

3. 数据模型

3.1. 对象、值与类型

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

每个对象都有相应的标识号、类型和值。一个对象被创建后它的标识号就绝不会改变；你可以将其理解为该对象在内存中的地址。[is](#) 运算符比较两个对象的标识号是否相同；[id\(\)](#) 函数返回一个代表其标识号的整数。

在 CPython 中，`id(x)` 就是存放 `x` 的内存的地址。

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

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

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

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

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

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

有些对象包含对其他对象的引用；它们被称为容器。容器的例子有元组、列表和字典等。这些引用是容器对象值的组成部分。在多数情况下，当谈论一个容器的值时，我们是指所包含对象的值而不是其编号；但是，当我们谈论一个容器的可变性时，则仅指其直接包含的对象的编号。因此，如果

一个不可变容器 (例如元组) 包含对一个可变对象的引用, 则当该可变对象被改变时容器的值也会改变。

类型会影响对象行为的几乎所有方面。甚至对象标识号的重要性也在某种程度上受到影响: 对于不可变类型, 计算新值的操作实际上可能返回一个指向具有相同类型和值的任何现存对象的引用, 而对于可变对象来说这是不允许的。例如在 `a = 1; b = 1` 之后, `a` 和 `b` 可能会也可能不会指向同一个值为 1 的对象。这是因为 `int` 是不可变对象, 因此对 1 的引用可以被重用。此行为依赖于所使用的具体实现, 因此不应该依赖它, 而在使用对象标识测试时需要注意。不过, 在 `c = []; d = []` 之后, `c` 和 `d` 保证会指向两个不同的、独特的、新创建的空列表。(注意 `e = f = []` 会将 同一个对象同时赋值给 `e` 和 `f`。)

3.2. 标准类型层级结构

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

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

3.2.1. None

此类型只有一种取值。是一个具有此值的单独对象。此对象通过内置名称 `None` 访问。在许多情况下它被用来表示空值, 例如未显式指明返回值的函数将返回 `None`。它的逻辑值为假。

3.2.2. NotImplemented

此类型只有一种取值。是一个具有该值的单独对象。此对象通过内置名称 `NotImplemented` 访问。数值方法和丰富比较方法如未实现指定运算符表示的运算则应返回该值。(解释器会根据具体运算符继续尝试反向运算或其他回退操作。) 它不应被解读为布尔值。

详情参见 [实现算术运算](#)。

在 3.9 版本发生变更: 在布尔运算上下文中对 `NotImplemented` 求值的做法已被弃用。

在 3.14 版本发生变更: 在布尔运算上下文中对 `NotImplemented` 求值现在会引发 `TypeError`。在之前版本中它会被求值为 `True` 并将从 Python 3.9 起发出 `DeprecationWarning`。

3.2.3. Ellipsis

此类型只有一种取值。是一个具有此值的单独对象。此对象通过字面值 `...` 或内置名称 `Ellipsis` 访问。它的逻辑值为真。

3.2.4. [numbers.Number](#)

此类对象由数字字面值创建, 并会被作为算术运算符和算术内置函数的返回结果。数字对象是不可变的; 一旦创建其值就不再改变。Python 中的数字当然非常类似数学中的数字, 但也受限于计算机

中的数字表示方法。

数字类的字符串表示形式，由 `__repr__()` 和 `__str__()` 算出，具有以下特征属性：

- 它们是有效的数字字面值，当被传给它们的类构造器时，将会产生具有原数字值的对象。
- 表示形式会在可能的情况下采用 10 进制。
- 开头的零，除小数点前可能存在的单个零之外，将不会被显示。
- 末尾的零，除小数点后可能存在的单个零之外，将不会被显示。
- 正负号仅在当数字为负值时会被显示。

Python 区分整型数、浮点型数和复数：

3.2.4.1. [numbers.Integral](#)

此类对象表示数学中整数集合的成员 (包括正数和负数)。

备注： 整型数表示规则的目的是在涉及负整型数的变换和掩码运算时提供最为合理的解释。

整型数可细分为两种类型：

整型 ([int](#))

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

布尔型 ([bool](#))

此类对象表示逻辑值 False 和 True。代表 False 和 True 值的两个对象是唯二的布尔对象。布尔类型是整型的子类型，两个布尔值在各种场合的行为分别类似于数值 0 和 1，例外情况只有在转换为字符串时分别返回字符串 "False" 或 "True"。

3.2.4.2. [numbers.Real](#) ([float](#))

此类对象表示机器级的双精度浮点数。其所接受的取值范围和溢出处理将受制于底层的机器架构 (以及 C 或 Java 实现)。Python 不支持单精度浮点数；支持后者通常的理由是节省处理器和内存消耗，但这点节省相对于在 Python 中使用对象的开销来说太过微不足道，因此没有理由包含两种浮点数而令该语言变得复杂。

3.2.4.3. [numbers.Complex](#) ([complex](#))

此类对象以一对机器级的双精度浮点数来表示复数值。有关浮点数的附带规则对其同样有效。一个复数值 `z` 的实部和虚部可通过只读属性 `z.real` 和 `z.imag` 来获取。

3.2.5. 序列

这些代表以非负数为索引的有限有序集合。内置函数 `len()` 将返回序列的项数。当序列的长度为 `n` 时，索引集合将包含数字 `0, 1, ..., n-1`。`a[i]` 是选择序列 `a` 中的第 `i` 项。某些序列，包括内置的序列，可通过加上序列长度来解读负下标值。例如，`a[-2]` 等价于 `a[n-2]`，即长度为 `n` 的 `a` 序列的倒数第二项。

序列还支持切片: `a[i:j]` 是选择索引为 k 使得 $i \leq k < j$ 的所有条目。当用作表达式时, 切片就是一个相同类型的新序列。以上有关负索引的注释也适用于切片位置的负值。

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

序列可根据其可变性来加以区分:

3.2.5.1. 不可变序列

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

以下类型属于不可变对象:

字符串

字符串是由代表 Unicode 码位的值组成的序列。取值范围在 `U+0000 - U+10FFFF` 之内的所有码位都可在字符串中使用。Python 没有 `char` 类型; 而是将字符串中的每个码位表示为一个长度为 1 的字符串对象。内置函数 `ord()` 可将一个码位由字符串形式转换为取值范围在 `0 - 10FFFF` 之内的整数; `chr()` 可将一个取值范围在 `0 - 10FFFF` 之内的整数转换为长度为 1 的对应字符串对象。`str.encode()` 可以使用给定的文本编码格式将 `str` 转换为 `bytes`, 而 `bytes.decode()` 则可以被用来实现相反的解码操作。

元组

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

字节串

字节串对象是不可变的数组。其中的条目是 8 比特位的字节, 以取值范围 $0 \leq x < 256$ 内的整数表示。字节串字面值 (如 `b'abc'`) 和内置的 `bytes()` 构造器可被用来创建字节串对象。并且, 字节串对象还可通过 `decode()` 方法被解码为字符串。

3.2.5.2. 可变序列

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

备注: `collections` 和 `array` 模块提供了可变序列类型的更多例子。

目前有两种内生可变序列类型:

列表

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

字节数组

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

3.2.6. 集合类型

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

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

目前有两种内生集合类型:

集合

这些对象代表可变集合。它们可通过内置 [set\(\)](#) 构造器创建并且在此之后可通过某些方法来修改, 例如 [add](#)。

冻结集合

此类对象表示不可变集合。它们可通过内置的 [frozenset\(\)](#) 构造器创建。由于 frozenset 对象不可变且 [hashable](#), 它可以被用作另一个集合的元素或是字典的键。

3.2.7. 映射

此类对象表示由任意索引集合所索引的对象的集合。通过下标 `a[k]` 可在映射 `a` 中选择索引为 `k` 的条目; 这可以在表达式中使用, 也可作为赋值或 [del](#) 语句的目标。内置函数 [len\(\)](#) 可返回一个映射中的条目数。

目前只有一种内生映射类型:

3.2.7.1. 字典

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

字典会保留插入顺序, 这意味着键将以它们被添加的顺序在字典中依次产生。替换某个现有的键不会改变其顺序, 但是移除某个键再重新插入则会将其添加到末尾而不会保留其原有位置。

字典是可变对象; 它们可通过 `{}` 标注方式来创建 (参见 [字典显示](#) 一节)。

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

在 3.7 版本发生变更: 在 Python 3.6 版之前字典不会保留插入顺序。在 CPython 3.6 中插入顺序会被保留, 但这在当时被当作是一个实现细节而非确定的语言特性。

3.2.8. 可调用类型

此类型可以被应用于函数调用操作 (参见 [调用](#) 小节):

3.2.8.1. 用户定义函数

用户定义函数对象可通过函数定义来创建 (参见 [函数定义](#) 小节)。它被调用时应附带一个参数列表, 其中包含的条目应与函数所定义的形参列表一致。

3.2.8.1.1. 特殊的只读属性

属性	含意
function.__globals__	对存放该函数中 全局变量 的 字典 的引用 —— 函数定义所在模块的全局命名空间。
function.__closure__	<div>None 或单元的 tuple, 其中包含了名称在函数的 代码对象 的 co_freevars 中对指定名称的绑定。</div> <div>单元对象具有 cell_contents 属性。这可被用来获取以及设置单元的值。</div>

3.2.8.1.2. 特殊的可写属性

这些属性大多会检查赋值的类型:

属性	含意
function.__doc__	函数的文档字符串, 或者如果不可用则为 None。
function.__name__	函数的名称。另请参阅: __name__ 属性。
function.__qualname__	<div>函数的 qualified name。另请参阅: __qualname__ 属性。</div> <div>Added in version 3.3.</div>
function.__module__	该函数所属模块的名称, 没有则为 None。
function.__defaults__	由具有默认值的形参的默认 parameter 值组成的 tuple , 或者如果无任何形参具有默认值则为 None。
function.__code__	代表已编译的函数体的 代码对象 。
function.__dict__	命名空间支持任意函数属性。另请参阅: __dict__ 属性。
function.__annotations__	<div>一个包含 形参 标注的 字典。该字典的键是形参名称, 以及表示返回标注的 'return', 如有提供的话。另请参阅: object.__annotations__。</div> <div>在 3.14 版本发生变更: 标记现在将被 惰性求值。参见 PEP 649。</div>

属性	含意
<code>function.__annotate__</code>	针对该函数的 annotate function ，或者如果函数没有标注则为 <code>None</code> 。参见 object.__annotate__ 。 <i>Added in version 3.14.</i>
<code>function.__kwdefaults__</code>	包含仅限关键字 形参 默认值的 字典 。
<code>function.__type_params__</code>	包含 泛型函数 类型形参 的 tuple 。 <i>Added in version 3.12.</i>

函数对象也支持获取和设置任意属性，举例来说，这可被用于将元数据关联到函数。通常使用带点号的属性标注来获取和设置这样的属性。

CPython 目前的实现仅支持用户自定义函数上的函数属性。未来可能会支持 [内置函数](#) 上的函数属性。

有关函数定义的额外信息可以从其 [代码对象](#) 中提取（可通过 [__code__](#) 属性来访问）。

3.2.8.2. 实例方法

实例方法用于结合类、类实例和任何可调用对象（通常为用户定义函数）。

特殊的只读属性：

<code>method.__self__</code>	指向方法所 绑定 的类实例对象。
<code>method.__func__</code>	指向原本的 函数对象
<code>method.__doc__</code>	方法的文档（等同于 method.__func__.__doc__ ）。如果原始函数具有文档字符串则为一个 字符串 ，否则为 <code>None</code> 。
<code>method.__name__</code>	方法名称（与 method.__func__.__name__ 相同）
<code>method.__module__</code>	方法定义所在模块的名称，如不可用则为 <code>None</code> 。

