>> "Hello" r', ' f"{name}!"
"Hello, Blaise!"
```

此特性可用于减少所需的反斜杠数量，方便地将长字符串分割到多行，甚至还能为字符串的特定部分添加注释。例如:

```
re.compile("[A-Za-z_]"       # 字母或下划线
           "[A-Za-z0-9_]*"   # 字母、数字或下划线
          )
```

不过，字节字面量只能与其他字节字面量组合，不能与任何类型的字符串字面量组合。此外，模板字符串字面量也只能与其他模板字符串字面量组合。

```
>>> t"Hello" t"{name}!"
Template(strings=('Hello', '!'), interpolations=(...))
```

6.2.3. 带圆括号的形式

带圆括号的形式是包含在圆括号中的可选表达式列表。

```
parenth_form: "(" [starred_expression] ")"
```

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

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

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

6.2.4. 列表、集合与字典的显示

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

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

推导式的常用句法元素为:

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

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

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

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

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

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

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

> *Added in version 3.6:* 引入了异步推导式。

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

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

6.2.5. 列表显示

列表显示是一个用方括号括起来的可能为空的表达式系列:

```
list_display: "[" [flexible_expression_list | comprehension] "]"
```

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

6.2.6. 集合显示

集合显示是用花括号标明的，与字典显示的区别在于没有冒号分隔的键和值:

```
set_display: "{" (flexible_expression_list | comprehension) "}"
```

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

空集合不能用 `{}` 来构建；该字面值所构建的是一个空字典。

6.2.7. 字典显示

字典显示是一个用花括号括起来的可能为空的字典条目（键/值对）系列:

```
dict_display:       "{" [dict_item_list | dict_comprehension] "}"
dict_item_list:     dict_item ("," dict_item)* [","]
dict_item:          expression ":" expression | "**" or_expr
dict_comprehension: expression ":" expression comp_for
```

字典显示会产生一个新的字典对象。

如果给出一个由逗号分隔的字典条目序列，它们会从左至右被求值以定义字典的条目：每个键对象会被用作字典中存放相应值的键。 这意味着你可以在字典条目列表中多次指定相同的键，而最终字典的值将由最后一次给出的键决定。

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

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

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

对键的取值类型的限制已列在之前的 标准类型层级结构 一节中。 （总的说来，键的类型应为 hashable，这就排除了所有可变对象。） 重复键之间的冲突不会被检测；指定键所保存的最后一个值（即在显示中排最右边的文本）将为最终的值。

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

6.2.8. 生成器表达式

生成器表达式是用圆括号括起来的紧凑形式生成器标注。

```
generator_expression: "(" expression comp_for ")"
```

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

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

圆括号在只附带一个参数的调用中可以被省略。 详情参见 调用 一节。

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

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

> *Added in version 3.6:* 引入了异步生成器表达式。

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

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

6.2.9. yield 表达式

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

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

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

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

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

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

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

当一个生成器函数被调用时，它将返回一个名为生成器的迭代器。 然后这个生成器将控制生成器函数的执行。 执行过程会在这个生成器的某个方法被调用时开始。 这时，函数会执行到第一个 yield 表达式，在那里它将再次被挂起，向生成器的调用方返回 `yield_list` 的值，或者如果 `yield_list` 被省略则返回 `None`。 所谓的挂起，就是说所有局部状态都会被保留，包括局部变量的当前绑定、指

令指针、内部求值栈及任何异常处理等等。 当通过调用生成器的某个方法恢复执行时，这个函数的运行就与 yield 表达式只是一个外部调用的情况完全一样。 在恢复执行后 yield 表达式的值取决于恢复执行所调用的方法。 如果是用 `__next__()` (一般是通过 `for` 或者 `next()` 内置函数) 则结果为 `None`。 在其他情况下，如果是用 `send()`，则结果将为传给该方法的值。

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

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

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

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

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

当yield表达式是赋值语句右侧的唯一表达式时，括号可以省略。

参见:

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

PEP 342 - 通过增强型生成器实现协程
增强生成器 API 和语法的提议，使其可以被用作简单的协程。

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

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

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

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

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

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

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

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

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

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

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

generator.**close**()

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

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

6.2.9.2. 例子

这里是一个简单的例子，演示了生成器和生成器函数的行为:

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

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

对于 `yield from` 的例子，参见"Python 有什么新变化"中的 PEP 380: 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。

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

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

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

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

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

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。

async agen.**__anext__**()

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

此方法通常是通过 `async for` 循环隐式地调用。

async agen.**asend**(*value*)

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

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

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

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

async agen.**aclose**()

返回一个可等待对象，它会在运行时向异步生成器函数暂停的位置抛入一个 `GeneratorExit`。如果该异步生成器函数正常退出、关闭或引发 `GeneratorExit` (由于未捕获该异常) 则返回的可等待对象将引发 `StopIteration` 异常。 后续调用异步生成器所返回的任何其他可等待对象将引发 `StopAsyncIteration` 异常。 如果异步生成器产生了一个值，该可等待对象会引发

RuntimeError。 如果异步生成器引发任何其他异常，它会被传播给可等待对象的调用者。 如果异步生成器已经由于异常或正常退出则后续调用 aclose() 将返回一个不会做任何事的可等待对象。

6.3. 原型

原型代表编程语言中最紧密绑定的操作。 它们的句法如下:

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

6.3.1. 属性引用

属性引用是后面带有一个句点加一个名称的原型:

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

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

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

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

6.3.2. 抽取

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

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

当一个对象被抽取时，解释器将对原型和表达式列表进行求值。

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

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

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

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

1. 映射。 如果原型是一个 mapping，则表达式列表必须求值为一个以该映射的某个键为值的对象，而抽取操作会在映射中选取该键所对应的值。 内置映射类的一个例子是 dict 类。

2. 序列。 如果原型是一个 sequence，则表达式列表必须求值为一个 int 或一个 slice (如下面的小节所讨论的)。 内置序列类的例子包括 str, list 和 tuple 等类。

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

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

6.3.3. 切片

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

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

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

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

6.3.4. 调用

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

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

