内置函数

Python 解释器内置了很多函数和类型，任何时候都能使用。以下按字母顺序给出列表。

内置函数			
A abs() aiter() all() anext() any() ascii() **B** bin() bool() breakpoint() bytearray() bytes() **C** callable() chr() classmethod() compile() complex() **D** delattr() dict() dir() divmod()	**E** enumerate() eval() exec() **F** filter() float() format() frozenset() **G** getattr() globals() **H** hasattr() hash() help() hex() **I** id() input() int() isinstance() issubclass() iter()	**L** len() list() locals() **M** map() max() memoryview() min() **N** next() **O** object() oct() open() ord() **P** pow() print() property()	**R** range() repr() reversed() round() **S** set() setattr() slice() sorted() staticmethod() str() sum() super() **T** tuple() type() **V** vars() **Z** zip() _ __import__()

abs(*number*, */*)

返回一个数字的绝对值。 参数可以是整数、浮点数或任何实现了 __abs__() 的对象。 如果参数是一个复数，则返回它的模。

aiter(*async_iterable*, */*)

返回 asynchronous iterable 的 asynchronous iterator 。相当于调用 `x.__aiter__()`。

注意：与 iter() 不同，aiter() 没有两个参数的版本。

> *Added in version 3.10.*

all(*iterable*, /)

如果 *iterable* 的所有元素均为真值（或可迭代对象为空）则返回 `True` 。 等价于：

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

awaitable **anext**(*async_iterator*, /)
awaitable **anext**(*async_iterator*, *default*, /)

当进入 await 状态时，从给定 asynchronous iterator 返回下一数据项，迭代完毕则返回 *default*。

这是内置函数 `next()` 的异步版本，类似于：

调用 *async_iterator* 的 `__anext__()` 方法，返回一个 awaitable。等待返回迭代器的下一个值。若有给出 *default*，则在迭代完毕后会返回给出的值，否则会触发 `StopAsyncIteration`。

> *Added in version 3.10.*

any(*iterable*, /)

如果 *iterable* 的任一元素为真值则返回 `True`。 如果可迭代对象为空，返回 `False`。 等价于:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

ascii(*object*, /)

与 `repr()` 类似，返回一个包含对象的可打印表示形式的字符串，但是使用 `\x`、`\u` 和 `\U` 对 `repr()` 返回的字符串中非 ASCII 编码的字符进行转义。生成的字符串和 Python 2 的 `repr()` 返回的结果相似。

bin(*integer*, /)

将一个整数转换为带前缀 "0b" 的二进制数字符串。 结果是一个合法的 Python 表达式。 如果 *integer* 不是一个 Python `int` 对象，则它必须定义返回一个整数的 `__index__()` 方法。 下面是一些例子:

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

若要控制是否显示前缀“0b”，可以采用以下两种方案：

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
```

```
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

另见 format() 获取更多信息。

class **bool**(*object=False*, */*)

返回布尔值，即 True 或 False 中的一个。 其参数将使用标准的 真值测试过程 来转换。 如果该参数为假值或被省略，则返回 False；在其他情况下，将返回 True。 bool 类是 int 的子类 (参见 数字类型 --- int, float, complex)。 它不能被继续子类化。 它只有 False 和 True 这两个实例 (参见 布尔类型 - bool)。

> *在 3.7 版本发生变更:* 该形参现在为仅限位置形参。

breakpoint(**args*, ***kws*)

此函数会在调用位置进入调试器。 具体来说，它将调用 sys.breakpointhook()，直接传递 args 和 kws。 在默认情况下，sys.breakpointhook() 将不带参数地调用 pdb.set_trace()。 在此情况下，它纯粹是一个便捷函数让你不必显式地导入 pdb 或键入过多代码即可进入调试器。 不过，sys.breakpointhook() 也可被设置为某些其他函数并被 breakpoint() 自动调用，允许你进入选定的调试器。 如果 sys.breakpointhook() 不可用，此函数将引发 RuntimeError。

在默认情况下，breakpoint() 的行为可使用 PYTHONBREAKPOINT 环境变量来改变。 请参阅 sys.breakpointhook() 了解详细用法。

请注意这并不保证 sys.breakpointhook() 会被替换。

引发一个 审计事件 builtins.breakpoint 并附带参数 breakpointhook。

> *Added in version 3.7.*

class **bytearray**(*source=b''*)
class **bytearray**(*source*, *encoding*, *errors='strict'*)

返回一个新的 bytes 数组。 bytearray 类是一个可变序列，包含范围为 0 <= x < 256 的整数。它有可变序列大部分常见的方法，见 可变序列类型 的描述；同时有 bytes 类型的大部分方法，参见 bytes 和 bytearray 操作。

可选形参 *source* 可以用不同的方式来初始化数组：

- 如果是一个 *string*，您必须提供 *encoding* 参数（*errors* 参数仍是可选的）；bytearray() 会使用 str.encode() 方法来将 string 转变成 bytes。
- 如果是一个 *integer*，会初始化大小为该数字的数组，并使用 null 字节填充。
- 如果是一个遵循 缓冲区接口 的对象，该对象的只读缓冲区将被用来初始化字节数组。
- 如果是一个 *iterable* 可迭代对象，它的元素的范围必须是 `0 <= x < 256` 的整数，它会被用作数组的初始内容。

如果没有实参，则创建大小为 0 的数组。

另见 二进制序列类型 --- bytes, bytearray, memoryview 和 bytearray 对象。

class **bytes**(*source=b''*)
class **bytes**(*source*, *encoding*, *errors='strict'*)

返回一个新的"bytes"对象，这是一个不可变序列，包含范围为 `0 <= x < 256` 的整数。`bytes` 是 `bytearray` 的不可变版本——带有同样不改变序列的方法，支持同样的索引、切片操作。

因此，构造函数的实参和 `bytearray()` 相同。

字节对象还可以用字面值创建，参见 字符串与字节串字面量。

另见 二进制序列类型 --- bytes, bytearray, memoryview，bytes 对象 和 bytes 和 bytearray 操作。

callable(*object*, /)

如果 *object* 参数是可调用的则返回 `True`，否则返回 `False`。 如果返回 `True`，调用仍可能失败，但如果返回 `False`，则调用 *object* 肯定不会成功。 请注意类是可调用的（调用类将返回一个新的实例）；如果实例所属的类有 `__call__()` 方法则它就是可调用的。

> *Added in version 3.2:* 这个函数一开始在 Python 3.0 被移除了，但在 Python 3.2 被重新加入。

chr(*codepoint*, /)

返回表示指定 Unicode 码位对应字符的字符串。例如，`chr(97)` 返回字符串 `'a'`，而 `chr(8364)` 返回字符串 `'€'`。此函数是 `ord()` 的逆操作。

该参数的有效范围是 0 到 1,114,111（16 进制表示是 0x10FFFF）。如果超过这个范围，会引发 `ValueError` 异常。

@classmethod

把一个方法封装成类方法。

类方法隐含的第一个参数就是类，就像实例方法接收实例作为参数一样。要声明一个类方法，按惯例请使用以下方案：

```
class C:
    @classmethod
    def f(cls, arg1, arg2): ...
```

`@classmethod` 这样的形式称为函数的 decorator -- 详情参阅 函数定义。

类方法的调用可以在类上进行 (例如 `C.f()`) 也可以在实例上进行 (例如 `C().f()`)。 其所属类以外的类实例会被忽略。 如果类方法在其所属类的派生类上调用，则该派生类对象会被作为隐含的第一个参数被传入。

类方法与 C++ 或 Java 中的静态方法不同。 如果你需要后者，请参阅本节中的 `staticmethod()`。 有关类方法的更多信息，请参阅 标准类型层级结构。

> *在 3.9 版本发生变更:* 类方法现在可以包装其他 描述器 例如 `property()`。

> *在 3.10 版本发生变更:* 类方法现在继承了方法的属性 (`__module__`, `__name__`, `__qualname__`, `__doc__` 和 `__annotations__`) 并具有新的 `__wrapped__` 属性。

> *Deprecated since version 3.11, removed in version 3.13:* 类方法不再可以包装其他 descriptors 例如 `property()`。

compile(*source*, *filename*, *mode*, *flags=0*, *dont_inherit=False*, *optimize=-1*)

将 *source* 编译成代码或 AST 对象。代码对象可以被 `exec()` 或 `eval()` 执行。*source* 可以是常规的字符串、字节字符串，或者 AST 对象。参见 `ast` 模块的文档了解如何使用 AST 对象。

filename 实参需要是代码读取的文件名；如果代码不需要从文件中读取，可以传入一些可辨识的值（经常会使用 `'<string>'`）。

mode 实参指定了编译代码必须用的模式。如果 *source* 是语句序列，可以是 `'exec'`；如果是单一表达式，可以是 `'eval'`；如果是单个交互式语句，可以是 `'single'`。（在最后一种情况下，如果表达式执行结果不是 `None` 将会被打印出来。）

可选参数 *flags* 和 *dont_inherit* 控制应当激活哪个 编译器选项 以及应当允许哪个 future 特性。如果两者都未提供 (或都为零) 则代码会应用与调用 `compile()` 的代码相同的旗标来编译。 如果给出了 *flags* 参数而未给出 *dont_inherit* (或者为零) 则会在无论如何都将被使用的旗标之外还会额外使用 *flags* 参数所指定的编译器选项和 future 语句。 如果 *dont_inherit* 为非零整数，则只使用 *flags* 参数 -- 外围代码中的旗标 (future 特性和编译器选项) 会被忽略。