方法还支持读取（但不能设置）下层 [函数对象](#) 的任意函数属性。

用户自定义方法对象可在获取一个类的属性（可能是通过该类的实例）时被创建，如果该属性是一个用户自定义 [函数对象](#) 或 [classmethod](#) 对象的话。

当通过从类的实例获取一个用户自定义 [函数对象](#) 的方式创建一个实例方法对象时，该方法对象的 [__self__](#) 属性即为该实例，而该方法对象将被称作已 [绑定](#)。该新建方法的 [__func__](#) 属性将是原来的函数对象。

当通过从类或实例获取一个 [classmethod](#) 对象的方式创建一个实例方法对象时，该对象的 [__self__](#) 属性即为该类本身，而其 [__func__](#) 属性将是类方法对应的下层函数对象。

当一个实例方法被调用时，会调用对应的下层函数 ([__func__](#))，并将类实例 ([__self__](#)) 插入参数列表的开头。例如，当 `c` 是一个包含 `f()` 函数定义的类，而 `x` 是 `c` 的一个实例，则调用 `x.f(1)` 就等价于调用 `c.f(x, 1)`。

当一个实例方法对象是派生自一个 [classmethod](#) 对象时，保存在 [__self__](#) 中的“类实例”实际上会是该类本身，因此无论是调用 `x.f(1)` 还是 `c.f(1)` 都等同于调用 `f(c, 1)`，其中 `f` 为对应的下层函数。

需要重点关注的是作为类实例的属性的用户自定义函数不会被转换为绑定方法；这 只会在函数是类的属性时才会发生。

3.2.8.3. 生成器函数

一个使用 [yield](#) 语句（见 [yield 语句](#) 章节）的函数或方法被称为 *生成器函数*。这样的函数在被调用时，总是返回一个可以执行该函数体的 [iterator](#) 对象：调用该迭代器的 [iterator.__next__\(\)](#) 方法将导致这个函数一直运行到它使用 `yield` 语句提供一个值。当这个函数执行 [return](#) 语句或到达函数体末尾时，将引发 [StopIteration](#) 异常并且该迭代器将到达所返回的值集合的末尾。

3.2.8.4. 协程函数

使用 [async def](#) 来定义的函数或方法就被称为 *协程函数*。这样的函数在被调用时会返回一个 [coroutine](#) 对象。它可能包含 [await](#) 表达式以及 [async with](#) 和 [async for](#) 语句。详情可参见 [协程对象](#) 一节。

3.2.8.5. 异步生成器函数

使用 [async def](#) 来定义并使用了 [yield](#) 语句的函数或方法被称为 *异步生成器函数*。这样的函数在被调用时，将返回一个 [asynchronous iterator](#) 对象，该对象可在 [async for](#) 语句中被用来执行函数体。

调用异步迭代器的 [aiterator.__anext__](#) 方法将返回一个 [awaitable](#)，此对象会在被等待时执行直到使用 [yield](#) 产生一个值。当函数执行到空的 [return](#) 语句或函数末尾时，将会引发 [StopAsyncIteration](#) 异常并且异步迭代器也将到达要产生的值集合的末尾。

3.2.8.6. 内置函数

内置函数是针对特定 C 函数的包装器。内置函数的例子包括 [len\(\)](#) 和 [math.sin\(\)](#) 等 ([math](#) 是一个标准内置模块)。参数的数量和类型是由 C 函数确定的。特殊的只读属性：

- [__doc__](#) 是函数的文档字符串，或者如果不可用则为 `None`。参见 [function.__doc__](#)。
- [__name__](#) 是函数的名称。参见 [function.__name__](#)。
- [__self__](#) 被设为 `None` (但请参见下一项)。
- [__module__](#) 是函数定义所在模块的名称，或者如果不可用则为 `None`。参见 [function.__module__](#)。

3.2.8.7. 内置方法

此类型实际上是内置函数的另一种形式，只不过还包含了一个转入 C 函数的对象作为隐式的额外参数。内置方法的一个例子是 `alist.append()`，其中 `alist` 是一个列表对象。在此示例中，特殊的只读属性 `__self__` 会被设为 `alist` 所标记的对象。（该属性的语义与 [其他实例方法](#) 的相同。）

3.2.8.8. 类

类是可调对象。这些对象通常是用作创建自身实例的“工厂”，但类也可以有重载 `__new__()` 的变体类型。调用的参数会传递给 `__new__()`，并且在通常情况下，也会传递给 `__init__()` 来初始化新的实例。

3.2.8.9. 类实例

任意类的实例可以通过在其所属类中定义 `__call__()` 方法变成可调对象。

3.2.9. 模块

模块是 Python 代码的基本组织单元，由 [导入系统](#) 创建，它或是通过 `import` 语句，或是通过调用 `importlib.import_module()` 和内置的 `__import__()` 等函数来唤起。模块对象具有通过 [字典](#) 对象实现的命名空间（就是被定义在模块中的函数的 `__globals__` 属性所引用的字典）。属性引用将被转换为在该字典中的查找操作，例如 `m.x` 就等价于 `m.__dict__["x"]`。模块对象不包含用于初始化模块的代码对象（因为初始化完成后已不再需要它）。

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

3.2.9.1. 模块对象上与导入相关的属性

模块对象具有下列与 [导入系统](#) 相关的属性。当使用关联到导入系统的机制创建模块时，这些属性将在 [loader](#) 执行和加载模块之前基于模块的 [规格](#) 来填充。

要动态创建模块而非使用导入系统创建，建议使用 `importlib.util.module_from_spec()`，它会将各种由导入控制的属性设置为适当的值。也可以使用 `types.ModuleType` 构造器直接创建模块，但这种方法更容易出错，因为在使用这种方法时，必须在创建模块对象后手动设置其大部分属性。

小心： 对 `__name__` 以外的用例，**强烈** 建议使用 `__spec__` 及其属性，而非此处列出的其他单独属性。请注意更新 `__spec__` 上的属性时不会连带更新模块本身的同名属性。

```
>>> import typing
>>> typing.__name__, typing.__spec__.name
('typing', 'typing')
>>> typing.__spec__.name = 'spelling'
>>> typing.__name__, typing.__spec__.name
('typing', 'spelling')
>>> typing.__name__ = 'keyboard_smashing'
>>> typing.__name__, typing.__spec__.name
('keyboard_smashing', 'spelling')
```

`module.__name__`

用于在导入系统中唯一地标识模块的名称。对于直接执行的模块，这将被设为 `"__main__"`。

该属性必须被设为模块的完整限定名称。它应当与 `module.__spec__.name` 的值相匹配。

`module.__spec__`

模块与导入系统相关联的状态的记录。

设置为导入模块时使用的 [模块规格](#)。请参阅 [模块规格说明](#) 了解详情。

Added in version 3.4.

`module.__package__`

模块所属的 [package](#)。

如果一个模块是顶层模块（不是任何包的一部分），该属性应被设置为空字符串 `''`。否则，它应被设置为模块所属包的名字（如果模块本身是一个包，它可以是 `module.__name__` 的值）。详见 [PEP 366](#)。

在为主模块计算显式相对导入时，这个属性而非 `__name__` 被使用。在使用 [types.ModuleType](#) 构造器动态创建模块时会被默认设为 `None`。要确保这个属性是 `str`，请使用 [importlib.util.module_from_spec\(\)](#)。

强烈 建议使用 `module.__spec__.parent` 而非 `module.__package__`。`__package__` 现在仅作 `__spec__.parent` 未被设置时的回退路径，且这条回退路径已被弃用。

在 3.4 版本发生变更： 对于使用 [types.ModuleType](#) 构造器动态创建的模块，该属性现在默认为 `None`。先前该属性是可选的。

在 3.6 版本发生变更： `__package__` 的值应与 `__spec__.parent` 相同。`__package__` 现在仅作导入解析期间 `__spec__.parent` 未被定义时的回退值。

在 3.10 版本发生变更： 如果导入解析回退到 `__package__` 而非 `__spec__.parent`，会引发 [ImportWarning](#)。

在 3.12 版本发生变更： 在导入解析期间回退到 `__package__` 时会引发 [DeprecationWarning](#) 而非 [ImportWarning](#)。

Deprecated since version 3.13, will be removed in version 3.15: `__package__` 将不再被设置或者被导入系统或标准库纳入考量。

`module.__loader__`

导入系统用来加载模块的 [loader](#) 对象。

该属性主要适用于自省，但也可用于额外的加载器专属功能，例如获取与加载器相关联的数据。

对于使用 [types.ModuleType](#) 构造器动态创建的模块 `__loader__` 默认为 `None`；请改用 [importlib.util.module_from_spec\(\)](#) 来确保将该属性设为 [loader](#) 对象。

强烈 建议你使用 `module.__spec__.loader` 来代替 `module.__loader__`。

在 3.4 版本发生变更: 对于使用 [types.ModuleType](#) 构造器动态创建的模块, 该属性现在默认为 `None`。先前该属性是可选的。

Deprecated since version 3.12, will be removed in version 3.16: 当设置 `__spec__.loader` 失败时在模块上设置 `__loader__` 的做法已被弃用。在 Python 3.16 中, `__loader__` 将不会再被设置或被导入系统或标准库纳入考虑。

`module.__path__`

一个 (可能为空的) 枚举用于查找包的子模块的位置的由字符串组成的 [sequence](#)。非包模块应当没有 `__path__` 属性。详情参见 [模块的 `__path__` 属性](#)。

强烈 建议你使用 [module.__spec__.submodule_search_locations](#) 来代替 `module.__path__`。

`module.__file__`

`module.__cached__`

`__file__` 和 `__cached__` 都是可设也可不设的可选属性。两个属性在可用时都应当为 [str](#)。

`__file__` 指明要载入的模块所在文件的路径名 (如果是从文件载入), 或者对于从共享库动态载入的扩展模块来说则是共享库文件的路径名。它对于特定类型的模块来说可能是缺失的, 例如静态链接到解释器中的 C 模块, 并且 [导入系统](#) 也可能在它没有语法意义时选择不设置它 (例如, 当一个模块是从数据库导入时)。

如果设置了 `__file__` 则 `__cached__` 属性也可能被设置, 它是指向任何代码的已编译版本的路径 (例如, 一个字节码文件)。设置此属性并不需要存在相应的路径; 该路径可以简单地指向已编译文件 将要 存在的位置 (参见 [PEP 3147](#))。

请注意 `__cached__` 即使在未设置 `__file__` 时也可能被设置。不过, 那样的场景是非典型的。最终, [loader](#) 会是 [finder](#) (from which `__file__` 和 `__cached__` 也是自它派生的) 所提供的模块规格的使用方。因此如果一个加载器可以从缓存加载模块但是不能从文件加载, 那样的非典型场景就是适当的。

强烈 建议你使用 [module.__spec__.cached](#) 来代替 `module.__cached__`。

Deprecated since version 3.13, will be removed in version 3.15: 当设置 `__spec__.cached` 失败时在模块上设置 `__cached__` 的做法已被弃用。在 Python 3.15 中, `__cached__` 将不会再被设置或被导入系统或标准库纳入考虑。

3.2.9.2. 模块对象上的其他可写属性

除了上面列出的导入相关属性, 模块对象还具有下列可写属性:

`module.__doc__`

模块的文档字符串, 或者如果不可用则为 `None`。另请参阅: [__doc__ 属性](#)。

`module.__annotations__`

一个包含在模块体执行期间收集的 [变量标注](#) 的字典。有关使用 `__annotations__` 的最佳实践, 请参阅 [annotationlib](#)。

在 3.14 版本发生变更: 标记现在将被 [惰性求值](#)。参见 [PEP 649](#)。

module.__annotate__

针对该模块的 [annotate function](#)，或者如果模块没有标注则为 None。另请参阅：[__annotate__](#) 属性。

Added in version 3.14.

