数据模型

ZhangXu

2019 年 1 月 28 日

1 对象,值和类型

*Objects*是python数据的抽象概念。所有程序数据均由对象或对象间关系表示。

每个对象都有一个*identity*标志,一个*type*类型,一个*value*值。一个对象的标志在创建后永不改变；可以认为其是**对象在内存中的地址**。'is'操作符用于比较两个对象的id；id()函数返回标志的整数值。

CPython implementation detail:id(x)是x的内存地址。

对象的**类型**决定了对象拥有的**操作**且还定义了该类型对象的可能值。type()函数返回对象的类型(它本身就是一个对象)。和id一样，对象的类型是不变的。

而某些对象的值可以改变分**可变类型和不可变类型**。不可变容器中包含可变对象，后者值可以更改，但仍视其为不可变的，因容器包含的对象集合无法更改。(不变性和不可改变值不完全相同)。类型决定对象的可变性否；例如不可变类型：实例，数字，字符串和元祖。可变类型：字典，列表。

对象从不显式销毁；当对象不可访问时它们被垃圾回收。允许实现推迟垃圾收集或完全忽略它——其是垃圾收集的实现质量问题，只要没有收集到仍然可以访问对象。

CPython implementation detail:CPython使用带有(可选的)延迟检测(cyclically linked garbage)循环链接垃圾的引用计数机制，它会在大多数对象无法访问时立即收集，但无法保证收集包含循环引用的的垃圾。(有关控制循环垃圾收集的信息，参阅gc模块文档)。当对象无法访问时，不要依赖于对象的立即终结(因此应该始终显式的关闭文件)。

注意：使用实现的追踪或调试工具可使对象保持活动状态，这些对象通常是可收集的。另注意'try...except'捕获异常也可能使对象保持活动状态。

一些对象可能包含对外界资源的引用，例如打开文件或窗口。当对象的被垃圾回收时，这些资源被释放，但由于**垃圾回收不保证发生**，这些对象也提供

了一种释放外部资源的明确方法，通常是close()方法。代码强烈建议显式关闭对象。'try...finally'语句和'with'语句提供了方便的方法。

一些对象包含对其他对象的引用；称之为containers容器。如元祖，列表和字典。引用仅是容器值的一部分。当我们讨论一个容器的值时，我们暗指其数值，而不是容器对象的标志；而当我们讨论容器的可变性时，暗指直接包含的对象的标志。所以，当一个不可变容器包含一个可变对象的引用时，当可变对象改变，其值改变。

类型几乎影响了对象所有方面的行为。在某种意义上，甚至对象的标志的重要性也会受到影响：比如对于不可变类型，计算新值的操作实际上可能返回对具有相同类型和值的任何**现有对象**的引用，而对于可变对象，其不允许。对于可变类型，保证指向不同的，独一无二的新创建的空对象。

2 标准类型层次结构

一些列出Python内建类型。扩展模块(用C,java,或其他语言编写，具体取决于实现)可以定义其他类型。未来版本会向类型层次结构添加类型。(例如有理数，有效存储的整数数组等)，这些附加类型通常会通过标准库提供。

下面一些类型描述包含一个列出"特殊属性"的段落。其属性提供对现实方法的访问，不适用于一般用途。他们的定义可能在未来发生变化。

2.1 None

此类型具有单个值。有一个具有此值的对象。通过内置名称None访问该对象。用于表示许多情况下值得缺少。其真值为假。

2.2 NotImplemented

此类型具有单个值。有一个具有此值的对象。通过内置名称NotImplemented访问此对象。如果数值方法和丰富的比较方法为实现所提供操作数的操作，则返回此值。其值为真。

2.3 Ellipsis

此类型具有单个值。有一个具有此值的对象。通过内置名称...或内建名称Ellipsis访问此对象。真值为真。

2.4 numbers.Number

由数字字面值创建，并由算术运算符和算术内置函数作为结果返回。数值对象是不可变的。
Python区分整数，浮点数和复数：

2.4.1 numbers.Integral

代表来自整数数学集合的元素(正和负)
两种整数类型:Integers(int) Booleans(bool)

numbers.Real(float) 代表机器级双精度浮点数。Python不支持单精度浮点数；

numbers.Complex(complex) 将复数表示为一对机器级双精度浮点数。z.real和z.imag分别访问复数z的实部和虚部。

2.5 序列Sequences

由非负数索引的有限有序集表示序列。
序列支持切片：$a[i:j]$选择$i \leq k < j$。当作为表达式，切片是相同类型的序列。

一些序列支持第三个'step'参数的"扩展":a[i:j:k]选择$x = i + n * k$的元素,$i \leq k < j$。

2.5.1 不可变序列

Strings 字符串是表示Unicode编号的值的序列。范围从U+0000 - U+10FFFF可在字符串中表示。Python没有char类型；相反的，字符串中每个Unicode编号代表长度为1。内建函数ord()将Unicode编号转化为整数，范围是0 - 10FFFF。chr()函数将一个整数值在0 - 10FFFF范围转化为长度为1的字符串对象。str.encode()函数转化一个str为一个bytes。bytes.decode()反过来。