编译器选项和 future 语句是由比特位来指明的。 比特位可以通过一起按位 OR 来指明多个选项。 指明特定 future 特性所需的比特位可以在 `__future__` 模块的 `_Feature` 实例的 `compiler_flag` 属性中找到。 编译器旗标 可以在 `ast` 模块中查找带有 `PyCF_` 前缀的名称。

optimize 实参指定编译器的优化级别；默认值 `-1` 选择与解释器的 `-O` 选项相同的优化级别。显式级别为 `0` （没有优化；`__debug__` 为真）、`1` （断言被删除，`__debug__` 为假）或 `2` （文档字符串也被删除）。

如果编译的源码不合法，此函数会触发 `SyntaxError` 异常；如果源码包含 null 字节，则会触发 `ValueError` 异常。

如果您想分析 Python 代码的 AST 表示，请参阅 `ast.parse()`。

引发一个 审计事件 `compile` 附带参数 `source` 和 `filename`。 此事件也可通过隐式编译来引发。

> **备注:** 在 `'single'` 或 `'eval'` 模式编译多行代码字符串时，输入必须以至少一个换行符结尾。 这使 `code` 模块更容易检测语句的完整性。

> **警告:** 在将足够大或者足够复杂的字符串编译成 AST 对象时，Python 解释器有可能因为 Python AST 编译器的栈深度限制而崩溃。

> *在 3.2 版本发生变更:* Windows 和 Mac 的换行符均可使用。而且在 `'exec'` 模式下的输入不必再以换行符结尾了。另增加了 *optimize* 参数。

> *在 3.5 版本发生变更:* 之前 *source* 中包含 null 字节的话会触发 `TypeError` 异常。

> *Added in version 3.8:* `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` 现在可在旗标中传入以启用对最高层级 `await`, `async for` 和 `async with` 的支持。

class **complex**(*number=0*, */*)
class **complex**(*string*, */*)
class **complex**(*real=0*, *imag=0*)

将特定的字符串或数字转换为一个复数，或基于特定的实部和虚部创建一个复数。

示例：

```
>>> complex('+1.23')
(1.23+0j)
>>> complex('-4.5j')
-4.5j
>>> complex('-1.23+4.5j')
(-1.23+4.5j)
>>> complex('\t( -1.23+4.5J )\n')
(-1.23+4.5j)
>>> complex('-Infinity+NaNj')
(-inf+nanj)
>>> complex(1.23)
(1.23+0j)
>>> complex(imag=-4.5)
-4.5j
>>> complex(-1.23, 4.5)
(-1.23+4.5j)
```

如果该参数为字符串，则它必须包含一个实部（格式与 `float()` 接受格式相同）或一个虚部（与实部格式相同但带有 `'j'` 或 `'J'` 后缀），或者同时包含实部和虚部（在此情况下虚部必须加上正负号）。 该字符串首尾可以被空白字符和圆括号 `'('` and `')'` 包裹，但它们会被忽略。该字符串中的 `'+'`, `'-'`, `'j'` 或 `'J'` 后缀以及十进制数字之间不可存在空格。 例如，`complex('1+2j')` 是可以的，但 `complex('1 + 2j')` 则会引发 `ValueError`。 更准确地说，输入在移除圆括号以及开头和末尾的空白字符之后，必须符合以下 `complexvalue` 产生式规则：

```
complexvalue: floatvalue |
              floatvalue ("j" | "J") |
              floatvalue sign absfloatvalue ("j" | "J")
```

如果该参数为数字，则此构造器将进行与 `int` 和 `float` 类似的数值转换。 对于一个普通的 Python 对象 `x`，`complex(x)` 会委托给 `x.__complex__()`。 如果未定义 `__complex__()` 则它将回退至 `__float__()`。 如果未定义 `__float__()` 则它将回退至 `__index__()`。

如果提供了两个参数或是使用了关键字参数，则每个参数可以为任意数字类型（包括复数）。如果两个参数均为实数值，则会返回一个实部为 *real* 而虚部为 *imag* 的复数。 如果两个参数均

为复数值，则会返回一个实部为 `real.real-imag.imag` 而虚部为 `real.imag+imag.real` 的复数。 如果有一个参数为实数值，则上面的表达式中将只用到实部。

另请参阅仅接受单个数字参数的 `complex.from_number()`。

如果省略所有参数，则返回 `0j`。

数字类型 --- int, float, complex 描述了复数类型。

> *在 3.6 版本发生变更:* 您可以使用下划线将代码文字中的数字进行分组。

> *在 3.8 版本发生变更:* 如果 `__complex__()` 和 `__float__()` 均未定义则回退至 `__index__()`。

> *自 3.14 版本弃用:* 参数 以 *real* 或 *imag* 的形式传递复数现在已弃用；只能以单个位置参数传递。

delattr(*object*, *name*, /)

这是 `setattr()` 的相关函数。 其参数是一个对象和一个字符串。 其中字符串必须是对象的某个属性的名称。 该函数会删除指定的属性，如果对象允许这样做的话。 例如，`delattr(x, 'foobar')` 等价于 `del x.foobar`。 *name* 不要求必须是 Python 标识符 (参见 `setattr()`)。

class **dict**(***kwargs*)
class **dict**(*mapping*, /, ***kwargs*)
class **dict**(*iterable*, /, ***kwargs*)

创建一个新的字典。`dict` 对象是一个字典类。参见 `dict` 和 映射类型 --- dict 了解这个类。

其他容器类型，请参见内置的 `list`、`set` 和 `tuple` 类，以及 `collections` 模块。

dir()
dir(*object*, /)

如果没有实参，则返回当前本地作用域中的名称列表。如果有实参，它会尝试返回该对象的有效属性列表。

如果对象有一个名为 `__dir__()` 的方法，则该方法将被调用并且必须返回由属列组成的列表。这允许实现自定义This allows objects that implement a custom `__getattr__()` 或 `__getattribute__()` 函数的对象能够定制 `dir()` 报告其属性的方式。

如果对象未提供 `__dir__()`，该函数会尽量从对象所定义的 `__dict__` 属性和其类型对象中收集信息。 结果列表不一定是完整的，并且当对象具有自定义的 `__getattr__()` 时还可能是不准确的。

默认的 `dir()` 机制对不同类型的对象行为不同，它会试图返回最相关而不是最全的信息：

- 如果对象是模块对象，则列表包含模块的属性名称。
- 如果对象是类型或类对象，则列表包含它们的属性名称，并且递归查找所有基类的属性。
- 否则，列表包含对象的属性名称，它的类属性名称，并且递归查找它的类的所有基类的属性。

返回的列表按字母表排序。例如：

```
>>> import struct
>>> dir()   # show the names in the module namespace
['__builtins__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
...
>>> s = Shape()
>>> dir(s)
['area', 'location', 'perimeter']
```

备注： 因为 `dir()` 主要是为了便于在交互式时使用，所以它会试图返回人们感兴趣的名字集合，而不是试图保证结果的严格性或一致性，它具体的行为也可能在不同版本之间改变。例如，当实参是一个类时，metaclass 的属性不包含在结果列表中。

divmod(*a*, *b*, */*)

接受两个（非复数）数字作为参数并返回由当对其使用整数除法时的商和余数组成的数字对。在混用不同的操作数类型时，则会应用二元算术运算符的规则。 对于整数来说，结果与 `(a // b, a % b)` 相同。 对于浮点数来说则结果为 `(q, a % b)`，其中 *q* 通常为 `math.floor(a / b)` 但可能会比它小 1。 在任何情况下 `q * b + a % b` 都非常接近 *a*，如果 `a % b` 为非零值则它将具有与 *b* 相同的正负号，并且 `0 <= abs(a % b) < abs(b)`。

enumerate(*iterable*, *start=0*)

返回一个枚举对象。*iterable* 必须是一个序列，或 iterator，或其他支持迭代的对象。`enumerate()` 返回的迭代器的 `__next__()` 方法返回一个元组，里面包含一个计数值（从 *start* 开始，默认为 0）和通过迭代 *iterable* 获得的值。

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

等价于:

```
def enumerate(iterable, start=0):
    n = start
    for elem in iterable:
        yield n, elem
        n += 1
```

eval(*source*, */*, *globals=None*, *locals=None*)

参数:

- **source** (str | code object) -- 一个 Python 表达式。

- **globals** (dict | None) -- 全局命名空间 (默认值: None)。
- **locals** (mapping | None) -- 局部命名空间 (默认值: None)。

返回: 被求值表达式的求值结果。

引发: 语法错误将作为异常被报告。

警告: 此函数可执行任意代码。 调用它时附带用户提供的输入可能导致安全弱点。

expression 参数将作为一个 Python 表达式 (从技术上说，是一个条件列表) 使用 *globals* 和 *locals* 映射作为全局和局部命名空间被解析并求值。 如果 *globals* 字典存在并且不包含 `__builtins__` 键对应的值，则在 *expression* 被解析之前会插入该键对应的指向内置模块 builtins 的字典的引用。 这样你就可以在将 *globals* 传给 eval() 之前通过向其传入你自己的 `__builtins__` 字典来控制被执行代码可以使用哪些内置对象。 如果 *locals* 映射被省略则它将默认为 *globals* 字典。 如果两个映射都被省略，则将使用调用 eval() 所在环境中的 *globals* 和 *locals* 来执行该表达式。 请注意，*eval()* 将只能访问所在环境中的 嵌套作用域 (非局部作用域)，如果它们已经在调用 eval() 的作用域中被引用的话 (例如通过 nonlocal 语句)。