3.2.9.3. 模块字典

模块对象还具有以下特殊的只读属性：Module objects also have the following special read-only attribute:

module.__dict__

以字典对象表示的模块命名空间。在此处列出的属性中它很特别，__dict__ 不能从模块内部作为全局变量来访问；它只能作为模块对象上的属性来访问。

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

3.2.10. 自定义类

自定义类这种类型一般是通过类定义来创建 (参见 [类定义](#) 一节)。每个类都有一个通过字典对象实现的命名空间。类属性引用会被转化为在此字典中查找，例如，C.x 会被转化为 C.__dict__["x"] (不过也存在一些钩子对象允许其他定位属性的方式)。当未在其中找到某个属性名称时，会继续在基类中查找。这种基类搜索使用 C3 方法解析顺序，即使存在 '钻石形' 继承结构既有多条继承路径连到一个共同祖先也能保持正确的行为。有关 Python 使用的 C3 MRO 的详情可在 [Python 2.3 方法解析顺序](#) 查看。

当一个类属性引用 (假设类名为 c) 会产生一个类方法对象时，它将转化为一个 [__self__](#) 属性为 c 的实例方法对象。当它会产生一个 [staticmethod](#) 对象时，它将转换为该静态方法对象所包装的对象。有关有类的 [__dict__](#) 实际包含内容以外获取属性的其他方式请参阅 [实现描述器](#) 一节。

类属性赋值会更新类的字典，但不会更新基类的字典。

类对象可被调用 (见上文) 以产生一个类实例 (见下文)。

3.2.10.1. 特殊属性

属性	含意
type.__name__	类的名称。另请参阅: __name__ 属性。
type.__qualname__	类的 qualified name 。另请参阅: __qualname__ 属性。
type.__module__	类定义所在模块的名称。
type.__dict__	一个提供类的命名空间的只读视图的 映射代理 。另请参阅: __dict__ 属性。

属性	含意
type.__bases__	一个包含类的基类的 tuple ，对于定义为 <code>class X(A, B, C)</code> 的类， <code>X.__bases__</code> 将等于 <code>(A, B, C)</code> 。
type.__base__	继承链中负责实例的内存布局的单独基类。该属性对应于 C 层级上的 tp_base 。
type.__doc__	类的文档字符串，如果未定义则为 <code>None</code> 。不会被子类继承。 if undefined. Not inherited by subclasses.
type.__annotations__	<p>一个包含在类体执行期间收集的 变量标注 的字典。另请参阅: __annotations__ attributes。</p> <p>有关使用 __annotations__ 的最佳实践，请参阅 annotationlib。请使用 annotationlib.get_annotations() 而不是直接该属性。</p> <div> <p>警告： 直接在类对象上访问 <code>__annotations__</code> 属性可能会返回错误类的注解，特别是在某些情况下，类、它的基类或元类是在 <code>from __future__ import annotations</code> 下定义的。详细信息请参见 <749#pep749-metaclasses>。</p> <p>此属性在某些内置类中不存在。对于不带 <code>__annotations__</code> 的用户定义类，它是一个空字典。</p> </div> <p>在 3.14 版本发生变更: 标记现在将被 惰性求值。参见 PEP 649。</p>
type.__annotate__()	<p>针对该类的 annotate function，或者如果类没有标注则为 <code>None</code>。另请参阅: __annotate__ 属性。</p> <p>Added in version 3.14.</p>
type.__type_params__	<p>一个包含 泛型类 的 类型形参 的 tuple。</p> <p>Added in version 3.12.</p>
type.__static_attributes__	<p>一个包含由通过 <code>self.X</code> 赋值为该类语句体中任何函数的类属性名称组成的 tuple。</p> <p>Added in version 3.13.</p>
type.__firstlineno__	类定义的第一行的行号，包括装饰器。设置 __module__ 属性将从类型的字典中移除 <code>__firstlineno__</code> 条目。

属性	含意
	<div>Added in version 3.13.</div>
<code>type.__mro__</code>	由在方法解析期间当查找基类时将被纳入考虑的类组成的 tuple 。

3.2.10.2. 特殊方法

除了上面介绍的特殊属性，所有的 Python 类还具有以下两个方法：

`type.mro()`

此方法可由一个元类来重写以便为其实例定制方法解析顺序。它会在类实例化时被调用，其结果将存储在 `__mro__` 中。

`type.__subclasses__()`

每个类都会保存一个由指向其直接子类的弱引用组成的列表。此方法将返回一个由所有仍然存在的这种引用组成的列表。列表项将按定义顺序排列。例如：

```
>>> class A: pass
>>> class B(A): pass
>>> A.__subclasses__()
[<class 'B'>]
```

3.2.11. 类实例

类实例可通过调用类对象来创建（见上文）。每个类实例都有通过一个字典对象实现的独立命名空间，属性引用会首先在此字典中进行查找。当未在其中发现某个属性，而实例对应的类中有该属性时，会继续在类属性中查找。如果找到的类属性是一个用户自定义函数对象，它会被转化为实例方法对象，其 `__self__` 属性即该实例。静态方法和类方法对象也会被转化；参见上文的“类”小节。要了解其他通过类实例来获取相应类属性的方式请参阅 [实现描述器](#) 小节，这样得到的属性可能与实际存放在类的 `__dict__` 中的对象不同。如果未找到类属性，而对象所属的类具有 `__getattr__()` 方法，则会调用该方法来满足查找要求。

属性赋值和删除会更新实例的字典，但绝不会更新类的字典。如果类具有 `__setattr__()` 或 `__delattr__()` 方法，则将调用该方法而不再直接更新实例的字典。

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

3.2.11.1. 特殊属性

`object.__class__`

类实例所属的类。

`object.__dict__`

一个用于存储对象的（可写）属性的字典或其他映射对象。并非所有实例都具有 `__dict__` 属性；请参阅 [slots](#) 章节了解详情。

3.2.12. I/O 对象 (或称文件对象)

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

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

3.2.13. 内部类型

某些由解释器内部使用的类型也被暴露给用户。它们的定义可能随未来解释器版本的更新而变化, 为内容完整起见在此处一并介绍。

3.2.13.1. 代码对象

代码对象表示 编译为字节的可执行 Python 代码, 或称 [bytecode](#)。代码对象和函数对象的区别在于函数对象包含对函数全局对象 (函数所属的模块) 的显式引用, 而代码对象不包含上下文; 而且默认参数值会存放于函数对象而不是代码对象内 (因为它们表示在运行时算出的值)。与函数对象不同, 代码对象不可变, 也不包含对可变对象的引用 (不论是直接还是间接)。

3.2.13.1.1. 特殊的只读属性

<code>codeobject.co_name</code>	函数名
<code>codeobject.co_qualname</code>	完整限定函数名 <div>Added in version 3.11.</div>
<code>codeobject.co_argcount</code>	函数的位置 形参 的总数 (包括仅限位置形参和具有默认值的形参)
<code>codeobject.co_posonlyargcount</code>	函数的仅限位置 形参 的总数 (包括具有默认值的参数)
<code>codeobject.co_kwonlyargcount</code>	函数的仅限关键字 形参 的数量 (包括具有默认值的参数)
<code>codeobject.co_nlocals</code>	函数使用的 局部变量 的数量 (包括形参)
<code>codeobject.co_varnames</code>	一个 tuple , 其中包含函数中局部变量的名称 (从形参名称开始)
<code>codeobject.co_cellvars</code>	包含被函数内至少一个 nested scope 所引用的 局部变量 的名称的 tuple 。
<code>codeobject.co_freevars</code>	一个 tuple , 其中包含某个 nested scope 引用在外部作用域中引用的 containing the names of 自由 (闭

	包) 变量 的名称。另请参阅 function.__closure__ 。 注意：对全局和内置名称的引用 不会被包括在内。
<code>codeobject.co_code</code>	一个表示函数中的 bytecode 指令序列的字符串
<code>codeobject.co_consts</code>	一个包含函数中的 bytecode 所使用的字面值的 tuple
<code>codeobject.co_names</code>	一个包含函数中的 bytecode 所使用的名称的 tuple
<code>codeobject.co_filename</code>	被编译代码所在文件的名称
<code>codeobject.co_firstlineno</code>	函数第一行所对应的行号
<code>codeobject.co_lnotab</code>	一个编码了从 bytecode 偏移量到行号的映射的字符串。要获取更多细节，请查看解释器的源代码。 <div style="border-left: 2px solid red; padding-left: 10px; margin-left: 10px;"> 自 3.12 版本弃用：代码对象的这个属性已被弃用，并可能在 Python 3.15 中移除。 </div>
<code>codeobject.co_stacksize</code>	需要的代码对象栈大小
<code>codeobject.co_flags</code>	用于对一系列解释器旗标进行编码的 整数 。

以下是针对 [co_flags](#) 定义的旗标位：如果函数使用 `*arguments` 语法来接受任意数量的位置参数则设置 `0x04` 位；如果函数使用 `**keywords` 语法来接受任意数量的关键字参数则设置 `0x08` 位；如果函数是一个生成器则设置 `0x20` 位。请参阅 [代码对象位标志](#) 可能出现的每个旗标的语义详情。

未来特性声明 (例如 `from __future__ import division`) 也使用 [co_flags](#) 中的比特位来指明代码对象在编译时是否启用了某个特性。参见 [compiler_flag](#)。

[co_flags](#) 中的其他位被保留供内部使用。

如果一个代码对象代表函数并且具有文档字符串，则会在 [co_flags](#) 中设置 [CO_HAS_DOCSTRING](#) 比特位并且 [co_consts](#) 中的第一个条目将是该函数的文档字符串。

3.2.13.1.2. 代码对象的方法

`codeobject.co_positions()`

返回一个包含代码对象中每条 [bytecode](#) 指令的源代码位置的可迭代对象。

此迭代器返回包含 `(start_line, end_line, start_column, end_column)` 的 [tuple](#)。其中第 i 个元组冲锋衣官方编译为第 i 个代码单元的源代码的位置。列信息是给定源代码行从 0 开始索引的 utf-8 字节偏移量。

此位置信息可能会丢失。可能发生这种情况下非详尽列表如下：

- 附带 `-X no_debug_ranges` 运行解释器。
- 在使用 `-X no_debug_ranges` 时加载一个已编译的 pyc 文件。

- 与人工指令相对应的位置元组。
- 由于具体实现专属的限制而无法表示的行号和列号。

当发生此情况时，元组的部分或全部元素可以为 [None](#)。

Added in version 3.11.