Tuples 元组的项是任意的python对象。由逗号分隔。当形成一个项目的元组，可通过加逗号附加到表达式(表达式本身不会创建元组，因为括号必须可用于表达式的分组)。而空元组由一对空括号组成。

Bytes bytes对象是一个不可变的数组。由0 - 255组成。Bytes字面值(b'abc')或内建函数bytes()构造器可以创造bytes对象。bytes通过decode()为字符串。

2.5.2 可变序列

The subscription and slicing notations can be used as the target of assignment and del (delete) statements.
有两种内在的可变序列类型：

Lists 列表项可以是python任意对象。由方括号内的逗号分隔。长度为0或1的列表不需要特殊情况。

Byte Arrays **bytearray对象是一个可变数组**。由内建构造器bytearray()构造。除了可变(因此不可哈希)之外，字节数组提供于不可变字节对象bytes相同的接口和功能。
扩展模块array提供了可变序列类型的另一个示例，collections模块也是。

2.6 Set types

代表无序，有限，唯一的不可变对象集。它们不能被任何下标索引。但是，它们可以迭代，len()返回集合中项目数。常见用途是快速成员资格测试，从序列中删除，以及计算数学运算，例如交集，并集，差异和对称差异。
对于set元素，相同的不变性规则适用于字典关键字。**注意：相同数字值之后能包含一个。**
有两种固有的集合类型：

2.6.1 Sets

代表可变集合。由内建函数set()构造并且可由一些函数修改，例如add().

2.6.2 Frozen sets

代表不可见类型。内建函数frozenset()构造。其是不可变且可哈希的，可以用作另一个集合的元素，或者作为字典关键字。

2.7 Mappings

表示由任意索引集索引的有限对象集。可以在表达式中使用，也可以作为赋值或del语句的目标。len()返回映射中的项数。有一种内在映射类型。

2.7.1 Dictionaries字典

唯一不能作为关键字的值是包含列表或字典或其他可变类型的值，这些类型按值而不是按对象的标志进行比较，原因是字典的有效实现需要关键字的哈希值保持不变。如果两数字如1和1.0相等，那么他们可以互换使用以索引相同的字典条目。(遵循数字之间比较的原则)
字典是可变的；可由...创建。
扩展库**dbm.ndbm和dbm.gnu**提供了额外字典类型的例子，collenctions模块同样如此。

2.8 Callable types

这些是应用函数操作(参阅Calls部分)的类型。

2.8.1 User-defined functions

用户定义函数对象是由函数定义创建。应该使用包含与函数形式参数列相同数量的项的参数列表来调用它。特殊属性如下：

Attribute	Meaning	
__doc__	函数的文档字符串，或None；不可被子类继承	可写
__name__	函数名	可写
__qualname__	函数的限定名	可写
__module__	函数定义的模块名，或None	可写
__defaults__	包含具有默认值的参数的默认参数值的元组，或None	可写
__code__	代码对象，表示已经编译好的函数体	可写
__globals__	对包含函数全局变量的字典的引用，即定义函数模块的全局命名空间	只读
__dict__	支持任意函数属性的命名空间	可写
__closure__	无或包含函数自由变量绑定的单元格元组	只读
__annotations__	包含注释的字典。dict的关键字是参数名称，返回注释的return（如果提供)	可写
__kwdefaults__	包含仅关键字参数的默认值的字典dict	可写

标记"可写"的大多数属性都会检查指定值的类型。

函数对象还支持获取和设置任意属性，例如，可以使用这些属性将元数据附加到函数。(ZX:?)常规使用属性点表示法用于获取或设置属性。注意：当前实现仅支持用户自定义函数的函数属性。将来可能会支持内置函数的函数属性。
单元对象(ZX:?)具有属性cell_contents。可以用于获取单元格的属性，以及设置值。
可以从代码对象中检索有关函数定义的其他信息；请参阅下面的内部类型说明。

2.8.2 实例方法(Instance methods)

实例方法对象包含了类，类实例和任何可调用的对象(通常是用户定义的函数)