```
                    ("," keyword_item | "," "**" expression)*
keyword_item:       identifier "=" expression
```

一个可选项为在位置和关键字参数后加上逗号而不影响语义。

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

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

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

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

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

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

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

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

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

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

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

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

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

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

如果类型为---

用户自定义函数:
: 函数的代码块会被执行，并向其传入参数列表。 代码块所做的第一件事是将正式形参绑定到对应的参数；相关说明参见 函数定义 一节。 当代码块执行 `return` 语句时，这将指定函数调用的返回值。 如果执行到达代码块的末尾时并未执行过 `return` 语句，则返回值将为 `None`。

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:
: 该类定义定义 `__call__()` 方法；其效果将等价于调用该方法。

6.4. await 表达式

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

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

> *Added in version 3.5.*

6.5. 幂运算符

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

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

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

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

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

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

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

6.6. 一元算术和位运算

所有算术和位运算具有相同的优先级:

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

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

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

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

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

6.7. 二元算术运算符

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

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

运算符 `*` (乘) 将产生其参数的乘积。 两个参数必须或者都为数字，或者一个参数必须为整数而另一个必须为序列。 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；重复因子为负则产生一个空序列。

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

> *在 3.14 版本发生变更:* 如果只有一个操作数为复数，另一参数将被转换为浮点数。

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

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

> *Added in version 3.5.*

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

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

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

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。

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

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

运算符 `+` (加) 将产生其参数的和。 两个参数必须或者都为数字，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，将执行序列的拼接。

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

> *在 3.14 版本发生变更:* 如果只有一个操作数为复数，另一参数将被转换为浮点数。

运算符 `-` (减) 将产生其参数的差。 两个数字参数将先被转换为相同的实数类型。

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

> *在 3.14 版本发生变更:* 如果只有一个操作数为复数，另一参数将被转换为浮点数。

6.8. 移位运算

移位运算的优先级低于算术运算:

```
shift_expr: a_expr | shift_expr ("<<" | ">>") a_expr
```

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

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

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

6.9. 二元位运算

三种位运算具有各不相同的优先级:

```
and_expr: shift_expr | and_expr "&" shift_expr
xor_expr: and_expr | xor_expr "^" and_expr
or_expr:  xor_expr | or_expr "|" xor_expr
```

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

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

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

6.10. 比较运算

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

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

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

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

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

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

6.10.1. 值比较

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

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

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

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

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

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

以下列表描述了最主要内置类型的比较行为。

- 内置数值类型 (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排序比较均返回假值。 还有一个反直觉的结果是非数字值不等于其自身。 举例来说，如果 `x = float('NaN')` 则 `3 < x`, `x < 3` 和 `x == x` 均为假值，而 `x != x` 则为真值。 此行为是遵循 IEEE 754 标准的。

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

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

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

  字符串和二进制码序列不能直接比较。

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

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

  内置多项集间的字典序比较规则如下:

  - 两个多项集若要相等，它们必须为相同类型、相同长度，并且每对相应的元素都必须相等（例如，`[1,2] == (1,2)` 为假值，因为类型不同）。
  - 对于支持次序比较的多项集，排序与其第一个不相等元素的排序相同（例如 `[1,2,x] <= [1,2,y]` 的值与 `x <= y` 相同）。 如果对应元素不存在，较短的多项集排序在前（例如 `[1,2] < [1,2,3]` 为真值）。

- 两个映射 (`dict` 的实例) 若要相等则必须当且仅当它们具有相等的 `(key, value)` 对。 键和值的相相等性比较强制要求自反射性。

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

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

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

  集合的比较强制规定其元素的自反射性。

- 大多数其他内置类型没有实现比较方法，因此它们会继承默认的比较行为。

在可能的情况下，用户定义类在定制其比较行为时应当遵循一些一致性规则:

- 相等比较应该是自反射的。 换句话说，相同的对象比较时应该相等:

  `x is y` 意味着 `x == y`

- 比较应该是对称的。 换句话说，下列表达式应该有相同的结果:

  `x == y` 和 `y == x`

  `x != y` 和 `y != x`

  `x < y` 和 `y > x`

  `x <= y` 和 `y >= x`

- 比较应该是可传递的。 下列（简要的）例子显示了这一点:

  `x > y and y > z` 意味着 `x > z`

`x < y and y <= z` 意味着 `x < z`

- 反向比较应该导致布尔值取反。 换句话说，下列表达式应该有相同的结果:

  `x == y` 和 `not x != y`

  `x < y` 和 `not x >= y` (对于完全排序)

  `x > y` 和 `not x <= y` (对于完全排序)

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

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

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

6.10.2. 成员检测运算

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

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

对于定义了For user-defined classes which define the `__contains__()` 方法来用户自定义类来说，如果 `y.__contains__(x)` 返回真值则 `x in y` 将返回 `True`，否则返回 `False`。

对于未定义 `__contains__()` 但定义了 `__iter__()` 的用户自定义类来说，如果在迭代 `y` 期间产生了值 `z` 使得表达式 `x is z or x == z` 为真值，则 `x in y` 将为 `True`。 如果在迭代期间引发了异常，则将等同于 `in` 引发了该异常。

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

运算符 `not in` 被定义为具有与 `in` 相反的逻辑值。

6.10.3. 标识号比较

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

6.11. 布尔运算

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

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

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

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

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

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

6.12. 赋值表达式

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

赋值表达式（有时又被称为"命名表达式"或"海象表达式"）将一个 `expression` 赋值给一个 `identifier`，同时还会返回 `expression` 的值。

一个常见用例是在处理匹配的正则表达式的时候:

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

或者是在处理分块的文件流的时候:

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

赋值表达式在被用作表达式语句及在被用作切片、条件表达式、lambda 表达式、关键字参数和推导式中的 if 表达式以及在 `assert`, `with` 和 `assignment` 语句中的子表达式时必须用圆括号括起来。 在其可使用的其他场合，圆括号则不是必须的，包括在 `if` 和 `while` 语句中。

> *Added in version 3.8:* 请参阅 **PEP 572** 了解有关赋值表达式的详情。

6.13. 条件表达式

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

A conditional expression (sometimes called a "ternary operator") is an alternative to the if-else statement. As it is an expression, it returns a value and can appear as a sub-expression.

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

请参阅 **PEP 308** 了解有关条件表达式的详情。

6.14. lambda 表达式

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

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

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

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

6.15. 表达式列表

```
starred_expression:       "*" or_expr | expression
flexible_expression:      assignment_expression | starred_expression
flexible_expression_list: flexible_expression ("," flexible_expression)* [","]
starred_expression_list:  starred_expression ("," starred_expression)* [","]
expression_list:          expression ("," expression)* [","]
yield_list:               expression_list | starred_expression "," [starred_express
```

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

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

> *Added in version 3.5:* 表达式列表中的可迭代对象拆包，最初由 **PEP 448** 提出。

> *Added in version 3.11:* 一个表达式列表中的任何条目都可以带星号。 参见 **PEP 646**。

末尾的逗号仅在创建单条目元组，比如 `1,` 时才是必需的；在所有其他情况下它都是可选项。 没有末尾逗号的单独表达式不会创建一个元组，而是产生该表达式的值。 （要创建一个空元组，应使用一对内容为空的圆括号: `()`。）

6.16. 求值顺序

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

在以下几行中，表达式将按其后缀的算术优先顺序被求值。:

```
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
expr1(expr2, expr3, *expr4, **expr5)
expr3, expr4 = expr1, expr2
```

6.17. 运算符优先级

下表对 Python 中运算符的优先顺序进行了总结，从最高优先级（最先绑定）到最低优先级（最后绑定）。 相同单元格内的运算符具有相同优先级。 除非语法显式地指明，否则运算符均为双目运算符。 相同单元格内的运算符从左至右组合的（只有幂运算符是从右至左组合的）。

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

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

运算符	描述
`:=`	赋值表达式

备注

[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] 幂运算符 `**` 绑定的紧密程度低于在其右侧的算术或按位一元运算符，也就是说 `2**-1` 为 `0.5`。

[6] `%` 运算符也被用于字符串格式化；在此场合下会使用同样的优先级。