备注： 此特性需要在代码对象中存储列位置，这可能会导致编译的 `which` may result in a small increase of disk usage of compiled Python 文件占用的磁盘空间或解释器占用的内存略有增加。要避免存储额外信息和/或取消打印额外的回溯信息，可以使用 `-X no_debug_ranges` 命令行旗标或 `PYTHONNODEBUGRANGES` 环境变量。

`codeobject.co_lines()`

返回一个产生有关 [bytecode](#) 的连续范围的信息的迭代器。其产生的每一项都是一个 `(start, end, lineno)` [tuple](#)：

- `start` (一个 [int](#)) 代表相对于 [bytecode](#) 范围开始位置的偏移量 (不包括该位置)。
- `end` ([int](#) 值) 代表相对于 [bytecode](#) 范围末尾位置的偏移量 (不包括该位置)。
- `lineno` 是一个代表 [bytecode](#) 范围内的行号的 [int](#)，或者如果给定范围内的字节码没有行号则为 `None`。

产生的条目将具有下列特征属性：

- 产出的第一个范围将以 0 作为 `start`。
- `(start, end)` 范围将是非递减和连续的。也就是说，对于任何一对 [tuple](#)，第二个的 `start` 将等于第一个的 `end`。
- 任何范围都不会是反向的：对于所有三元组均有 `end >= start`。
- 产生的最后一个 [tuple](#) 的 `end` 将等于 [bytecode](#) 的大小。

零宽度范围，即 `start == end` 也是允许的。零宽度范围的使用场景是源代码中存在，但被 [bytecode](#) 编译器所去除的那些行。

Added in version 3.10.

参见：

[PEP 626](#) - 在调试和其他工具中使用精确的行号。

引入 `co_lines()` 方法的 PEP。

`codeobject.replace(**kwargs)`

返回代码对象的一个副本，使用指定的新字段值。

代码对象也被泛型函数 [copy.replace\(\)](#) 所支持。

Added in version 3.8.

3.2.13.2. 帧对象

帧对象表示执行帧。它们可能出现在 [回溯对象](#) 中，还会被传递给已注册的跟踪函数。

3.2.13.2.1. 特殊的只读属性

<code>frame.f_back</code>	指向前一个栈帧（对于调用方而言），或者如果这是最底部的栈帧则为 <code>None</code>
<code>frame.f_code</code>	该帧中正在执行的 代码对象 。访问该属性将引发一个 审计事件 <code>object.__getattr__</code> ，附带参数 <code>obj</code> 和 <code>"f_code"</code> 。
<code>frame.f_locals</code>	被该帧用来查找 局部变量 的映射。如果该帧指向一个 optimized scope ，这可能返回一个直通写入代理对象。 <div>在 3.13 版本发生变更: 返回一个已优化作用域的代理。</div>
<code>frame.f_globals</code>	被帧用于查找 全局变量 的字典
<code>frame.f_builtins</code>	被帧用于查找 内置（内建）名称 的字典
<code>frame.f_lasti</code>	帧对象的“准确指令”（这是 代码对象 的 bytecode 字符串的索引）
<code>frame.f_generator</code>	拥有该帧的 generator 或 coroutine ，或者如果该帧属于常规函数则为 <code>None</code> 。 <div>Added in version 3.14.</div>

3.2.13.2.2. 特殊的可写属性

<code>frame.f_trace</code>	如果不为 <code>None</code> ，则是在代码执行期间调用各类事件的函数（由调试器使用）。通常每个新的源代码行会触发一个事件（参见 f_trace_lines ）。
<code>frame.f_trace_lines</code>	将该属性设为 False 以禁用为每个源代码行触发跟踪事件。
<code>frame.f_trace_opcodes</code>	将该属性设为 True 以允许请求每个操作码事件。请注意如果跟踪函数引发的异常逃逸到被跟踪的函数中这可能会导致未定义的解释器行为。
<code>frame.f_lineno</code>	该帧的当前行号 -- 在这里写入从一个跟踪函数内部跳转到的给定行（仅用于最底层的帧）。调试器可以通过写入该属性实现一个 <code>Jump</code> 命令（即设置下一条语句）。

3.2.13.2.3. 帧对象方法

帧对象支持一个方法:

`frame.clear()`

此方法将清除该帧持有的全部对 [局部变量](#) 的引用。并且，如果该帧归属于一个 [generator](#)，此生成器将被终结。这有助于打破涉及帧对象的循环引用（例如当捕获一个 [异常](#) 并保存其 [回溯](#)

供以后使用)。

如果该帧当前正在执行或已挂起则会引发 [RuntimeError](#)。

Added in version 3.4.

在 3.13 版本发生变更: 尝试清除已挂起的帧将引发 [RuntimeError](#) (执行帧的情况将总是如此)。

3.2.13.3. 回溯对象

回溯对象代表一个 [异常](#) 的栈跟踪信息。当异常发生时隐式地创建一个回溯对象，也可以通过调用 [types.TracebackType](#) 显式地创建。

在 3.7 版本发生变更: 现在回溯对象可以通过 Python 代码显式地实例化。

对于隐式地创建的回溯对象，当查找异常处理器使得执行栈展开时，会在每个展开层级的当前回溯之前插入一个回溯对象。当进入一个异常处理器时，程序将可以使用栈跟踪。（参见 [try 语句](#) 一节。）它可作为 [sys.exc_info\(\)](#) 所返回的元组的第三项，以及所捕获异常的 [__traceback__](#) 属性被获取。

当程序不包含适用的处理器时，栈跟踪会（以良好的格式）写入到标准错误流；如果解释器处于交互模式，它也将作为 [sys.last_traceback](#) 供用户使用。

对于显式地创建的回溯对象，则由回溯对象的创建者来决定应该如何连接 [tb_next](#) 属性以构成完整的线跟踪。

特殊的只读属性：

<code>traceback.tb_frame</code>	指向当前层级的执行 帧对象 。 访问该属性将引发一个 审计事件 <code>object.__getattr__</code> ，附带参数 <code>obj</code> 和 <code>"tb_frame"</code> 。
<code>traceback.tb_lineno</code>	给出异常发生所在的行号
<code>traceback.tb_lasti</code>	表示“精确指令”。

回溯中的行号和最后一条指令可能与其 [帧对象](#) 的行号不同，如果异常发生在 [try](#) 语句中且没有匹配的 `except` 子句或是有 [finally](#) 子句的话。

`traceback.tb_next`

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

在 3.7 版本发生变更: 该属性现在是可写的。

3.2.13.4. 切片对象

切片对象被用来表示 [__getitem__\(\)](#) 方法所使用的切片。该对象也可使用内置的 [slice\(\)](#) 函数来创建。

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

切片对象支持一个方法:

`slice.indices(self, length)`

此方法接受一个整型参数 `length` 并计算在切片对象被应用到 `length` 指定长度的条目序列时切片的相关信息应如何描述。其返回值为三个整型数组成的元组; 这些数分别为切片的 `start` 和 `stop` 索引号以及 `step` 步长值。索引号缺失或越界则按照与正规切片相一致的方式处理。

3.2.13.5. 静态方法对象

静态方法对象提供了一种胜过上文所述将函数对象转换为方法对象的方式。静态方法对象是对任何其他对象的包装器, 通常用来包装用户自定义的方法对象。当从类或类实例获取一个静态方法对象时, 实际返回的是经过包装的对象, 它不会被进一步转换。静态方法对象也是可调用对象。静态方法对象可通过内置的 [staticmethod\(\)](#) 构造器来创建。

3.2.13.6. 类方法对象

类方法对象与静态方法类似, 是对其他对象的包装器, 会改变从类或类实例获取该对象的方式。类方法对象在这种获取操作中的行为已在上文中描述, 见 "[实例方法](#)" 一节。类方法对象是通过内置 [classmethod\(\)](#) 构造器创建的。

3.3. 特殊方法名称

一个类可以通过定义具有特殊名称的方法来实现由特殊语法来唤起的特定操作 (例如算术运算或抽取与切片)。这是 Python 实现 [运算符重载](#) 的方式, 允许每个类自行定义基于该语言运算符的特定行为。举例来说, 如果一个类定义了名为 [__getitem__\(\)](#) 的方法, 并且 `x` 是该类的一个实例, 则 `x[i]` 基本就等价于 `type(x).__getitem__(x, i)`。除非有说明例外情况, 在没有定义适当方法的时候尝试执行某种操作将引发一个异常 (通常为 [AttributeError](#) 或 [TypeError](#))。

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

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

3.3.1. 基本定制

`object.__new__(cls[, ...])`

调用以创建一个 `cls` 类的新实例。[__new__\(\)](#) 是一个静态方法 (因为是特例所以你不需要显式地声明), 它会将所请求实例所属的类作为第一个参数。其余的参数会被传递给对象构造器表达式

(对类的调用)。[__new__\(\)](#) 的返回值应为新对象实例 (通常是 *cls* 的实例)。

典型的实现会附带适当的参数使用 `super().__new__(cls[, ...])` 通过唤起超类的 [__new__\(\)](#) 方法来创建一个新的类实例然后在返回它之前根据需要修改新创建的实例。

如果 [__new__\(\)](#) 在构造对象期间被唤起并且它返回了一个 *cls* 的实例, 则新实例的 [__init__\(\)](#) 方法将以 `__init__(self[, ...])` 的形式被唤起, 其中 *self* 为新实例而其余的参数与被传给对象构造器的参数相同。

如果 [__new__\(\)](#) 未返回一个 *cls* 的实例, 则新实例的 [__init__\(\)](#) 方法就不会被执行。

[__new__\(\)](#) 的目的主要是允许不可变类型的子类 (例如 `int`, `str` 或 `tuple`) 定制实例创建过程。它也会常在自定义元类中被重载以便定制类创建过程。

`object.__init__(self[, ...])`

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

```
super().__init__([args...]).
```

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

`object.__del__(self)`

在实例将被销毁时调用。这还被称为终结器或析构器 (不适当)。如果一个基类具有 [__del__\(\)](#) 方法, 则其所派生的类如果也有 [__del__\(\)](#) 方法, 就必须显式地调用它以确保实例基类部分的正确清除。

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

当解释器退出时并不保证会为仍然存在的对象调用 [__del__\(\)](#) 方法。[weakref.finalize](#) 提供了一种直观的方式来注册当对象被作为垃圾回收时要调用的清理函数。

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

一个引用循环可以阻止对象的引用计数归零。在这种情况下, 循环将稍后被检测到并被 [循环垃圾回收器](#) 删除。导致引用循环的一个常见原因是当一个异常在局部变量中被捕获。帧的局部变量将会引用该异常, 这将引用它自己的回溯信息, 它会又引用在回溯中捕获的所有帧的局部变量。

参见: [gc](#) 模块的文档。

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

- `del__()` 可在任意代码被执行时启用，包括来自任意线程的代码。如果 `del__()` 需要接受锁或启用其他阻塞资源，可能会发生死锁，例如该资源已被为执行 `del__()` 而中断的代码所获取。
- `del__()` 可以在解释器关闭阶段被执行。因此，它需要访问的全局变量（包含其他模块）可能已被删除或设为 `None`。Python 会保证先删除模块中名称以单个下划线打头的全局变量再删除其他全局变量；如果已不存在其他对此类全局变量的引用，这有助于确保导入的模块在 `del__()` 方法被调用时仍然可用。

`object.__repr__(self)`

由 `repr()` 内置函数调用以输出一个对象的“官方”字符串表示。如果可能，这应类似一个有效的 Python 表达式，能被用来重建具有相同取值的对象（只要有适当的环境）。如果这不可能，则应返回形式如 `<...some useful description...>` 的字符串。返回值必须是一个字符串对象。如果一个类定义了 `__repr__()` 但未定义 `__str__()`，则在需要该类的实例的“非正式”字符串表示时也会使用 `__repr__()`。