特殊只读属性 __self__是类实例对象，__func__是函数对象；__doc__是方法的文档(同__func__.__doc__)；__name__是方法名称(同__func__.__name__)；
__module__是定义方法的模块的名称，不可用则是None。
方法还支持访问（但不可设置）底层函数对象上的任意函数属性。
当获取类的属性(可能通过该类的实例)时，如果该属性是用户自定义的函数对象或类方法对象，则可以创建用户定义的方法对象。
如果从类或实例检索另一个方法对象来创建用户定义的方法对象时，行为与函数对象的行为相同，除了新实例的__func__属性不是原始方法对象，而是其__func__属性。(ZX:不是太理解)
当通过从类或实例检索类方法对象来创建实例方法对象时，其__self__属性是类本身，其__func__属性是类方法的底层函数对象。
调用实例方法对象时，将调用底层函数__func__，将类实例__self__插入参数列表前面。例如：C的包含函数f()定义的类，而x是C的一个实例，调用x.f(1)等于调用C.f(x, 1)。
当一个实例方法对象是从类方法对象派生时，__self__中存储的"类实例"实际上就是类本身，因此调用x.f(1）或C.f(1)等同于调用f(C, 1)f是底层函数。
注意，每次从实例检索属性时，都会发生从函数对象到实例方法对象的转换。在某些情况下，卓有成效的优化是将属性分配给局部变量并调用该局部变量。
注意：此转换仅适用于用户定义的函数；在不可进行转换的情况下检索其他可调用对象(以及所有不可调用对象)。同样注意的是，作为类实例属性的用户定义函数不会转换为绑定方法；这只有在函数是类的属性时才会发生。

2.8.3 生成器函数

使用yield语句的函数或方法，称为generator function生成器函数。此类函数在被调用时总是返回一个迭代器对象，他可以用来执行函数体：调用迭代器的iterator.__next__()方法将使函数执行，直到它使用yield语句提供一个值。当函数执行return语句或结束时，会发生StopIteration异常，且迭代器将到达要返回的值集的末尾。

2.8.4 协程函数(Coroutine functions)

使用async def(异步def)定义的方法或函数成为协程函数。此类函数调用时返回一个协程对象。它可能包含await表达式，以及async with和async for语句。另参见Coroutine Objects部分。

2.8.5 异步生成函数

使用aync def定义并使用yield语句的函数或方法称为$asynchronous generator function$。此类函数在被调用时，返回一个异步迭代器对象，该对象可在async for语句中用于执行函数体。
调用异步迭代器的aiterator.__anext__()方法将返回一个等待的，等待执行，直到它使用yield表达式提供一个值。当函数执行空的return语句或从结尾处失败时，会引发StopAsyncIteration异常，异步迭代器将到达生成值集的末尾。

2.8.6 内置函数

内置函数对象是C函数的包装器。内置函数示例：len() math.sin()(math是标准内置模块)。参数数量和类型由C函数确定。特殊的只读属性：__ doc__是函数的文档字符串，如果不可用，则为None; __name__是函数的名称; __self__设置为None（但请参阅下一项）; __module__是定义函数的模块的名称，如果不可用则是None

2.8.7 内置方法

实际上是内置函数的不同伪装，这次包含了作为隐式额外参数传递给C函数的对象。内建方法的例子：alist.append()，假设alist是一个列表对象。在这种情况下，特殊的只读属性__self__被设置为alist表示的一个对象。

2.8.8 类

类是可调用的。这些对象通常作为新实例的工厂，但是对于覆盖__new__()的类的类型可能存在变化。调用的参数传递给__new__()，在典型情况下，传递给__init__()以初始化新实。

2.8.9 类实例

任意类的实例均通过调用__call__()方法可使其可调用。

2.9 模块

模块是python代码的基本组织单元，由导入系统创建，import语句调用或通过调用importlib.import_module()和内置函数__import__()。
模块对象具有由字典对象实现的**命名空间**（这是由模块中定义的函数的__globals__属性引用的字典）。属性引用被转换为该字典中的查找，例如m.x相当于m.__dict__["x"]。模块对象不包含用于初始化的代码对象（因为初始化完成后不再需要它）
属性赋值会更新模块命名空间的字典。如m.x = 1等价于m.__dict__["x"] = 1。

预定义(可写)属性： __name__是模块的名称; __doc__是模块的文档字符串，如果不可用，则为None;__annotations__（可选）是包含在模块体执行期间收集的变量注释的字典;__file__是从中加载模块的文件的路径名（如果是从文件加载的）。某些类型的模块可能缺少__file__属性，例如静态链接到解释器的C模块。对于从共享库动态加载的扩展模块，它是共享库文件的路径名。

特殊只读属性： __dict__是作为字典对象的模块的命名空间。

CPython implementation detail: 由于CPython清楚模块字典的方式，当模块超出范围时，该模块字典将被清除，即使其仍有实时引用。避免这种情况，请复制字典或再直接使用字典时保留模块。

2.10 自定义类

通常由类定义创建。类具有字典对象实现的命名空间。类属性引用被转换为字典中查找。例如：C.x被转化为C.__dict__["x"](尽管有许多钩子允许其他方法来定位属性)。当此类的字典中找不到属性名时，在其基类中继续搜索。**对于基类**