示例:

```
>>> x = 1
>>> eval('x+1')
2
```

该函数还可用于执行任意代码对象（比如由 compile() 创建的对象）。 这时传入的是代码对象，而非一个字符串了。如果代码对象已用参数为 *mode* 的 `'exec'` 进行了编译，那么 eval() 的返回值将为 `None`。

提示： exec() 函数支持语句的动态执行。 globals() 和 locals() 函数分别返回当前的全局和本地字典，可供传给 eval() 或 exec() 使用。

如果给出的源数据是个字符串，那么其前后的空格和制表符将被剔除。

另外可以参阅 ast.literal_eval()，该函数可以安全执行仅包含文字的表达式字符串。

引发一个 审计事件 `exec` 附带代码对象作为参数。 代码编译事件也可能被引发。

> *在 3.13 版本发生变更:* 现在可以将 *globals* 和 *locals* 作为关键字参数传入。

> *在 3.13 版本发生变更:* 默认 *locals* 命名空间的语义已被调整为与 locals() 内置函数的描述一致。

exec(*source*, */*, *globals=None*, *locals=None*, ***, *closure=None*)

警告: 此函数可执行任意代码。 调用它时附带用户提供的输入可能导致安全弱点。

这个函数支持动态执行 Python 代码。 *source* 必须是字符串或代码对象。 如果是字符串，那么该字符串将被解析为一组 Python 语句并随即被执行（除非发生语法错误）。 [1] 如果是代码对象，那么它将被直接执行。 在所有情况下，被执行的代码都应当是有效的文件输入（见参考手

册中的 文件输入 一节）。 请注意即使是在传递给 `exec()` 函数的代码的上下文中 `nonlocal`，`yield` 和 `return` 语句也不可在函数定义以外使用。 函数的返回值为 `None`。

在所有情况下，如果省略了可选部分，代码将在当前作用域中执行。 如果只提供了 *globals*，则它必须是一个字典（并且不能是字典的子类），它将被同时用于全局和局部变量。 如果给出了 *globals* 和 *locals*，它们将被分别用于全局和局部变量。 如果提供了 *locals*，它可以是任何映射对象。 请记住在模块层级上，globals 和 locals 是同一个字典。

> **备注:** 当 `exec` 获得两个不同的对象作为 *globals* 和 *locals* 时，代码被执行时就会像是嵌套在一个类定义中那样。 这意味着在被执行代码中定义的函数和类将无法访问在最高层级上赋值的变量（因为"最高层级"变量会被当作是类定义中的类变量来对待）。

如果 *globals* 字典不包含 `__builtins__` 键值，则将为该键插入对内建 `builtins` 模块字典的引用。因此，在将执行的代码传递给 `exec()` 之前，可以通过将自己的 `__builtins__` 字典插入到 *globals* 中来控制可以使用哪些内置代码。

closure 参数指定了一个闭包 —— 一个单元变量的元组。 它只有有 *object* 是一个包含 自由（闭包）变量 的代码对象时才有效。 元组的长度必须与代码对象的 `co_freevars` 属性的长度完全匹配。

引发一个 审计事件 `exec` 附带代码对象作为参数。 代码编译事件也可能被引发。

> **备注:** 内置函数 `globals()` 和 `locals()` 分别返回当前的全局和局部字典，这在用作 `exec()` 的第二个和第三个参数进行传递时会很有用处。

> **备注:** 默认的 *locals* 行为与下面 `locals()` 函数所描述的一样。 如果你需要在 `exec()` 返回之后查看代码对 *locals* 的影响可以显式地传入一个 *locals* 字典。

> *在 3.11 版本发生变更:* 添加了 *closure* 参数。

> *在 3.13 版本发生变更:* 现在可以将 *globals* 和 *locals* 作为关键字参数传入。

> *在 3.13 版本发生变更:* 默认 *locals* 命名空间的语义已被调整为与 `locals()` 内置函数的描述一致。

filter(*function*, *iterable*, /)

使用 *iterable* 中 *function* 返回真值的元素构造一个迭代器。 *iterable* 可以是一个序列，一个支持迭代的容器或者一个迭代器。 如果 *function* 为 `None`，则会使用标识号函数，也就是说，*iterable* 中所有具有假值的元素都将被移除。

请注意， `filter(function, iterable)` 相当于一个生成器表达式，当 function 不是 `None` 的时候为 `(item for item in iterable if function(item))`；function 是 `None` 的时候为 `(item for item in iterable if item)` 。

请参阅 `itertools.filterfalse()` 来了解返回 *iterable* 中 *function* 返回假值的元素的补充函数。

class **float**(*number=0.0*, */*)
class **float**(*string*, */*)

返回基于一个数字或字符串构建的浮点数。

示例：

```
>>> float('+1.23')
1.23
>>> float('   -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf
```

如果该参数是一个字符串，则它应当包含一个十进制数字，前面可以选择带一个符号，也可以选择嵌入空格。 可选的符号有 `'+'` 或 `'-'`；`'+'` 符号对所产生的值没有影响。 该参数还可以是一个代表 NaN (not-a-number) 或者正负无穷大的字符串。 更确切地说，在移除前导和尾随的空格之后，输入必须为符合以下语法的 floatvalue 产生规则：

```
sign:          "+" | "-"
infinity:      "Infinity" | "inf"
nan:           "nan"
digit:         <a Unicode decimal digit, i.e. characters in Unicode general ca
digitpart:     digit (["_"] digit)*
number:        [digitpart] "." digitpart | digitpart ["."]
exponent:      ("e" | "E") [sign] digitpart
floatnumber:   number [exponent]
absfloatvalue: floatnumber | infinity | nan
floatvalue:    [sign] absfloatvalue
```

大小写是无影响的，因此举例来说，"inf", "Inf", "INFINITY" 和 "iNfINity" 都是正无穷可接受的拼写形式。

另一方面，如果参数是整数或浮点数，则返回一个具有相同值（在 Python 浮点精度范围内）的浮点数。 如果参数超出了 Python 浮点数的取值范围，则会引发 OverflowError。

对于一个普通 Python 对象 `x`，`float(x)` 会委托给 `x.__float__()`。 如果 `__float__()` 未定义则将回退至 `__index__()`。

另请参见 `float.from_number()`，它只接受数字参数。

如果没有实参，则返回 `0.0` 。

数字类型 --- int, float, complex 描述了浮点类型。

> *在 3.6 版本发生变更:* 您可以使用下划线将代码文字中的数字进行分组。

> *在 3.7 版本发生变更:* 该形参现在为仅限位置形参。

> *在 3.8 版本发生变更:* 如果 `__float__()` 未定义则回退至 `__index__()`。

format(*value*, *format_spec=''*, /)

将 *value* 转换为"格式化后"的形式，格式由 *format_spec* 进行控制。*format_spec* 的解释方式取决于 *value* 参数的类型；但大多数内置类型使用一种标准的格式化语法：格式规格迷你语言。

默认的 *format_spec* 是一个空字符串，它通常给出与调用 `str(value)` 相同的结果。

对 `format(value, format_spec)` 的调用会转写为 `type(value).__format__(value, format_spec)`，这样在搜索值的 `__format__()` 方法时将绕过实例字典。如果方法搜索到达 `object` 并且 *format_spec* 不为空，或者如果 *format_spec* 或返回值不为字符串则会引发 `TypeError` 异常。

> *在 3.4 版本发生变更:* 当 *format_spec* 不是空字符串时，`object().__format__(format_spec)` 会触发 `TypeError`。

class **frozenset**(*iterable=()*, /)

返回一个新的 `frozenset` 对象，它包含可选参数 *iterable* 中的元素。`frozenset` 是一个内置的类。有关此类的文档，请参阅 `frozenset` 和 集合类型 --- set, frozenset。

请参阅内建的 `set`、`list`、`tuple` 和 `dict` 类，以及 `collections` 模块来了解其它的容器。

getattr(*object*, *name*, /)
getattr(*object*, *name*, *default*, /)

object 中指定名称的属性的值。*name* 必须是字符串。如果该字符串是对象的某一属性的名称，则结果将为该属性的值。例如，`getattr(x, 'foobar')` 等同于 `x.foobar`。如果指定名称的属性不存在，则如果提供了 *default* 则返回该值，否则将引发 `AttributeError`。*name* 不必是一个 Python 标识符 (参见 `setattr()`)。

> **备注:** 由于 私有名称混合 发生在编译时，因此必须手动混合私有属性（以两个下划线打头的属性）名称以使用 `getattr()` 来提取它。

globals()

返回实现当前模块命名空间的字典。对于函数内的代码，这是在定义函数时设置的，无论函数在哪里被调用都保持不变。

hasattr(*object*, *name*, /)

该实参是一个对象和一个字符串。如果字符串是对象的属性之一的名称，则返回 `True`，否则返回 `False`。（此功能是通过调用 `getattr(object, name)` 看是否有 `AttributeError` 异常来实现的。）

hash(*object*, /)

返回该对象的哈希值（如果它有的话）。哈希值是整数。它们在字典查找元素时用来快速比较字典的键。相同大小的数字变量有相同的哈希值（即使它们类型不同，如 1 和 1.0）。

备注： 对于具有自定义 `__hash__()` 方法的对象，请注意 `hash()` 会根据宿主机的字长来截断返回值。

help()
help(*request*)