此方法通常被用于调试，因此确保其表示的内容包含丰富信息且无歧义是很重要的。`object` 类本身提供了一个默认实现。

`object.__str__(self)`

由 `str(object)`，默认的 `__format__()` 实现以及内置函数 `print()` 调用，以生成一个对象的“非正式”或适合打印的字符串表示形式。返回值必须为一个 `str` 对象。

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

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

`object.__bytes__(self)`

由 `bytes` 调用以生成一个对象的字节串表示形式。这应当返回一个 `bytes` 对象。`object` 类本身不提供此方法。

`object.__format__(self, format_spec)`

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

请参看 [格式规格迷你语言](#) 了解标准格式化语法的描述。

返回值必须为一个字符串对象。

应当为由 `object` 类提供的默认实现给出一个空的 `format_spec` 字符串。它将委托给 `__str__()`。

在 3.4 版本发生变更: `object` 本身的 `__format__` 方法如果被传入任何非空字符, 将会引发一个 `TypeError`。

在 3.7 版本发生变更: `object.__format__(x, '')` 现在等同于 `str(x)` 而不再是 `format(str(x), '')`。

```
object.__lt__(self, other)
object.__le__(self, other)
object.__eq__(self, other)
object.__ne__(self, other)
object.__gt__(self, other)
object.__ge__(self, other)
```

以上这些被称为“富比较”方法。运算符与方法名称的对应关系如下: `x<y` 调用 `x.__lt__(y)`、`x<=y` 调用 `x.__le__(y)`、`x==y` 调用 `x.__eq__(y)`、`x!=y` 调用 `x.__ne__(y)`、`x>y` 调用 `x.__gt__(y)`、`x>=y` 调用 `x.__ge__(y)`。

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

在默认情况下, `object` 通过使用 `is` 来实现 `__eq__()`, 并在比较结果为假值时返回 `NotImplemented`: `True if x is y else NotImplemented`。对于 `__ne__()`, 默认会委托给 `__eq__()` 并对结果取反, 除非结果为 `NotImplemented`。比较运算符之间没有其他隐含关系或默认实现; 例如, `(x<y or x==y)` 为真并不意味着 `x<=y`。要根据单根运算自动生成排序操作, 请参看 `functools.total_ordering()`。

在默认情况下, `object` 类提供与 [值比较](#) 一致的实现: 相等比较是根据对象标识号, 而顺序比较会引发 `TypeError`。每个默认方法都可能直接生成这样的结果, 但也可能返回 `NotImplemented`。

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

这些方法都没有对调参数版本 (在左边参数不支持该操作但右边参数支持时使用); 而是 `__lt__()` 和 `__gt__()` 互为对方的反向, `__le__()` 和 `__ge__()` 互为对方的反射, 而 `__eq__()` 和 `__ne__()` 则是它们自己的反射。如果两个操作数的类型不同, 且右操作数的类型是左操作数类型的直接或间接子类, 则优先选择右操作数的反射方法, 在其他情况下优先选择左操作数的方法。虚拟子类化不会被考虑。

当没有合适的方法返回任何 `NotImplemented` 以外的值时, `==` 和 `!=` 运算符将分别回退至 `is` 和 `is not`。

```
object.__hash__(self)
```

通过内置函数 `hash()` 调用以对哈希集的成员进行操作, 属于哈希集的类型包括 `set`、`frozenset` 以及 `dict`。`__hash__()` 应该返回一个整数。对象比较结果相同所需的唯一特征属

性是其具有相同的哈希值；建议的做法是把参与比较的对象全部组件的哈希值混在一起，即将它们打包为一个元组并对该元组做哈希运算。例如：

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

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

如果一个类没有定义 `__eq__()` 方法，那么它也不应该定义 `__hash__()` 操作；如果它定义了 `__eq__()` 但没有定义 `__hash__()`，则其实例将不可被用作可哈希多项集的条目。如果一个类定义了可变对象并实现了 `__eq__()` 方法，则它不应该实现 `__hash__()`，因为 `hashable` 多项集的实现要求键的哈希值是不可变的（如果对象的哈希值发生改变，它将位于错误的哈希桶中）。

用户自定义的类默认带有 `__eq__()` 和 `__hash__()` 方法（继承自 `object` 类）；因为它们的存在，所有对象（自己除外）相互比较必定不相等并且 `x.__hash__()` 将返回一个恰当的值以使得 `x == y` 同时意味着 `x is y` 且 `hash(x) == hash(y)`。

一个类如果重载了 `__eq__()` 且没有定义 `__hash__()` 则会将其 `__hash__()` 隐式地设为 `None`。当一个类的 `__hash__()` 方法为 `None` 时，该类的实例将在一个程序尝试获取其哈希值时正确地引发 `TypeError`，并会在检测 `isinstance(obj, collections.abc.Hashable)` 时被正确地识别为不可哈希对象。

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

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

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

这是为了防止通过精心选择输入来利用字典插入操作在最坏情况下的执行效率即 $O(n^2)$ 复杂度制度的拒绝服务攻击。请参阅 <http://ocert.org/advisories/ocert-2011-003.html> 了解详情。

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

另见 [PYTHONHASHSEED](#)。

在 3.3 版本发生变更: 默认启用哈希随机化。

`object.__bool__(self)`

调用此方法以实现真值检测以及内置的 `bool()` 操作; 应当返回 `False` 或 `True`。当未定义此方法时, 则会在定义了 `__len__()` 的情况下调用它, 如果其结果不为零则该对象将被视为具有真值。如果一个类的 `__len__()` 和 `__bool__()` 均未定义 (这也是 `object` 类本身的情况), 则其所有实例都将被视为具有真值。

3.3.2. 自定义属性访问

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

`object.__getattr__(self, name)`

当默认属性访问因引发 `AttributeError` 而失败时被调用 (可能是调用 `__getattribute__()` 时由于 `name` 不是一个实例属性或 `self` 的类层级树中的属性而引发了 `AttributeError`; 或者是由于 `name` 特征属性的 `__get__()` 引发了 `AttributeError`)。此方法应当返回 (找到的) 属性值或是引发一个 `AttributeError` 异常。 `object` 类本身没有提供此方法。

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

`object.__getattribute__(self, name)`

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

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

对于特定的敏感属性访问, 引发一个 [审计事件](#) `object.__getattr__`, 附带参数 `obj` 和 `name`。

`object.__setattr__(self, name, value)`

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

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

对特定敏感属性的赋值，会引发一个 [审计事件](#) `object.__setattr__`，附带参数 `obj`, `name`, `value`。

`object.__delattr__(self, name)`

类似于 [__setattr__\(\)](#) 但其作用为删除而非赋值。此方法应该仅在 `del obj.name` 对于该对象有意义时才被实现。

对于特定的敏感属性删除，引发一个 [审计事件](#) `object.__delattr__`，附带参数 `obj` 和 `name`。

`object.__dir__(self)`

此方法会在针对相应对象调用 [dir\(\)](#) 时被调用。返回值必须为一个可迭代对象。[dir\(\)](#) 会把返回的可迭代对象转换为列表并对其排序。

3.3.2.1. 自定义模块属性访问

`module.__getattr__()`

`module.__dir__()`

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

`__dir__` 函数应当不接受任何参数，并且返回一个表示模块中可访问名称的字符串可迭代对象。此函数如果存在，将会重写一个模块中的标准 [dir\(\)](#) 搜索操作。

`module.__class__`

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

```
import sys
from types import ModuleType

class VerboseModule(ModuleType):
    def __repr__(self):
        return f'Verbose {self.__name__}'

    def __setattr__(self, attr, value):
        print(f'Setting {attr}...')
        super().__setattr__(attr, value)

sys.modules[__name__].__class__ = VerboseModule
```

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

在 3.5 版本发生变更: `__class__` 模块属性改为可写。

Added in version 3.7: `__getattr__` 和 `__dir__` 模块属性。

参见:

[PEP 562](#) - 模块 `__getattr__` 和 `__dir__`

描述用于模块的 `__getattr__` 和 `__dir__` 函数。

3.3.2.2. 实现描述器

下列方法仅当一个包含该方法的类（即所谓 *描述器* 类）的实例出现在一个 *所有者* 类之中的时候才会起作用（该描述器必须在所有者类或它的某个上级类的类字典中）。在下面的例子中，“属性”是指名称为所有者类的 `__dict__` 中的特征属性的键名的属性。`object` 类本身没有实现这些协议。

`object.__get__(self, instance, owner=None)`

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

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

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

`object.__set__(self, instance, value)`

调用此方法以设置 `instance` 指定的所有者类的实例的属性为新值 `value`。

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

`object.__delete__(self, instance)`

调用此方法以删除 `instance` 指定的所有者类的实例的属性。

描述器的实例也可能存在 `__objclass__` 属性：

`object.__objclass__`

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

3.3.2.3. 调用描述器

总的说来，描述器就是具有“绑定行为”的对象属性，其属性访问已被描述器协议中的方法所重载：

`__get__()`、`__set__()` 和 `__delete__()`。如果一个对象定义了以上方法中的任意一个，它就被称为描述器。

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

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

描述器唤起的开始点是一个绑定 `a.x`。参数的组合方式依 `a` 而定：

直接调用

最简单但最不常见的调用方式是用户代码直接唤起一个描述器方法：`x.__get__(a)`。

实例绑定

如果绑定到一个对象实例，`a.x` 会被转换为调用：`type(a).__dict__['x'].__get__(a, type(a))`。

类绑定

如果绑定到一个类，`A.x` 会被转换为调用：`A.__dict__['x'].__get__(None, A)`。

超绑定

类似 `super(A, a).x` 这样的带点号查找将在 `a.__class__.__mro__` 中搜索紧接在 `A` 之后的基类 `B` 并返回 `B.__dict__['x'].__get__(a, A)`。如果 `x` 不是描述器，则不加改变地返回它。

对于实例绑定，发起描述器调用的优先级取决于定义了哪些描述器方法。一个描述器可以定义 `__get__()`、`__set__()` 和 `__delete__()` 的任意组合。如果它没有定义 `__get__()`，则访问属性将返回描述器对象自身，除非对象的实例字典中有相应的属性值。如果描述器定义了 `__set__()` 和/或 `__delete__()`，则它是一个数据描述器；如果两者均未定义，则它是一个非数据描述器。通常，数据描述器会同时定义 `__get__()` 和 `__set__()`，而非数据描述器则只有 `__get__()` 方法。定义了 `__get__()` 和 `__set__()` (和/或 `__delete__()`) 的数据描述器总是会重载实例字典中的定义。与之相对地，非数据描述器则可被实例所重载。

Python 方法（包括用 `@staticmethod` 和 `@classmethod` 装饰的方法）都是作为非数据描述器来实现的。因而，实例可以重定义和重写方法。这允许单个实例获得与相同类的其他实例不一样的行为。

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

3.3.2.4. `__slots__`

`__slots__` 允许我们显式地声明数据成员（如特征属性）并禁止创建 `__dict__` 和 `__weakref__`（除非是在 `__slots__` 中显式地声明或是在父类中可用。）

相比使用 `__dict__` 可以显著节省空间。属性查找速度也可得到显著的提升。

`object.__slots__`

这个类变量可赋值为字符串、可迭代对象或由实例使用的变量名组成的字符串序列。`__slots__` 会为已声明的变量保留空间并阻止自动为每个实例创建 `__dict__` 和 `__weakref__`。

使用 `__slots__` 的注意事项:

- 当继承自一个没有 `__slots__` 的类时, 实例的 `__dict__` 和 `__weakref__` 属性将总是可访问的。
- 没有 `__dict__` 变量, 实例就不能给未在 `__slots__` 定义中列出的新变量赋值。尝试给一个未列出的变量名赋值将引发 `AttributeError`。如果需要动态地给新变量赋值, 则要将 `'__dict__'` 加入到在 `__slots__` 中声明的字符串序列中。
- 如果未给每个实例设置 `__weakref__` 变量, 则定义了 `__slots__` 的类就不支持对其实例的 [弱引用](#)。如果需要在支持弱引用, 则要将 `'__weakref__'` 加入到在 `__slots__` 中声明的字符串序列中。
- `__slots__` 是通过为每个变量名创建 [描述器](#) 在类层级上实现的。因此, 类属性不能被用来为通过 `__slots__` 定义的实例变量设置默认值; 否则, 类属性将会覆盖描述器赋值。
- `__slots__` 声明的作用不只限于定义它的类。在父类中声明的 `__slots__` 在其子类中同样可用。不过, 子类的实例将会获得 `__dict__` 和 `__weakref__`, 除非子类也定义了 `__slots__` (它应当只包含 [附加槽位](#) 的名称)。
- 如果一个类定义的位置在某个基类中也有定义, 则由基类位置定义的实例变量将不可访问 (除非通过直接从基类获取其描述器的方式)。这会使得程序的含义变成未定义。未来可能会添加一个防止此情况的检查。
- 如果为派生自 ["variable-length" 内置类型](#) 如 `int`, `bytes` 和 `tuple` 的类定义了非空的 `*__slots__*` 则将引发 `TypeError`。
- 任何非字符串的 [iterable](#) 都可以被赋值给 `__slots__`。
- 如果是使用一个 [字典](#) 来给 `__slots__` 赋值, 则该字典的键将被用作槽位名称。字典的值可被用来为每个属性提供将被 `inspect.getdoc()` 识别并在 `and displayed in the output of help()` 的输出中显示的文档字符串。
- `__class__` 赋值仅在两个类具有相同的 `__slots__` 时才会起作用。
- 带有多槽位父类的 [多重继承](#) 也是可用的, 但仅允许一个父类具有由槽位创建的属性 (其他基类必须具有空的槽位布局) —— 违反此规则将引发 `TypeError`。
- 如果将 [iterator](#) 用于 `__slots__` 则会将该迭代器的每个值创建一个 [descriptor](#)。但是, `__slots__` 属性将为一个空迭代器。

3.3.3. 自定义类创建

当一个类继承另一个类时, 会在这个父类上调用 `__init_subclass__()`。这样, 就使得编写改变子类行为的类成为可能。这与类装饰器有很密切的关联, 但类装饰器只能影响它们所应用的特定类, 而 `__init_subclass__` 则只作用于定义了该方法的类在未来的子类。

`classmethod object.__init_subclass__(cls)`

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

传给一个新类的关键字参数会被传给上级类的 `__init_subclass__`。为了与其他使用 `__init_subclass__` 的类兼容, 应当去掉需要的关键字参数再将其他参数传给基类, 例如:

```
class Philosopher:
    def __init_subclass__(cls, /, default_name, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.default_name = default_name
```

```
class AustralianPhilosopher(Philosopher, default_name="Bruce"):
    pass
```

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

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

Added in version 3.6.

当一个类被创建时，`type.__new__()` 会扫描类变量并对其中带有 `__set_name__()` 钩子的对象执行回调。

`object.__set_name__(self, owner, name)`

在所有者类 `owner` 被创建时自动调用。此对象已被赋值给该类中的 `name`：

```
class A:
    x = C() # 自动调用: x.__set_name__(A, 'x')
```

如果类变量赋值是在类被创建之后进行的，`__set_name__()` 将不会被自动调用。如有必要，可以直接调用 `__set_name__()`：

```
class A:
    pass

c = C()
A.x = c # 钩子未被调用
c.__set_name__(A, 'x') # 手动唤起钩子
```

详情参见 [创建类对象](#)。

Added in version 3.6.

3.3.3.1. 元类

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

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

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

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

当一个类定义被执行时，将发生以下步骤：

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

3.3.3.2. 解析 MRO 条目

`object.__mro_entries__(self, bases)`

如果一个出现在类定义中的基类不是 [type](#) 的实例，则会在该基类中搜索 `__mro_entries__()` 方法。如果找到了 `__mro_entries__()` 方法，则在创建类时该基类会被替换为调用 `__mro_entries__()` 的结果。该方法被调用时将附带传给 `bases` 形参的原始基类元组，并且必须返回一个由将被用来替代该基类的类组成的元组。返回的元组可能为空：在此情况下，原始基类将被忽略。

参见：

[types.resolve_bases\(\)](#)

动态地解析不属于 [type](#) 实例的基类。

[types.get_original_bases\(\)](#)

在类被 [__mro_entries__\(\)](#) 修改之前提取其“原始基类”。

[PEP 560](#)

对 `typing` 模块和泛用类型的核心支持。

3.3.3.3. 确定适当的元类

为一个类定义确定适当的元类是根据以下规则：

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

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

3.3.3.4. 准备类命名空间

一旦确定了适当的元类，就开始准备类的命名空间。如果元类具有 `__prepare__` 属性，它将以 `namespace = metaclass.__prepare__(name, bases, **kwargs)` 的形式被调用（其中如果存在任何额外关键字参数，则应来自类定义）。`__prepare__` 方法应当被实现为 [类方法](#)。`__prepare__`

所返回的命名空间会被传入 `__new__`，但是当最终的类对象被创建时该命名空间会被拷贝到一个新的 `dict` 中。

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

参见:

[PEP 3115](#) - Python 3000 中的元类

引入 `__prepare__` 命名空间钩子

3.3.3.5. 执行类主体

类主体会以（类似于）`exec(body, globals(), namespace)` 的形式被执行。普通调用与 [exec\(\)](#) 的关键区别在于当类定义发生于函数内部时，词法作用域允许类主体（包括任何方法）引用来自当前和外部作用域的名称。

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

3.3.3.6. 创建类对象

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

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

在 CPython 3.6 及之后的版本中，`__class__` 单元会作为类命名空间中的 `__classcell__` 条目被传给元类。如果存在，它必须被向上传播给 `type.__new__` 调用，以便能正确地初始化该类。如果不这样做，在 Python 3.8 中将引发 [RuntimeError](#)。

当使用默认的元类 [type](#)，或者任何最终会调用 `type.__new__` 的元类时，以下额外的自定义步骤将在创建类对象之后被唤起：

1. `type.__new__` 方法会收集类命名空间中所有定义了 [__set_name__\(\)](#) 方法的属性；
2. 这些 `__set_name__` 方法将附带所定义的类和指定的属性所赋的名称进行调用；
3. 在新类基于方法解析顺序所确定的直接父类上调用 [__init_subclass__\(\)](#) 钩子。

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

当通过 `type.__new__` 创建新类时，作为命名空间形参提供的对象会被拷贝到一个新的有序映射并丢弃原始对象。这个新拷贝包装在一个只读代理中，该代理会成为类对象的 [__dict__](#) 属性。

参见:

[PEP 3135](#) - 新的超类型

描述隐式的 `__class__` 闭包引用

3.3.3.7. 元类的作用

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

3.3.4. 自定义实例及子类检查

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

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

`type.__instancecheck__(self, instance)`

如果 *instance* 应被视为 *class* 的一个（直接或间接）实例则返回真值。如果定义了此方法，则会被调用以实现 `isinstance(instance, class)`。

`type.__subclasscheck__(self, subclass)`

Return true 如果 *subclass* 应被视为 *class* 的一个（直接或间接）子类则返回真值。如果定义了此方法，则会被调用以实现 `issubclass(subclass, class)`。

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

参见:

[PEP 3119](#) - 引入抽象基类

包括通过 [__instancecheck__\(\)](#) 和 [__subclasscheck__\(\)](#) 来定制 [isinstance\(\)](#) 和 [issubclass\(\)](#) 行为的说明，加入此功能的动机是出于向语言添加抽象基类的场景（参见 [abc](#) 模块）。

3.3.5. 模拟泛型类型

当使用 [类型标注](#) 时，使用 Python 的方括号标记来形参化一个 [generic type](#) 往往会很有用处。例如，`list[int]` 这样的标注可以被用来表示一个 [list](#) 中的所有元素均为 [int](#) 类型。

参见:

[PEP 484](#) —— 类型注解

介绍 Python 中用于类型标注的框架

[泛用别名类型](#)

代表形参化泛用类的对象的文档

泛型 (Generic), 用户自定义泛型 和 `typing.Generic`

有关如何实现可在运行时被形参化并能被静态类型检查器所识别的泛用类的文档。

一个类 通常 只有在定义了特殊的类方法 `__class_getitem__()` 时才能被形参化。

`classmethod object.__class_getitem__(cls, key)`

按照 `key` 参数指定的类型返回一个表示泛型类的专门化对象。

当在类上定义时, `__class_getitem__()` 会自动成为类方法。因此, 当它被定义时没有必要使用 `@classmethod` 来装饰。

3.3.5.1. `__class_getitem__` 的目的

`__class_getitem__()` 的目的是允许标准库泛型类的运行时形参化以更方便地对这些类应用 [类型提示](#)。

要实现可以在运行时被形参化并可被静态类型检查所理解的自定义泛型类, 用户应当从已经实现了 `__class_getitem__()` 的标准库类继承, 或是从 `typing.Generic` 继承, 这个类拥有自己的 `__class_getitem__()` 实现。

标准库以外的类上的 `__class_getitem__()` 自定义实现可能无法被第三方类型检查器如 `mypy` 所理解。不建议在任何类上出于类型提示以外的目的使用 `__class_getitem__()`。

3.3.5.2. `__class_getitem__` 与 `__getitem__`

通常, 使用方括号语法 [抽取](#) 一个对象将会调用在该对象的类上定义的 `__getitem__()` 实例方法。

不过, 如果被拟抽取的对象本身是一个类, 则可能会调用 `__class_getitem__()` 类方法。

`__class_getitem__()` 如果被正确地定义, 则应当返回一个 `GenericAlias` 对象。

使用 [表达式](#) `obj[x]` 来呈现, Python 解释器会遵循下面这样的过程来确定应当调用 `__getitem__()` 还是 `__class_getitem__()`:

```
from inspect import isclass

def subscribe(obj, x):
    """返回表达式 'obj[x]' 的结果"""

    class_of_obj = type(obj)

    # 如果 obj 所属的类定义了 __getitem__,
    # 则调用 class_of_obj.__getitem__(obj, x)
    if hasattr(class_of_obj, '__getitem__'):
        return class_of_obj.__getitem__(obj, x)

    # 否则, 如果 obj 是一个类并且定义了 __class_getitem__,
    # 则调用 obj.__class_getitem__(x)
    elif isclass(obj) and hasattr(obj, '__class_getitem__'):
        return obj.__class_getitem__(x)

    # 否则, 引发一个异常
    else:
```

```
raise TypeError(
    f"'{class_of_obj.__name__}' object is not subscriptable"
)
```