启动内置的帮助系统（此函数主要在交互式中使用）。如果没有实参，解释器控制台里会启动交互式帮助系统。如果实参是一个字符串，则在模块、函数、类、方法、关键字或文档主题中搜索该字符串，并在控制台上打印帮助信息。如果实参是其他任意对象，则会生成该对象的帮助页。

请注意，如果在调用 `help()` 时，目标函数的形参列表中存在斜杠（/），则意味着斜杠之前的参数只能是位置参数。详情请参阅 有关仅限位置形参的 FAQ 条目。

该函数通过 `site` 模块加入到内置命名空间。

> *在 3.4 版本发生变更:* `pydoc` 和 `inspect` 的变更使得可调用对象的签名信息更加全面和一致。

hex(*integer*, /)

将整数转换为带前缀 "0x" 前缀的小写十六进制数字符串。 如果 *integer* 不是一个 Python `int` 对象，则它必须定义返回一个整数的 `__index__()` 方法。 下面是一些例子:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

如果要将整数转换为大写或小写的十六进制字符串，并可选择有无“0x”前缀，则可以使用如下方法：

```
>>> '%#x' % 255, '%x' % 255, '%X' % 255
('0xff', 'ff', 'FF')
>>> format(255, '#x'), format(255, 'x'), format(255, 'X')
('0xff', 'ff', 'FF')
>>> f'{255:#x}', f'{255:x}', f'{255:X}'
('0xff', 'ff', 'FF')
```

另见 `format()` 获取更多信息。

另请参阅 `int()` 将十六进制字符串转换为以 16 为基数的整数。

备注： 如果要获取浮点数的十六进制字符串形式，请使用 `float.hex()` 方法。

id(*object*, /)

返回对象的“标识值”。该值是一个整数，在此对象的生命周期中保证是唯一且恒定的。两个生命期不重叠的对象可能具有相同的 `id()` 值。

这是对象在内存中的地址。

引发一个 审计事件 `builtins.id` 并附带参数 `id`。

input()
input(*prompt*, /)

如果存在 *prompt* 实参，则将其写入标准输出，末尾不带换行符。接下来，该函数从输入中读取一行，将其转换为字符串（除了末尾的换行符）并返回。当读取到 EOF 时，则触发 `EOFError`。例如:

```
>>> s = input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

如果加载了 `readline` 模块，`input()` 将使用它来提供复杂的行编辑和历史记录功能。

在读取输入前引发一个 审计事件 `builtins.input` 附带参数 `prompt`

在成功读取输入之后引发一个 审计事件 `builtins.input/result` 附带结果。

class **int**(*number=0*, /)
class **int**(*string*, /, *base=10*)

返回从一个数字或字符串构建的整数对象，或者如果未给出参数则返回 `0`。

示例:

```
>>> int(123.45)
123
>>> int('123')
123
>>> int('   -12_345\n')
-12345
>>> int('FACE', 16)
64206
>>> int('0xface', 0)
64206
>>> int('01110011', base=2)
115
```

如果参数定义了 `__int__()`，`int(x)` 返回 `x.__int__()`。 如果参数定义了 `__index__()`，则返回 `x.__index__()`。 对于浮点数，则向零截断。

如果参数不是数字或者如果给定了 *base*，则它必须是表示一个以 *base* 为基数的整数的字符串、`bytes` 或 `bytearray` 实例。 字符串前面还可选择加上 `+` 或 `-` (中间没有空格)，带有前导的零，带有两侧的空格，以及带有数位之间的单个下划线。

一个以 n 为基数的整数字符串包含多个数位，每个数位代表从 0 到 n-1 范围内的值。 0--9 的值可以用任何 Unicode 十进制数码来表示。 10--35 的值可以用 `a` 到 `z` (或 `A` 到 `Z`) 来表示。 默认的 *base* 为 10。 允许的基数为 0 和 2--36。 对于基数 2, -8 和 -16 来说字符串前面还能加上可选的 `0b/0B`, `0o/0O` 或 `0x/0X` 前缀，就像代码中的整数字面值那样。 对于基数 0 来说，字符串会以与 代码中的整数字面值 类似的方式来解读，即实际的基数将由前缀确定为 2, 8, 10 或 16。

基数为 0 还会禁用前导的零: `int('010', 0)` 将是无效的，而 `int('010')` 和 `int('010', 8)` 则是有效的。

整数类型定义请参阅 数字类型 --- int, float, complex 。

> *在 3.4 版本发生变更:* 如果 *base* 不是 `int` 的实例，但 *base* 对象有 `base.__index__` 方法，则会调用该方法来获取进制数。以前的版本使用 `base.__int__` 而不是 `base.__index__`。

> *在 3.6 版本发生变更:* 您可以使用下划线将代码文字中的数字进行分组。

> *在 3.7 版本发生变更:* 第一个形参现在是仅限位置形参。

> *在 3.8 版本发生变更:* 如果 `__int__()` 未定义则回退至 `__index__()`。

> *在 3.11 版本发生变更:* `int` 字符串输入和字符串表示形式可受到限制以帮助避免拒绝服务攻击。当将一个字符串转换为 `int` 或者将一个 `int` 转换为字符串的操作走出限制时会引发 `ValueError`。 请参阅 整数字符串转换长度限制 文档。

> *在 3.14 版本发生变更:* `int()` 不再委托 `__trunc__()` 方法

isinstance(*object*, *classinfo*, */*)

如果 *object* 参数是 *classinfo* 参数的实例，或者是其 (直接、间接或 虚拟) 子类的实例则返回 `True`。 如果 *object* 不是给定类型的对象，则该函数总是返回 `False`。 如果 *classinfo* 是由类型对象结成的元组 (或是由其他此类元组递归生成) 或者是多个类型的 union 类型，则如果 *object* 是其中任一类型的实例时将会返回 `True`。 如果 *classinfo* 不是一个类型或类型元组及此类元组，则会引发 `TypeError` 异常。 如果之前的检查成功执行则可以不会为无效的类型引发 `TypeError`。

> *在 3.10 版本发生变更:* *classinfo* 可以是一个 union 类型。

issubclass(*class*, *classinfo*, */*)

如果 *class* 是 *classinfo* 的子类（直接、间接或 虚的 ），则返回 `True`。类将视为自己的子类。*classinfo* 可为类对象的元组（或递归地，其他这样的元组）或 union 类型，这时如果 *class* 是 *classinfo* 中任何条目的子类，则返回 `True` 。任何其他情况都会触发 `TypeError` 异常。

> *在 3.10 版本发生变更:* *classinfo* 可以是一个 union 类型。

iter(*iterable*, */*)
iter(*callable*, *sentinel*, */*)

返回一个 iterator 对象。第一个参数的解释方式会根据第二个参数是否存在而完全不同：若没有提供第二个参数，则该单一参数必须是一个支持 iterable 协议（即实现了 `__iter__()` 方法）的多项集对象，或者必须支持序列协议（即实现了以从 `0` 开始的整数参数调用的 `__getitem__()` 方法）。 如果该参数既不支持可迭代协议也不支持序列协议，将会引发 `TypeError` 异常。如果提供了第二个参数 *sentinel*，则第一个参数必须是一个可调用对象。 在这种情况下创建的迭代器会在每次调用其 `__next__()` 方法时，无参数地调用该 *可调用对象*；如果返回的值等于 *sentinel*，则会引发 `StopIteration` 异常，否则将返回该值。

另请参阅 迭代器类型。

适合 `iter()` 的第二种形式的应用之一是构建块读取器。 例如，从二进制数据库文件中读取固定宽度的块，直至到达文件的末尾:

```
from functools import partial
with open('mydata.db', 'rb') as f:
    for block in iter(partial(f.read, 64), b''):
        process_block(block)
```

len(*object*, /)

返回对象的长度（元素个数）。实参可以是序列（如 string、bytes、tuple、list 或 range 等）或集合（如 dictionary、set 或 frozen set 等）。

`len` 对于大于 `sys.maxsize` 的长度如 `range(2 ** 100)` 会引发 `OverflowError`。

class **list**(*iterable=()*, /)

虽然被称为函数，`list` 实际上是一种可变序列类型，详情请参阅 列表 和 序列类型 --- list, tuple, range。

locals()

返回一个代表当前局部符号表的映射对象，以变量名称作为键，而以其当前绑定的引用作为值。

在模块作用域上，以及当附带单个命名空间使用 `exec()` 或 `eval()` 时，此函数将返回与 `globals()` 相同的命名空间。

在类作用域上，它会返回将被传给元类构造器的命名空间。

当附带不同的 local 和 global 参数使用 `exec()` 或 `eval()` 时，它将返回传入函数调用的 local 命名空间。

在上述所有情况下，在一个给定的执行帧中对 `locals()` 的每次调用都将返回 *同一个* 映射对象。 通过从 `locals()` 返回的映射对象所做的修改都将如局部变量的赋值、重新赋值或删除一样可见，而局部变量的赋值、重新赋值或删除都将立即影响所返回映射对象的内容。

在一个 optimized scope 中（包括函数、生成器和协程），每个对 `locals()` 的调用将改为返回一个新字典，其中包含函数的局部变量及任何非局部单元引用的当前绑定。 在此情况下，通过所返回字典对名称绑定的改变将 *不会* 写回到对应的局部变量或非局部单元引用，并且赋值、重新赋值或删除局部变量和非局部单元引用也 *不会* 影响之前返回的字典的内容。affect the contents of previously returned dictionaries.

将 `locals()` 作为函数、生成器或协程中的一个推导式的组成部分来调用相当于在外层作用域中调用它，不同之处在于推导式所初始化的迭代变量将被包括在内。 在其他作用域下，其行为与将推导式作为嵌套函数来运行类似。

将 `locals()` 作为生成器表达式的组成部分来调用相当于在嵌套的生成器函数中调用它。

> *在 3.12 版本发生变更:* 在推导式中的 `locals()` 的行为已被更新为符合 **PEP 709** 中的描述。

> *在 3.13 版本发生变更:* 作为 **PEP 667** 的组成部分，改变从此函数返回的映射对象的语义现在已获得定义。 在 已优化作用域 中的行为现在如上所述。 除了已获得定义，在其他作用域中的行为相比之前的版本仍然保持不变。

map(*function*, *iterable*, */*, **iterables*, *strict=False*)

返回一个将 *function* 应用于的每个 *iterable* 项目的迭代器，返回每个结果。 如果传递了额外的 *iterables* 参数，则 *function* 必须使用相同数量的参数，并将并行应用于所有 iterables 中的项目。 在多个 iterables 的情况下， 迭代器会在最短的 iterable 用尽后停止。 如果 *strict* 是 `True`，且其中一个 iterables 在其他之前耗尽，则引发 `ValueError` 。对于 function 输入已排列成参数元组的情况，请参阅 `itertools.starmap()`。

> *在 3.14 版本发生变更:* 增加了 *strict* 形参。

max(*iterable*, */*, ***, *key=None*)
max(*iterable*, */*, ***, *default*, *key=None*)
max(*arg1*, *arg2*, */*, **args*, *key=None*)

返回可迭代对象中最大的元素，或者返回两个及以上实参中最大的。

如果只提供了一个位置参数，它必须是非空 iterable，返回可迭代对象中最大的元素；如果提供了两个及以上的位置参数，则返回最大的位置参数。

有两个可选只能用关键字的实参。*key* 实参指定排序函数用的参数，如传给 `list.sort()` 的。*default* 实参是当可迭代对象为空时返回的值。如果可迭代对象为空，并且没有给 *default* ，则会触发 `ValueError`。

如果有多个最大元素，则此函数将返回第一个找到的。这和其他稳定排序工具如 `sorted(iterable, key=keyfunc, reverse=True)[0]` 和 `heapq.nlargest(1, iterable, key=keyfunc)` 保持一致。

> *在 3.4 版本发生变更:* 增加了 *default* 仅限关键字形参。

> *在 3.8 版本发生变更:* *key* 可以为 `None`。

class **memoryview**(*object*)

返回由给定实参创建的"内存视图"对象。有关详细信息，请参阅 内存视图。

min(*iterable*, */*, ***, *key=None*)
min(*iterable*, */*, ***, *default*, *key=None*)
min(*arg1*, *arg2*, */*, **args*, *key=None*)

返回可迭代对象中最小的元素，或者返回两个及以上实参中最小的。

如果只提供了一个位置参数，它必须是 iterable，返回可迭代对象中最小的元素；如果提供了两个及以上的位置参数，则返回最小的位置参数。

有两个可选只能用关键字的实参。*key* 实参指定排序函数用的参数，如传给 `list.sort()` 的。*default* 实参是当可迭代对象为空时返回的值。如果可迭代对象为空，并且没有给 *default*，则会触发 `ValueError`。

如果有多个最小元素，则此函数将返回第一个找到的。这和其他稳定排序工具如 `sorted(iterable, key=keyfunc)[0]` 和 `heapq.nsmallest(1, iterable, key=keyfunc)` 保持一致。

> *在 3.4 版本发生变更:* 增加了 *default* 仅限关键字形参。

> *在 3.8 版本发生变更:* *key* 可以为 `None`。

next(*iterator*, /)
next(*iterator*, *default*, /)

通过调用 iterator 的 `__next__()` 方法获取下一个元素。如果迭代器耗尽，则返回给定的 *default*，如果没有默认值则触发 `StopIteration`。

class **object**

这是所有其他类的终极基类。 它提供了所有 Python 类实例均具有的方法。 当其构造器被调用时，它将返回一个新的基本对象。 该构造器不接受任何参数。

> **备注:** `object` 实例 *没有* `__dict__` 属性，因此你无法将任意属性赋给 `object` 的实例。

oct(*integer*, /)

将整数转换为带前缀 "0o" 的八进制数字符串。 结果是一个合法的 Python 表达式。 如果 *integer* 不是一个 Python `int` 对象，则它必须定义返回一个整数的 `__index__()` 方法。 例如:

```
>>> oct(8)
'0o10'
>>> oct(-56)
'-0o70'
```

若要将整数转换为八进制字符串，并可选择是否带有“0o”前缀，可采用如下方法：

```
>>> '%#o' % 10, '%o' % 10
('0o12', '12')
>>> format(10, '#o'), format(10, 'o')
('0o12', '12')
>>> f'{10:#o}', f'{10:o}'
('0o12', '12')
```

另见 `format()` 获取更多信息。

open(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

打开 *file* 并返回对应的 file object。 如果该文件不能被打开，则引发 `OSError`。 请参阅 读写文件 获取此函数的更多用法示例。

file 是一个 path-like object，表示将要打开的文件的路径（绝对路径或者相对当前工作目录的路径），也可以是要封装文件对应的整数类型文件描述符。（如果给出的是文件描述符，则当返回的 I/O 对象关闭时它也会关闭，除非将 *closefd* 设为 `False` 。）

mode 是一个指明文件打开模式的可选字符串。 它默认为 `'r'` 表示以文本模式读取。 其他常见模式有表示写入的 `'w'` (若文件已存在则将其清空)，表示独占创建的 `'x'`，以及表示追加写入的 `'a'` (在 *某些* Unix 系统上，这意味着无论当前查找位置在哪里 *所有* 写入操作都将追加到文件末尾)。 在文本模式下，如果未指定 *encoding* 则所使用的编码格式将依赖于具体平台: `locale.getencoding()` 会被调用以获取当前语言区域的编码格式。 (对于读取和写入原始字节数据请使用二进制模式并且不要指定 *encoding*。) 可用的模式有:

字符	含意
`'r'`	读取（默认）
`'w'`	写入，并先截断文件
`'x'`	排它性创建，如果文件已存在则失败
`'a'`	打开文件用于写入，如果文件存在则在末尾追加
`'b'`	二进制模式
`'t'`	文本模式（默认）
`'+'`	打开用于更新（读取与写入）

默认模式为 `'r'` （打开文件用于读取文本，与 `'rt'` 同义）。`'w+'` 和 `'w+b'` 模式将打开文件并清空内容。而 `'r+'` 和 `'r+b'` 模式将打开文件但不清空内容。

正如在 概述 中提到的，Python区分二进制和文本I/O。以二进制模式打开的文件（包括 *mode* 参数中的 `'b'` ）返回的内容为 `bytes` 对象，不进行任何解码。在文本模式下（默认情况下，或者在 *mode* 参数中包含 `'t'` ）时，文件内容返回为 `str` ，首先使用指定的 *encoding* （如果给定）或者使用平台默认的的字节编码解码。

> **备注:** Python不依赖于底层操作系统的文本文件概念;所有处理都由Python本身完成，因此与平台无关。

buffering 是一个可选的整数，用于设置缓冲策略。 传入 0 来关闭缓冲（仅在二进制模式下允许），传入 1 来选择行缓冲（仅在文本模式下写入时可用），传一个整数 > 1 来表示固定大小的块缓冲区的字节大小。 注意这样指定缓冲区的大小适用于二进制缓冲的 I/O,但 `TextIOWrapper` (即用 `mode='r+'` 打开的文件) 会有另一种缓冲。 要禁用 `TextIOWrapper` 中的缓冲，请考虑为 `io.TextIOWrapper.reconfigure()` 使用 `write_through` 旗标。 当没有给出 *buffering* 参数时，默认的缓冲策略规则如下:

- 二进制文件以固定大小的块缓冲；当设备块大小可用时，缓冲区的大小就是 `max(min(blocksize, 8 MiB), DEFAULT_BUFFER_SIZE)` 。 在大多数系统中，缓冲区的长

度通常为 128KB 。
- “交互式”文本文件（ `isatty()` 返回 `True` 的文件）使用行缓冲。其他文本文件使用上述策略用于二进制文件。

encoding 是用于编码或解码文件的编码格式名称。 这应当只有文本模式下使用。 默认的编码格式依赖于具体平台 (即 `locale.getencoding()` 所返回的值)，但是任何 Python 支持的 text encoding 都可以被使用。 请参阅 `codecs` 模块获取受支持的编码格式列表。

errors 是一个可选的字符串参数，用于指定如何处理编码和解码错误 - 这不能在二进制模式下使用。可以使用各种标准错误处理程序（列在 错误处理方案 ），但是使用 `codecs.register_error()` 注册的任何错误处理名称也是有效的。标准名称包括:

- 如果存在编码错误，`'strict'` 会引发 `ValueError` 异常。 默认值 `None` 具有相同的效果。
- `'ignore'` 忽略错误。请注意，忽略编码错误可能会导致数据丢失。
- `'replace'` 会将替换标记（例如 `'?'` ）插入有错误数据的地方。
- `'surrogateescape'` 将把任何不正确的字节表示为 U+DC80 至 U+DCFF 范围内的下方替代码位。 当在写入数据时使用 `surrogateescape` 错误处理器时这些替代码位会被转回到相同的字节。 这适用于处理具有未知编码格式的文件。
- `'xmlcharrefreplace'` 仅在写入文件时才受到支持。 编码格式不支持的字符将被替换为相应的 XML 字符引用 `&#nnn;`。
- `'backslashreplace'` 用Python的反向转义序列替换格式错误的数据。
- `'namereplace'` （也只在编写时支持）用 `\N{...}` 转义序列替换不支持的字符。

newline 决定如何解析来自流的换行符。 它可以为 `None`, `''`, `'\n'`, `'\r'` 和 `'\r\n'`。 它的工作原理如下:

- 从流中读取输入时，如果 *newline* 为 `None`，则启用通用换行模式。输入中的行可以以 `'\n'`，`'\r'` 或 `'\r\n'` 结尾，这些行被翻译成 `'\n'` 在返回呼叫者之前。如果它是 `''`，则启用通用换行模式，但行结尾将返回给调用者未翻译。如果它具有任何其他合法值，则输入行仅由给定字符串终止，并且返回给调用者时行结尾不会被转换。
- 将输出写入流时，如果 *newline* 为 `None`，则写入的任何 `'\n'` 字符都将转换为系统默认行分隔符 `os.linesep`。如果 *newline* 是 `''` 或 `'\n'`，则不进行翻译。如果 *newline* 是任何其他合法值，则写入的任何 `'\n'` 字符将被转换为给定的字符串。

如果 *closefd* 为 `False` 且给出的不是文件名而是文件描述符，那么当文件关闭时，底层文件描述符将保持打开状态。如果给出的是文件名，则 *closefd* 必须为 `True` （默认值），否则将触发错误。

可以通过传递可调用的 *opener* 来使用自定义开启器。然后通过使用参数（ *file*，*flags* ）调用 *opener* 获得文件对象的基础文件描述符。 *opener* 必须返回一个打开的文件描述符（使用 `os.open` as *opener* 时与传递 `None` 的效果相同）。

新创建的文件是 不可继承的。

下面的示例使用 `os.open()` 函数的 dir_fd 的形参，从给定的目录中用相对路径打开文件:

```
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
...
>>> with open('spamspam.txt', 'w', opener=opener) as f:
...     print('This will be written to somedir/spamspam.txt', file=f)
...
>>> os.close(dir_fd)  # 不要泄漏文件描述符
```

open() 函数所返回的 file object 类型取决于所用模式。 当使用 open() 以文本模式 ('w', 'r', 'wt', 'rt' 等) 打开文件时，它将返回 io.TextIOBase (具体为 io.TextIOWrapper) 的一个子类。 当使用缓冲以二进制模式打开文件时，返回的类是 io.BufferedIOBase 的一个子类。 具体的类会有多种：在只读的二进制模式下，它将返回 io.BufferedReader；在写入二进制和追加二进制模式下，它将返回 io.BufferedWriter，而在读/写模式下，它将返回 io.BufferedRandom。 当禁用缓冲时，则会返回原始流，即 io.RawIOBase 的一个子类 io.FileIO。

另请参阅文件操作模块，如 fileinput、io （声明了 open()）、os、os.path、tempfile 和 shutil。

引发一个 审计事件 open 并附带参数 path, mode, flags。

mode 与 flags 参数可以在原始调用的基础上被修改或传递。

> *在 3.3 版本发生变更:*
>
> - 增加了 *opener* 形参。
> - 增加了 'x' 模式。
> - 过去触发的 IOError，现在是 OSError 的别名。
> - 如果文件已存在但使用了排它性创建模式（ 'x' ），现在会触发 FileExistsError。

> *在 3.4 版本发生变更:*
>
> - 文件现在禁止继承。

> *在 3.5 版本发生变更:*
>
> - 如果系统调用被中断，但信号处理程序没有触发异常，此函数现在会重试系统调用，而不是触发 InterruptedError 异常 (原因详见 **PEP 475**)。
> - 增加了 'namereplace' 错误处理接口。

> *在 3.6 版本发生变更:*
>
> - 增加对实现了 os.PathLike 对象的支持。
> - 在 Windows 上，打开一个控制台缓冲区将返回 io.RawIOBase 的子类，而不是 io.FileIO。

> *在 3.11 版本发生变更:* 'U' 模式已被移除。

ord(*character*, /)

返回字符的码序值。

如果参数是单字符字符串，则返回该字符的 Unicode 码位。例如，`ord('a')` 返回整数 `97`，`ord('€')``（欧元符号）返回 ``8364`。此函数是 `chr()` 的逆操作。

如果参数是长度为 1 的 `bytes` 或 `bytearray` 对象，则返回其单个字节值。例如，`ord(b'a')` 返回整数 `97`。

pow(*base*, *exp*, *mod=None*)

返回 *base* 的 *exp* 次幂；如果 *mod* 存在，则返回 *base* 的 *exp* 次幂对 *mod* 取余（比 `pow(base, exp) % mod` 更高效）。 两参数形式 `pow(base, exp)` 等价于乘方运算符: `base**exp`。

当参数为具有混用操作数类型的内置数字类型时，将应用针对二元算术运算符的强制转换规则。 对于 `int` 操作数，结果具有与操作数相同的类型（转换之后）除非第二个参数为负值；在那种情况下，所有参数将被转换为浮点数并输出浮点数形式的结果。 例如，`pow(10, 2)` 返回 `100`，而 `pow(10, -2)` 返回 `0.01`。 对于 `int` 或 `float` 的基数为负值而幂为非整数的情况，将产生一个复数形式的结果。 例如，`pow(-9, 0.5)` 将返回一个接近 `3j` 的值。 最后，对于 `int` 或 `float` 的基数为负值而幂为整数的情况，将产生一个浮点数形式的结果。 例如，`pow(-9, 2.0)` 将返回 `81.0`。

对于 `int` 操作数 *base* 和 *exp*，如果给出 *mod*，则 *mod* 必须为整数类型并且 *mod* 必须不为零。 如果给出 *mod* 并且 *exp* 为负值，则 *base* 必须相对于 *mod* 不可整除。 在这种情况下，将会返回 `pow(inv_base, -exp, mod)`，其中 *inv_base* 为 *base* 的倒数对 *mod* 取余。

下面的例子是 `38` 的倒数对 `97` 取余:

```
>>> pow(38, -1, mod=97)
23
>>> 23 * 38 % 97 == 1
True
```

> *在 3.8 版本发生变更:* 对于 `int` 操作数，三参数形式的 `pow` 现在允许第二个参数为负值，即可以计算倒数的余数。

> *在 3.8 版本发生变更:* 允许关键字参数。 之前只支持位置参数。

print(**objects*, *sep=' '*, *end='\n'*, *file=None*, *flush=False*)

将 *objects* 打印输出至 *file* 指定的文本流，以 *sep* 分隔并在末尾加上 *end*。 *sep* 、 *end* 、 *file* 和 *flush* 必须以关键字参数的形式给出。

所有非关键字参数都会被转换为字符串，就像是执行了 `str()` 一样，并会被写入到流，以 *sep* 分隔并在末尾加上 *end*。 *sep* 和 *end* 都必须为字符串；它们也可以为 `None`，这意味着使用默认值。 如果没有给出 *objects*，则 `print()` 将只写入 *end*。

file 参数必须是一个具有 `write(string)` 方法的对象；如果参数不存在或为 `None`，则将使用 `sys.stdout`。 由于要打印的参数会被转换为文本字符串，因此 `print()` 不能用于二进制模式的文件对象。 对于这些对象，应改用 `file.write(...)`。

输出缓冲通常由 *file* 确定。 但是，如果 *flush* 为真值，流将被强制刷新。

> *在 3.3 版本发生变更:* 增加了 *flush* 关键字参数。

class **property**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

返回 property 属性。

fget 是获取属性值的函数。 *fset* 是用于设置属性值的函数。 *fdel* 是用于删除属性值的函数。并且 *doc* 为属性对象创建文档字符串。

一个典型的用法是定义一个托管属性 `x`:

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

如果 *c* 为 *C* 的实例，`c.x` 将调用 getter，`c.x = value` 将调用 setter， `del c.x` 将调用 deleter。

如果给出，*doc* 将成为该 property 属性的文档字符串。 否则该 property 将拷贝 *fget* 的文档字符串（如果存在）。 这令使用 property() 作为 decorator 来创建只读的特征属性可以很容易地实现:

```
class Parrot:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

`@property` 装饰器会将 `voltage()` 方法转化为一个具有相同名称的只读属性 "getter"，并将 *voltage* 的文档字符串设为 "Get the current voltage."