在 Python 中，所有的类自身也是其他类的实例。一个类所属的类被称为该类的 [metaclass](#)，并且大多数类都将 [type](#) 类作为它们的元类。[type](#) 没有定义 [__getitem__\(\)](#)，这意味着 `list[int]`，`dict[str, float]` 和 `tuple[str, bytes]` 这样的表达式都将导致 [__class_getitem__\(\)](#) 被调用：

```
>>> # list 以 "type" 类作为其元类，与大多数类一样：
>>> type(list)
<class 'type'>
>>> type(dict) == type(list) == type(tuple) == type(str) == type(bytes)
True
>>> # "list[int]" 将调用 "list.__class_getitem__(int)"
>>> list[int]
list[int]
>>> # list.__class_getitem__ 将返回一个 GenericAlias 对象：
>>> type(list[int])
<class 'types.GenericAlias'>
```

然而，如果一个类属于定义了 [__getitem__\(\)](#) 的自定义元类，则抽取该类可能导致不同的行为。这方面的一个例子可以在 [enum](#) 模块中找到：

```
>>> from enum import Enum
>>> class Menu(Enum):
...     """A breakfast menu"""
...     SPAM = 'spam'
...     BACON = 'bacon'
...
>>> # 枚举类有一个自定义元类：
>>> type(Menu)
<class 'enum.EnumMeta'>
>>> # EnumMeta 定义了 __getitem__，
>>> # 因此 __class_getitem__ 不会被调用，
>>> # 并且结果不是一个 GenericAlias 对象：
>>> Menu['SPAM']
<Menu.SPAM: 'spam'>
>>> type(Menu['SPAM'])
<enum 'Menu'>
```

参见：

[PEP 560](#) - 对 typing 模块和泛型的核心支持

介绍 [__class_getitem__\(\)](#)，并指明 [抽取](#) 在何时会导致 [__class_getitem__\(\)](#) 而不是 [__getitem__\(\)](#) 被调用

3.3.6. 模拟可调用对象

`object.__call__(self[, args...])`

此方法会在实例作为一个函数被“调用”时被调用；如果定义了此方法，则 `x(arg1, arg2, ...)` 大致可以被转写为 `type(x).__call__(x, arg1, ...)`。[object](#) 类本身没有提供此方法。

3.3.7. 模拟容器类型

以下方法可被定义以实现容器对象。这些方法均不由 `object` 类本身提供。容器通常是 [序列](#) (如 [列表](#) 或 [元组](#)) 或 [映射](#) (如 [字典](#))，但也可表示其他容器类型。第一组方法用于模拟序列或映射；区别在于对于序列，允许的键应为满足 $0 \leq k < N$ 的整数 k (其中 N 为序列长度) 或定义项目范围的 [slice](#) 对象。还建议映射提供以下方法：行为类似于 Python 标准 [字典](#) 对象的 `keys()`、`values()`、`items()`、`get()`、`clear()`、`setdefault()`、`pop()`、`popitem()`、`copy()` 和 `update()`。`collections.abc` 模块提供了一个 [MutableMapping abstract base class](#)，用于基于一组基础方法 (`__getitem__()`、`__setitem__()`、`__delitem__()` 和 `keys()`) 帮助创建这些方法。

可变序列应提供以下方法：`append()`、`clear()`、`count()`、`extend()`、`index()`、`insert()`、`pop()`、`remove()` 和 `reverse()`，与 Python 标准 [list](#) 对象类似。最后，序列类型应通过定义以下方法实现加法（表示拼接）和乘法（表示重复）：`__add__()`、`__radd__()`、`__iadd__()`、`__mul__()`、`__rmul__()` 和 `__imul__()`；不应定义其他数值运算符。

建议映射和序列都实现 `__contains__()` 方法以支持高效的 `in` 运算符使用：对于映射，`in` 应搜索映射的键；对于序列，则应搜索值。还建议映射和序列都实现 `__iter__()` 方法以支持对容器的高效迭代：对于映射，`__iter__()` 应迭代对象的键；对于序列，则应迭代值。

`object.__len__(self)`

调用此方法以实现内置函数 `len()`。应该返回对象的长度，以一个 ≥ 0 的整数表示。此外，如果一个对象未定义 `__bool__()` 方法而其 `__len__()` 方法返回值为零则它在布尔运算中将被视为具有假值。

在 CPython 中，要求长度最大只能为 `sys.maxsize`。如果长度大于 `sys.maxsize` 则某些特性 (如 `len()`) 可能会引发 `OverflowError`。要防止真值测试引发 `OverflowError`，对象必须定义 `__bool__()` 方法。

`object.__length_hint__(self)`

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

Added in version 3.4.

备注：切片是通过下述三个专门方法完成的。以下形式的调用

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

会为转写为

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

其他形式以此类推。略去的切片项总是以 `None` 补全。

`object.__getitem__(self, key)`

调用此方法以实现 `self[key]` 的求值。对于 [sequence](#) 类型，接受的键应为整数。作为可选项，它们也可能支持 [slice](#) 对象。对负数索引的支持也是可选项。如果 `key` 的类型不正确，则可能引发 [TypeError](#)。如果 `key` 为序列索引集合范围以外的值（在进行任何负数索引的特殊解读之后），则应当引发 [IndexError](#)。对于 [mapping](#) 类型，如果 `key` 找不到（不在容器中），则应当引发 [KeyError](#)。

备注： [for](#) 循环在有不合法索引时会期待捕获 [IndexError](#) 以便正确地检测到序列的结束。

备注： 当 [抽取](#) 一个 `class` 时，可能会调用特殊类方法 [__class_getitem__\(\)](#) 而不是 [__getitem__\(\)](#)。请参阅 [__class_getitem__](#) 与 [__getitem__](#) 了解详情。

`object.__setitem__(self, key, value)`

调用此方法以实现向 `self[key]` 赋值。注意事项与 [__getitem__\(\)](#) 相同。为对象实现此方法应该仅限于需要映射允许基于键修改值或添加键，或是序列允许元素被替换时。不正确的 `key` 值所引发的异常应与 [__getitem__\(\)](#) 方法的情况相同。

`object.__delitem__(self, key)`

调用此方法以实现 `self[key]` 的删除。注意事项与 [__getitem__\(\)](#) 相同。为对象实现此方法应该仅限于需要映射允许移除键，或是序列允许移除元素时。不正确的 `key` 值所引发的异常应与 [__getitem__\(\)](#) 方法的情况相同。

`object.__missing__(self, key)`

此方法由 [dict.__getitem__\(\)](#) 在找不到字典中的键时调用以实现 `dict` 子类的 `self[key]`。

`object.__iter__(self)`

此方法会在需要为一个容器创建 [iterator](#) 时被调用。此方法应当返回一个新的迭代器对象，它可以对容器中的所有对象执行迭代。对于映射，它应当对窗口中的键执行迭代。

`object.__reversed__(self)`

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

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

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

`object.__contains__(self, item)`

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

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

3.3.8. 模拟数字类型

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

```
object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__matmul__(self, other)
object.__truediv__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__divmod__(self, other)
object.__pow__(self, other[, modulo])
object.__lshift__(self, other)
object.__rshift__(self, other)
object.__and__(self, other)
object.__xor__(self, other)
object.__or__(self, other)
```

调用这些方法来实现双目算术运算 (+, -, *, @, /, //, %, [divmod\(\)](#), [pow\(\)](#), **, <<, >>, &, ^, |)。例如，求表达式 `x + y` 的值，其中 `x` 是具有 `__add__()` 方法的类的一个实例，则会调用 `type(x).__add__(x, y)`。[divmod\(\)](#) 方法应该等价于使用 [floordiv__\(\)](#) 和 [mod__\(\)](#)；它不应该被关联到 [truediv__\(\)](#)。注意，如果要支持内置 [pow\(\)](#) 函数的三个参数版本，[pow__\(\)](#) 应该定义为接受可选的第三个参数。

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

```
object.__radd__(self, other)
object.__rsub__(self, other)
object.__rmul__(self, other)
object.__rmatmul__(self, other)
object.__rtruediv__(self, other)
object.__rfloordiv__(self, other)
object.__rmod__(self, other)
object.__rdivmod__(self, other)
object.__rpow__(self, other[, modulo])
object.__rlshift__(self, other)
object.__rrshift__(self, other)
object.__rand__(self, other)
object.__rxor__(self, other)
object.__ror__(self, other)
```

调用这些方法来实现具有反射（交换）操作数的双目算术运算 (+, -, *, @, /, //, %, [divmod\(\)](#), [pow\(\)](#), **, <<, >>, &, ^, |)。只有当操作数的类型不同、左操作数不支持相应的操作 [3]，或者右操作数的类派生自左操作数的类时，才调用这些函数。[4] 例如，求表达式 `x - y` 的值，其中 `y` 是具有 `__rsub__()` 方法的类的一个实例，则当 `type(x).__sub__(x, y)` 返回 [NotImplemented](#) 或“`type(y)`”是“`type(x)`”的子类时调用 `type(y).__rsub__(y, x)`。[5]

请注意应当定义 `__rpow__()` 以在需要支持内置 `pow()` 函数的三参数版本时接受可选的第三个参数。

在 3.14 版本发生变更: 三参数的 `pow()` 现在会在必要时尝试调用 `__rpow__()`。在之前版本中它只会在二参数的 `pow()` 和二元幂运算符中被调用。

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

```
object.__iadd__(self, other)
object.__isub__(self, other)
object.__imul__(self, other)
object.__imatmul__(self, other)
object.__itruediv__(self, other)
object.__ifloordiv__(self, other)
object.__imod__(self, other)
object.__ipow__(self, other[, modulo])
object.__ilshift__(self, other)
object.__irshift__(self, other)
object.__iand__(self, other)
object.__ixor__(self, other)
object.__ior__(self, other)
```

调用这些方法来实现增强算术赋值 (`+=`, `-=`, `*=`, `@=`, `/=`, `//=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=`, `|=`)。

这些方法应当尝试原地执行操作 (对 `self` 进行修改) 并返回结果 (结果可以为 `self` 但这并非必须)。如果某个方法未被定义, 或者如果该方法返回 `NotImplemented`, 则相应的增强赋值将回退到普通方法。举例来说, 如果 `x` 是一个具有 `__iadd__()` 方法的类的实例, 则 `x += y` 就等价于 `x = x.__iadd__(y)`。如果 `__iadd__()` 不存在, 或者如果 `x.__iadd__(y)` 返回 `NotImplemented`, 则将使用 `x.__add__(y)` 和 `y.__radd__(x)`, 如同对 `x + y` 求值一样。在某些情况下, 增强赋值可能导致未预期的错误 (参见 [为什么 a tuple\[i\] += \['item'\] 会引发异常?](#)), 但此行为实际上是数据模型的一部分。

```
object.__neg__(self)
object.__pos__(self)
object.__abs__(self)
object.__invert__(self)
```

调用此方法以实现一元算术运算 (`-`, `+`, `abs()` 和 `~`)。

```
object.__complex__(self)
object.__int__(self)
object.__float__(self)
```

调用这些方法以实现内置函数 `complex()`, `int()` 和 `float()`。应当返回一个相应类型的值。

```
object.__index__(self)
```

调用此方法以实现 `operator.index()` 以及 Python 需要无损地将数字对象转换为整数对象的场合 (例如切片或是内置的 `bin()`, `hex()` 和 `oct()` 函数)。存在此方法表明数字对象属于整数类型。必须返回一个整数。

如果未定义 `__int__()`, `__float__()` 和 `__complex__()` 则相应的内置函数 `int()`, `float()` 和 `complex()` 将回退为 `__index__()`。

```
object.__round__(self[, ndigits])
object.__trunc__(self)
object.__floor__(self)
object.__ceil__(self)
```

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

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

3.3.9. with 语句上下文管理器

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

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

有关上下文管理器的更多信息，请参阅 [上下文管理器类型](#)。`object` 类本身不提供上下文管理器方法。

```
object.__enter__(self)
```

进入与此对象相关的运行时上下文。`with` 语句将会绑定这个方法的返回值到 `as` 子句中指定的目标，如果有的话。

```
object.__exit__(self, exc_type, exc_value, traceback)
```

退出关联到此对象的运行时上下文。各个参数描述了导致上下文退出的异常。如果上下文是无异常地退出的，三个参数都将为 `None`。

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

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

参见:

[PEP 343](#) - "with" 语句

Python [with](#) 语句的规范描述、背景和示例。

3.3.10. 定制类模式匹配中的位置参数

当在模式中使用类名称时，默认不允许模式中出现位置参数，例如在 `MyClass` 没有特别支持的情况下 `case MyClass(x, y)` 通常是无效的。要能使用这样的模式，类必须定义一个 `__match_args__` 属性。

object.__match_args__

该类变量可以被赋值为一个字符串元组。当该类被用于带位置参数的类模式时，每个位置参数都将被转换为关键字参数，并使用 `__match_args__` 中的对应值作为关键字。缺失此属性就等价于将其设为 `()`。

举例来说，如果 `MyClass.__match_args__` 为 `("left", "center", "right")` 则意味着 `case MyClass(x, y)` 就等价于 `case MyClass(left=x, center=y)`。请注意模式中参数的数量必须小于等于 `__match_args__` 中元素的数量；如果前者大于后者，则尝试模式匹配时将引发 [TypeError](#)。

Added in version 3.10.

参见:

[PEP 634](#) - 结构化模式匹配

有关 Python `match` 语句的规范说明。

3.3.11. 模拟缓冲区类型

[缓冲区协议](#) 为 Python 对象提供了一种向低层级内存数组暴露高效访问的方式。该协议是通过内置类型如 [bytes](#) 和 [memoryview](#) 实现的，还可能由第三方库定义额外的缓冲区类型。

虽然缓冲区类型通常都是用 C 实现的，但用 Python 来实现该协议也是可能的。

object.__buffer__(self, flags)

当从 `self` 请求一个缓冲区时将被调用（例如，从 [memoryview](#) 构造器）。`flags` 参数是代表所请求缓冲区的类别的整数，例如这会影响返回的缓冲区是只读还是可写。[inspect.BufferFlags](#) 提供了解读旗标的便利方式。此方法必须返回一个 [memoryview](#) 对象。

object.__release_buffer__(self, buffer)

当一个缓冲区不再需要时将被调用。`buffer` 参数是在此之前由 [__buffer__\(\)](#) 返回的 [memoryview](#) 对象。此方法必须释放任何关联到该缓冲区的资源。此方法应当返回 `None`。不需要执行任何清理的缓冲区对象不要求实现此方法。

Added in version 3.12.

参见:

[PEP 688](#) - 使缓冲区协议在 Python 中可访问

引入 Python `__buffer__` 和 `__release_buffer__` 方法。

[collections.abc.Buffer](#)

缓冲区类型的 ABC。

3.3.12. 标注

函数、类和模块可能包含 [注解](#)，这是一种将信息（通常是 [类型注解](#)）与符号关联起来的方法。

object.__annotations__

此属性包含对象的注解。它是 [惰性求值](#)，因此访问该属性可能会执行任意代码并引发异常。如果求值成功，则将属性设置为从变量名到注解的字典映射。

在 3.14 版本发生变更：标注现在将被惰性求值。

`object.__annotate__(format)`

一个 [annotate function](#)。将返回一个将属性/形参名称映射到其标注值的新字典对象。

接受一个格式形参，该参数指定应以何种格式提供注解值。它必须是 [annotationlib.Format](#) 枚举的成员，或者是一个值对应于枚举成员的整数。

如果注解函数不支持请求的格式，它必须引发 [NotImplementedError](#)。注解函数必须始终支持 [VALUE](#) 格式；当以这种格式调用时，它们不能引发 [NotImplementedError\(\)](#)。

当以 [VALUE](#) 格式调用时，注解函数可能引发 [NameError](#)；当调用请求任何其他格式时，它不得引发 [NameError](#)。

如果一个对象没有任何注解，`__annotate__` 最好设置为 `None`（不能删除），而不是设置为返回空字典的函数。

Added in version 3.14.

参见:

[PEP 649](#) --- 使用描述器进行延迟标注求值

引入标注的惰性求值以及 `__annotate__` 函数。

3.3.13. 特殊方法查找

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

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

此行为背后的原理在于包括类型对象在内的所有对象都会实现的几个特殊方法如 [__hash__\(\)](#) 和 [__repr__\(\)](#)。如果这些方法的隐式查找使用了传统的查找过程，则当它们在针对类型对象自身被唤起时将会失败：

```
>>> 1.__hash__() == hash(1)
True
>>> int.__hash__() == hash(int)
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
TypeError: descriptor '__hash__' of 'int' object needs an argument
```

以这种方式不正确地尝试唤起一个类的未绑定方法有时被称为‘元类混淆’，可以通过在查找特殊方法时绕过实例的方式来避免：

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

除了出于正确性考虑而会绕过任何实例属性，隐式特殊方法查找通常还会绕过 `__getattr__()` 方法，甚至包括对象的元类：

```
>>> class Meta(type):
...     def __getattr__(*args):
...         print("Metaclass getattr invoked")
...         return type.__getattr__(*args)
...
>>> class C(object, metaclass=Meta):
...     def __len__(self):
...         return 10
...     def __getattr__(*args):
...         print("Class getattr invoked")
...         return object.__getattr__(*args)
...
>>> c = C()
>>> c.__len__()                                # 通过实例的显式查找
Class getattr invoked
10
>>> type(c).__len__(c)                         # 通过类型的显式查找
Metaclass getattr invoked
10
>>> len(c)                                    # 隐式查找
10
```

以这种方式绕过 `__getattr__()` 机制为解释器内部的速度优化提供了显著的空间，其代价则是牺牲了一些处理特殊方法时的灵活性（特殊方法 *must* 必须设置在类对象自身上以便始终一致地由解释器唤起）。

3.4. 协程

3.4.1. 可等待对象

`awaitable` 对象主要实现了 `__await__()` 方法。从 `async def` 函数返回的 [协程对象](#) 即为可等待对象。

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

```
object.__await__(self)
```

必须返回一个 [iterator](#)。应当被用来实现 [awaitable](#) 对象。例如，[asyncio.Future](#) 实现了此方法以与 [await](#) 表达式兼容。[object](#) 类本身不是可等待对象因而不提供此方法。

备注： 本语言不会对 `__await__` 所返回的迭代器产生的对象的类型或值施加任何限制，因为这是负责管理 [awaitable](#) 对象的异步执行框架的具体实现 (如 [asyncio](#)) 专属特性。

Added in version 3.5.

参见： [PEP 492](#) 了解有关可等待对象的详细信息。

3.4.2. 协程对象

[协程对象](#) 属于 [awaitable](#) 对象。协程的执行可以通过调用 `__await__()` 并迭代其结果来控制。当协程结束执行并返回时，迭代器会引发 [StopIteration](#)，而该异常的 `value` 属性将存放返回值。如果协程引发了异常，它会被迭代器传播出去。协程不应当直接引发未被处理的 [StopIteration](#) 异常。

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

在 3.5.2 版本发生变更： 等待一个协程超过一次将引发 [RuntimeError](#)。

`coroutine.send(value)`

开始或恢复协程的执行。如果 `value` 为 `None`，这将等价于前往 `__await__()` 所返回的迭代器的下一项。如果 `value` 不为 `None`，此方法将委托给导致协挂起的迭代器的 `send()` 方法。其结果 (返回值, [StopIteration](#) 或是其他异常) 将与上述对 `__await__()` 返回值进行迭代的结果相同。

`coroutine.throw(value)`

`coroutine.throw(type[, value[, traceback]])`

在协程内引发指定的异常。此方法将委托给导致该协程挂起的迭代器的 `throw()` 方法，如果存在此方法的话。否则，该异常将在挂起点被引发。其结果 (返回值, [StopIteration](#) 或是其他异常) 将与上述对 `__await__()` 返回值进行迭代的结果相同。如果该异常未在协程内被捕获，则将回传给调用方。

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。

`coroutine.close()`

此方法会使得协程清理自身并退出。如果协程被挂起，此方法会先委托给导致协程挂起的迭代器的 `close()` 方法，如果存在该方法。然后它会在挂起点引发 [GeneratorExit](#)，使得协程立即清理自身。最后，协程会被标记为已结束执行，即使它根本未被启动。

当协程对象将要被销毁时，会使用以上处理过程来自动关闭。

3.4.3. 异步迭代器

异步迭代器可以在其 `__anext__` 方法中调用异步代码。

异步迭代器可在 [async for](#) 语句中使用。

[object](#) 类本身不提供这些方法。

`object.__aiter__(self)`

必须返回一个 异步迭代器 对象。

`object.__anext__(self)`

必须返回一个 可等待对象 输出迭代器的下一结果值。当迭代结束时应该引发 [StopAsyncIteration](#) 错误。

异步可迭代对象的一个示例:

```
class Reader:
    async def readline(self):
        ...

    def __aiter__(self):
        return self

    async def __anext__(self):
        val = await self.readline()
        if val == b'':
            raise StopAsyncIteration
        return val
```

Added in version 3.5.

在 3.7 版本发生变更: 在 Python 3.7 之前, [__aiter__\(\)](#) 可以返回一个 可等待对象 并将被解析为 异步迭代器。

从 Python 3.7 开始, [__aiter__\(\)](#) 必须返回一个异步迭代器对象。返回任何其他对象都将导致 [TypeError](#) 错误。

3.4.4. 异步上下文管理器

异步上下文管理器是 上下文管理器 的一种, 它能够在其 `__aenter__` 和 `__aexit__` 方法中暂停执行。

异步上下文管理器可在 [async with](#) 语句中使用。

[object](#) 类本身不提供这些方法。

`object.__aenter__(self)`

在语义上类似于 [__enter__\(\)](#), 仅有的区别在于它必须返回一个 可等待对象。

`object.__aexit__(self, exc_type, exc_value, traceback)`

在语义上类似于 [__exit__\(\)](#), 仅有的区别在于它必须返回一个 可等待对象。

异步上下文管理器类的一个示例:

```
class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')
```

Added in version 3.5.

备注

- [1] 在某些情况下 *有可能* 基于可控的条件改变一个对象的类型。但这通常不是个好主意，因为如果处理不当会导致一些非常怪异的行为。
- [2] `__hash__()`, `__iter__()`, `__reversed__()`, `__contains__()`, `__class_getitem__()` 和 `__fspath__()` 方法对此有特殊处理。其他方法仍然会引发 `TypeError`，但可能会依赖 `None` 是不可调用对象的行为来做到这一点。
- [3] 这里的“不支持”是指该类无此方法，或方法返回 `NotImplemented`。如果你想强制回退到右操作数的反射方法，请不要设置方法为 `None` — 那会造成显式地 *阻塞* 此种回退的相反效果。
- [4] 对于相同类型的操作数，如果非反射方法 (如 `__add__()`) 执行失败则相应的操作将被视为不受支持，这就是反射方法不会被调用的原因。
- [5] 如果右操作数的类型为左操作数的类型的子类，则具有优先地位的反射方法将允许子类重写其祖先类的操作。