@getter

@setter

@deleter

特征属性对象具有 `getter`, `setter` 和 `deleter` 方法，它们可用作装饰器来创建该特征属性的副本，并将相应的访问函数设为所装饰的函数。 这最好是用一个例子来说明：

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

上述代码与第一个例子完全等价。 注意一定要给附加函数与原始的特征属性相同的名称 (在本例中为 `x`。)

返回的特征属性对象同样具有与构造器参数相对应的属性 `fget`, `fset` 和 `fdel`。

> *在 3.5 版本发生变更:* 特征属性对象的文档字符串现在是可写的。

__name__

保存特征属性名称的属性。 特性属性名称可在运行时被修改。

> *Added in version 3.13.*

class **range**(*stop*, */*)
class **range**(*start*, *stop*, *step=1*, */*)

虽然被称为函数，但 `range` 实际上是一个不可变的序列类型，参见在 range 对象 与 序列类型 --- list, tuple, range 中的文档说明。

repr(*object*, */*)

返回包含一个对象的可打印表示形式的字符串。 对于许多类型而言，此函数会尝试返回一个具有与传给 `eval()` 时相同的值的字符串；在其他情况下，其表示形式将为一个包含对象类型名称和通常包括对象名称和地址的额外信息的用尖括号括起来的字符串。 一个类可以通过定义 `__repr__()` 方法来控制此函数为其实例所返回的内容。 如果 `sys.displayhook()` 不可访问，则此函数将会引发 `RuntimeError`。

该类具有自定义的表示形式，它可被求值为:

```
class Person:
   def __init__(self, name, age):
      self.name = name
      self.age = age
```

```
    def __repr__(self):
        return f"Person('{self.name}', {self.age})"
```

reversed(*object*, /)

返回一个反向的 iterator。 该参数必须是一个具有 `__reversed__()` 方法或是支持序列协议（具有 `__len__()` 方法和从 `0` 开始的整数参数的 `__getitem__()` 方法）的对象。

round(*number*, *ndigits=None*)

返回 *number* 舍入到小数点后 *ndigits* 位精度的值。 如果 *ndigits* 被省略或为 `None`，则返回最接近输入值的整数。

对于支持 `round()` 方法的内置类型，结果值会舍入至最接近的 10 的负 *ndigits* 次幂的倍数；如果与两个倍数同样接近，则选用偶数。因此，`round(0.5)` 和 `round(-0.5)` 均得出 `0` 而 `round(1.5)` 则为 `2`。*ndigits* 可为任意整数值（正数、零或负数）。如果省略了 *ndigits* 或为 `None`，则返回值将为整数。否则返回值与 *number* 的类型相同。

对于一般的 Python 对象 `number`, `round` 将委托给 `number.__round__`。

> **备注:** 对浮点数执行 `round()` 的行为可能会令人惊讶：例如，`round(2.675, 2)` 将给出 `2.67` 而不是期望的 `2.68`。 这不算是程序错误：这一结果是由于大多数十进制小数实际上都不能以浮点数精确地表示。 请参阅 浮点算术：争议和限制 了解更多信息。

class **set**(*iterable=()*, /)

返回一个新的 `set` 对象，可以选择带有从 *iterable* 获取的元素。 `set` 是一个内置类型。 请查看 `set` 和 集合类型 --- set, frozenset 获取关于这个类的文档。

有关其他容器请参看内置的 `frozenset`, `list`, `tuple` 和 `dict` 类，以及 `collections` 模块。

setattr(*object*, *name*, *value*, /)

本函数与 `getattr()` 相对应。其参数为一个对象、一个字符串和一个任意值。字符串可以为某现有属性的名称，或为新属性。只要对象允许，函数会将值赋给属性。如 `setattr(x, 'foobar', 123)` 等价于 `x.foobar = 123`。

name 无需为在 名称（标识符和关键字） 中定义的 Python 标识符除非对象选择强制这样做，例如在一个自定义的 `__getattribute__()` 中或是通过 `__slots__`。 一个名称不为标识符的属性将不可使用点号标记来访问，但是可以通过 `getattr()` 等来访问。

> **备注:** 由于 私有名称混合 发生在编译时，因此必须手动混合私有属性（以两个下划线打头的属性）名称以便使用 `setattr()` 来设置它。

class **slice**(*stop*, /)
class **slice**(*start*, *stop*, *step=None*, /)

返回一个表示由 `range(start, stop, step)` 指定的索引集的 slice 对象。 *start* 和 *step* 参数默认为 `None`。

切片对象具有只读的数据属性 `start`, `stop` 和 `step`，它们将简单地返回相应的参数值（或其默认值）。 它们没有其他显式的功能；但是，它们会被 NumPy 和其他第三方包所使用。

start

stop

step

当使用扩展索引语法时也会生成切片对象。 例如: `a[start:stop:step]` 或 `a[start:stop, i]`。 请参阅 `itertools.islice()` 了解返回 iterator 的替代版本。

> *在 3.12 版本发生变更:* Slice 对象现在将为 hashable (如果 `start`, `stop` 和 `step` 均为可哈希对象)。

sorted(*iterable*, */*, ***, *key=None*, *reverse=False*)

根据 *iterable* 中的项返回一个新的已排序列表。

具有两个可选参数，它们都必须指定为关键字参数。

key 指定带有单个参数的函数，用于从 *iterable* 的每个元素中提取用于比较的键 (例如 `key=str.lower`)。 默认值为 `None` (直接比较元素)。

reverse 为一个布尔值。 如果设为 `True`，则每个列表元素将按反向顺序比较进行排序。

使用 `functools.cmp_to_key()` 可将老式的 *cmp* 函数转换为 *key* 函数。

内置的 `sorted()` 确保是稳定的。 如果一个排序确保不会改变比较结果相等的元素的相对顺序就称其为稳定的 --- 这有利于进行多重排序（例如先按部门、再按薪级排序）。

排序算法只使用 `<` 在项目之间比较。 虽然定义一个 `__lt__()` 方法就足以进行排序，但 **PEP 8** 建议实现所有六个 富比较 。 这将有助于避免在与其他排序工具（如 `max()` ）使用相同的数据时出现错误，这些工具依赖于不同的底层方法。实现所有六个比较也有助于避免混合类型比较的混乱，因为混合类型比较可以调用反射到 `__gt__()` 的方法。

有关排序示例和简要排序教程，请参阅 排序的技术 。

@staticmethod

将方法转换为静态方法。

静态方法不会接收隐式的第一个参数。要声明一个静态方法，请使用此语法

```
class C:
    @staticmethod
    def f(arg1, arg2, argN): ...
```

`@staticmethod` 这样的形式称为函数的 decorator -- 详情参阅 函数定义。

静态方式既可以在类上调用 (如 `C.f()`)，也可以在实例上调用 (如 `C().f()`)。 此外，静态方法 descriptor 也属于可调用对象，因而它们可以在类定义中使用 (如 `f()`)。

Python 的静态方法与 Java 或 C++ 类似。另请参阅 `classmethod()` ，可用于创建另一种类构造函数。

像所有装饰器一样，也可以像常规函数一样调用 `staticmethod` ，并对其结果执行某些操作。比如某些情况下需要从类主体引用函数并且您希望避免自动转换为实例方法。对于这些情况，请使用此语法:

```
def regular_function():
    ...

class C:
    method = staticmethod(regular_function)
```

想了解更多有关静态方法的信息，请参阅 标准类型层级结构 。

> *在 3.10 版本发生变更:* 静态方法现在继承了方法的属性 (`__module__`, `__name__`, `__qualname__`, `__doc__` 和 `__annotations__`)，并具有新的 `__wrapped__` 属性，现在是属于与常规函数类似的可调用对象。

class **str**(**, encoding='utf-8', errors='strict'*)
class **str**(*object*)
class **str**(*object, encoding, errors='strict'*)
class **str**(*object, *, errors*)

返回一个 `str` 版本的 *object* 。有关详细信息，请参阅 `str()` 。

`str` 是内置字符串 class 。更多关于字符串的信息查看 文本序列类型 --- str。

sum(*iterable, /, start=0*)

从 *start* 开始自左向右对 *iterable* 的项求和并返回总计值。 *iterable* 的项通常为数字，而 start 值则不允许为字符串。

对于某些用例，存在 `sum()` 的更好替代。 拼接字符串序列的更好、更快的方式是调用 `''.join(sequence)`。 要以扩展的精度执行浮点数值的求和，请参阅 `math.fsum()`。 要拼接一系列可迭代对象，请考虑使用 `itertools.chain()`。

> *在 3.8 版本发生变更:* *start* 形参可用关键字参数形式来指定。

> *在 3.12 版本发生变更:* 浮点数的求和已切换为一种可在大多数构建版本中给出更高精确度和更好适应性的算法。

> *在 3.14 版本发生变更:* 添加了复数求和的特殊化，使用与浮点数求和相同的算法。

class **super**
class **super**(*type, object_or_type=None, /*)

返回一个代理对象，它会将方法调用委托给 *type* 的父类或兄弟类。 这对于访问已在类中被重写的继承方法很有用。

object_or_type 确定要用于搜索的 method resolution order。 搜索会从 *type* 之后的类开始。

举例来说，如果 *object_or_type* 的 `__mro__` 为 `D -> B -> C -> A -> object` 并且 *type* 的值为 `B`，则 `super()` 将会搜索 `C -> A -> object`。

对应于 *object_or_type* 的类的 `__mro__` 属性列出了 `getattr()` 和 `super()` 所共同使用的方法解析搜索顺序。 该属性是动态的并可在任何继承层级结构更新时被改变。

如果省略第二个参数，则返回的超类对象是未绑定的。 如果第二个参数为一个对象，则 `isinstance(obj, type)` 必须为真值。 如果第二个参数为一个类型，则 `issubclass(type2, type)` 必须为真值（这适用于类方法）。

当在普通方法或类中直接调用时，这两个参数均可被省略 (即 "零参数 `super()`")。 在此情况下，*type* 将为其外层的类，而 *obj* 将为其所在函数的第一个参数 (通常为 `self`)。 (这意味着零参数 `super()` 在嵌套的函数内的行为将不会如预期那样，这也包括生成器表达式，因为它会隐式地创建嵌套的函数。)

super 有两个典型用例。 在具有单继承的类层级结构中，*super* 可用来引用父类而不必显式地指定它们的名称，从而令代码更易维护。 这种用法与其他编程语言中 *super* 的用法非常相似。

第二个用例是在动态执行环境中支持协作多重继承。 此用例为 Python 所独有而不存在于静态编码语言或仅支持单继承的语言当中。 这使用实现"菱形图"成为可能，即有多个基类实现相同的方法。 好的设计强制要求这样的方法在每个情况下都具有相同的调用签名（因为调用顺序是在运行时确定的，也因为这个顺序要适应类层级结构的更改，还因为这个顺序可能包括在运行时之前未知的兄弟类）。

对于以上两个用例，典型的超类调用看起来是这样的:

```
class C(B):
    def method(self, arg):
        super().method(arg)    # 它的作用像:
                               # super(C, self).method(arg)
```

除了方法查找之外，`super()` 也可用于属性查找。 一个可能的应用场合是在上级或同级类中调用 描述器。

请注意 `super()` 被实现为为显式的带点号属性查找的绑定过程的组成部分，例如 `super().__getitem__(name)`。 它做到这一点是通过实现自己的 `__getattribute__()` 方法以便能够按支持协作多重继承的可预测的顺序来搜索类。 相应地，`super()` 在像 `super()[name]` 这样使用语句或运算符进行隐式查找时则是未定义的。

还要注意的是，除了零个参数的形式以外，`super()` 并不限于在方法内部使用。 两个参数的形式明确指定参数并进行相应的引用。 零个参数的形式仅适用于类定义内部，因为编译器需要填入必要的细节以正确地检索到被定义的类，还需要让普通方法访问当前实例。

对于有关如何使用 `super()` 来如何设计协作类的实用建议，请参阅 使用 super() 的指南。

> *在 3.14 版本发生变更:* `super` 对象现在是 `pickleable` 和 `copyable` 。

class **tuple**(*iterable=()*, */*)

虽然被称为函数，但 `tuple` 实际上是一个不可变的序列类型，参见在 元组 与 序列类型 --- list, tuple, range 中的文档说明。

class **type**(*object*, */*)
class **type**(*name*, *bases*, *dict*, */*, ***kwargs*)

传入一个参数时，返回 *object* 的类型。 返回值是一个 type 对象并且通常与 `object.__class__` 所返回的对象相同。

推荐使用 `isinstance()` 内置函数来检测对象的类型，因为它会考虑子类的情况。

传入三个参数时，返回一个新的 type 对象。 这在本质上是 `class` 语句的一种动态形式。 *name* 字符串即类名并会成为 `__name__` 属性；*bases* 元组包含基类并会成为 `__bases__` 属性；如果为空，则会添加所有类的终极基类，即 `object`。 *dict* 字典包含类体的属性和方法定义；它在成为 `__dict__` 属性之前可能会被拷贝或包装。 下面两条语句会创建同样的 `type` 对象：

```
>>> class X:
...     a = 1
...
>>> X = type('X', (), dict(a=1))
```

另请参阅：

- 有关类的属性和方法的文档。
- 类型对象

提供给三参数形式的关键字参数会被传递给适当的元类机制 (通常为 `__init_subclass__()`)，相当于类定义中关键字 (除了 *metaclass*) 的行为方式。

另请参阅 自定义类创建。

> *在 3.6 版本发生变更:* `type` 的子类如果未重写 `type.__new__` 将不再能使用一个参数的形式来获取对象的类型。

vars()
vars(*object*, */*)

返回模块、类、实例或任何其他具有 `__dict__` 属性的对象的 `__dict__` 属性。

模块和实例这样的对象具有可更新的 `__dict__` 属性；但是，其他对象的 `__dict__` 属性可能会设置写入限制（例如，类会使用 `types.MappingProxyType` 来防止直接更新字典）。

不带参数时，`vars()` 的行为将类似于 `locals()`。

如果指定了一个对象但它没有 `__dict__` 属性（例如，当它所属的类定义了 `__slots__` 属性时）则会引发 `TypeError` 异常。

> *在 3.13 版本发生变更:* 不带参数调用此函数的结果已被更新为与 `locals()` 内置函数的描述类似。

zip(**iterables*, *strict=False*)

在多个迭代器上并行迭代，从每个迭代器返回一个数据项组成元组。

示例:

```
>>> for item in zip([1, 2, 3], ['sugar', 'spice', 'everything nice']):
...     print(item)
...
(1, 'sugar')
(2, 'spice')
(3, 'everything nice')
```

更正式的说法： zip() 返回元组的迭代器，其中第 *i* 个元组包含的是每个参数迭代器的第 *i* 个元素。

不妨换一种方式认识 zip() ：它会把行变成列，把列变成行。这类似于 矩阵转置 。

zip() 是延迟执行的：直至迭代时才会对元素进行处理，比如 for 循环或放入 list 中。

值得考虑的是，传给 zip() 的可迭代对象可能长度不同；有时是有意为之，有时是因为准备这些对象的代码存在错误。Python 提供了三种不同的处理方案：

- 默认情况下，zip() 在最短的迭代完成后停止。较长可迭代对象中的剩余项将被忽略，结果会裁切至最短可迭代对象的长度：

  ```
  >>> list(zip(range(3), ['fee', 'fi', 'fo', 'fum']))
  [(0, 'fee'), (1, 'fi'), (2, 'fo')]
  ```

- 通常 zip() 用于可迭代对象等长的情况下。这时建议用 `strict=True` 的选项。输出与普通的 zip() 相同：。

  ```
  >>> list(zip(('a', 'b', 'c'), (1, 2, 3), strict=True))
  [('a', 1), ('b', 2), ('c', 3)]
  ```

  与默认行为不同，如果一个可迭代对象在其他几个之前被耗尽则会引发 ValueError:

  ```
  >>> for item in zip(range(3), ['fee', 'fi', 'fo', 'fum'], strict=True):
  ...     print(item)
  ...
  (0, 'fee')
  (1, 'fi')
  (2, 'fo')
  Traceback (most recent call last):
    ...
  ValueError: zip() argument 2 is longer than argument 1
  ```

  如果未指定 `strict=True` 参数，所有导致可迭代对象长度不同的错误都会被抑制，这可能会在程序的其他地方表现为难以发现的错误。

- 为了让所有的可迭代对象具有相同的长度，长度较短的可用常量进行填充。这可由 itertools.zip_longest() 来完成。

极端例子是只有一个可迭代对象参数，`zip()` 会返回一个一元组的迭代器。如果未给出参数，则返回一个空的迭代器。

小技巧：

- 可确保迭代器的求值顺序是从左到右的。这样就能用 `zip(*[iter(s)]*n, strict=True)` 将数据列表按长度 n 进行分组。这将重复 *相同* 的迭代器 `n` 次，输出的每个元组都包含 `n` 次调用迭代器的结果。这样做的效果是把输入拆分为长度为 n 的块。
- `zip()` 与 `*` 运算符相结合可以用来拆解一个列表：

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> list(zip(x, y))
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zip(x, y))
>>> x == list(x2) and y == list(y2)
True
```

> *在 3.10 版本发生变更:* 增加了 `strict` 参数。

__import__(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)

> **备注:** 与 `importlib.import_module()` 不同，这是一个日常 Python 编程中不需要用到的高级函数。

此函数会由 `import` 语句唤起。 它可以被替换 (通过导入 `builtins` 模块并赋值给 `builtins.__import__`) 以便修改 `import` 语句的语义，但是 **强烈** 不建议这样做，因为使用导入钩子 (参见 **PEP 302**) 通常更容易实现同样的目标，并且不会导致代码问题，因为许多代码都会假定所用的是默认实现。 同样也不建议直接使用 `__import__()` 而应该用 `importlib.import_module()`。

本函数会导入模块 *name*，利用 *globals* 和 *locals* 来决定如何在包的上下文中解释该名称。 *fromlist* 给出了应从 *name* 模块中导入的对象或子模块的名称。标准的实现代码完全不会用到 *locals* 参数，只用到了 *globals* 用于确定 `import` 语句所在的包上下文。

level 指定是使用绝对还是相对导入。 `0` (默认值) 意味着仅执行绝对导入。 *level* 为正数值表示相对于模块调用 `__import__()` 的目录，将要搜索的父目录层数 (详情参见 **PEP 328**)。

当 *name* 变量的形式为 `package.module` 时，通常将会返回最高层级的包（第一个点号之前的名称），而 *不是* 以 *name* 命名的模块。 但是，当给出了非空的 *fromlist* 参数时，则将返回以 *name* 命名的模块。

例如，语句 `import spam` 的结果将为与以下代码作用相同的字节码：

```
spam = __import__('spam', globals(), locals(), [], 0)
```

语句 `import spam.ham` 的结果将为以下调用：

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

请注意在这里 `__import__()` 是如何返回顶层模块的，因为这是通过 `import` 语句被绑定到特定名称的对象。

另一方面，语句 `from spam.ham import eggs, sausage as saus` 的结果将为

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], 0)
eggs = _temp.eggs
saus = _temp.sausage
```

在这里， `spam.ham` 模块会由 `__import__()` 返回。 要导入的对象将从此对象中提取并赋值给它们对应的名称。

如果您只想按名称导入模块（可能在包中），请使用 `importlib.import_module()`

> *在 3.3 版本发生变更:* *level* 的值不再支持负数（默认值也修改为0）。

> *在 3.9 版本发生变更:* 当使用了命令行参数 `-E` 或 `-I` 时，环境变量 `PYTHONCASEOK` 现在将被忽略。

备注

[1] 解析器只接受 Unix 风格的行结束符。如果您从文件中读取代码，请确保用换行符转换模式转换 Windows 或 Mac 风格的换行符。