的搜索使用C3方法解析顺序，即使在存在"菱形"继承结构的情况下也能正常允许，其中有多个继承路径返回到共同的祖先。有关C3 MRO请参阅
当一个类属性引用(如类c)会产生一个类方法对象时，他会被转换成一个实例方法对象，其__self__属性为c。当它产生一个静态方法对象时，它会被转换为由静态方法对象包装的对象。请参阅Implementing Descriptors了解从类中检索的属性可能于其__dict__中实际包含的属性不同的另一种方式。
类属性赋值更新类的字典，而不是基类的字典。
通过调用一个类对象来产生一个类实例

特殊属性： __name__是类名; __module__是定义类的模块名称;__dict__是包含类的命名空间的字典;__bases__是一个包含基类的元组，按它们在基类列表中出现的顺序排列;__doc__是类的文档字符串，如果未定义，则为None; __annotations__（可选）是包含在类主体执行期间收集的变量注释的字典。

2.11 类实例

类实例具有作为字典实现的名称空间，该字典时搜索属性引用的第一位置。当在那找不到属性，且实例的类具有该名称的属性时，搜索继续使用类属性。如果找到的类属性是用户定义的函数对象，则将其转换为实例方法对象，
其__self__属性为实例。静态方法和类方法对象也被转换；如果没有找到类属性，并且对象的类具有__getattr__()方法，则调用该方法以满足查找。
属性分配和删除会更新实例的字典，而不是类的字典。如果类具有__setattr__()或__delattr__()方法，则调用此方法而不是直接更新字典。
如过类实例具有某些特殊名称的方法，则它们可以假装为数字，序列或映射参加Seecial method names

特殊属性： __dict__是任意的字典；__class__是实例的类。

2.12 I/O对象(也叫做文件对象)

文件对象代表打开的文件。创建文件对象的不同方法:内建函数open()，还有os.open()方法和套接字对象的makefile()方法（也许其他扩展模块提供的函数或方法）。
sys.stdin, sys.stdout, sys.stderr对象被初始化为与解释器的标准输入，输出和错误流相对于的文件对象；它们都以文本模式打开，因此遵循io.TextIOBase抽象类定义的接口。

2.13 内部类型(Internal types)

解释权内部使用的一些类型向用户公开。它们的定义可能会随着解释器版本改变，但为了完整起见，这里提到它们。

2.13.1 代码对象Code objects

代码对象表示*byte – compiled*字节编译的可执行Python代码或字节码。代码对象和函数对象之间的区别在于函数对象博阿寒对函数的全局变量(定义在它的模块)的显示引用，而代码对象不包含上下文；默认参数值也存储在函数对象中，而不是存储在代码对象中（因为它们表示在运行时计算）。与函数对象不同，代码对象是不可变的且不包含(直接或间接)可变对象的引用。

特殊只读属性： co_name给出函数名称; co_argcount是位置参数的数量（包括具有默认值的参数）; co_nlocals是函数使用的局部变量数（包括参数）; co_varnames是一个元组，包含局部变量的名称（以参数名称开头）; co_cellvars是一个元组，包含嵌套函数引用的局部变量的名称; co_freevars是一个包含自由变量名称的元组; co_code是表示字节码指令序列的字符串; co_consts是一个元组，包含字节码使用的文字; co_names是一个元组，包含字节码使用的名称; co_filename是编译代码的文件名; co_firstlineno是函数的第一个行号; co_lnotab是一个字符串，用于编码从字节码偏移到行号的映射（有关详细信息，请参阅解释器的源代码）; co_stacksize是必需的堆栈大小（包括局部变量）; co_flags是一个整数，用于编码解释器的许多标志。

co_flags定义以下标志位： 如果函数使用* arguments语法接受任意数量的位置参数，则设置位0x04;如果函数使用**关键字语法接受任意关键字参数，则设置位0x08;如果函数是生成器，则设置位0x20。
未来的特性声明（来自__future__ import division）也使用co_flags中的位来指示代码对象是否在启用特定功能的情况下编译：如果函数是在未来分区启用的情况下编译的，则设置位0x2000;在早期版本的Python中使用了位0x10和0x1000。
co_flags中其他位保留做内部使用。
如果代码对象表示函数，则co_consts中第一项是函数的文档字符串，若未定义，即None。

2.13.2 帧对象Frame objects

帧对象表示执行帧。他们可能出现在回溯对象中，也会传递给已注册的跟踪对象。

特殊只读属性： f_back是前一个堆栈帧（朝向调用者），如果这是底部堆栈帧，则为None; f_code是在此帧中执行的代码对象; f_locals是用于查找局部变量的字典; f_globals用于全局变量; f_builtins用于内置（内在）名称; f_lasti给出了精确的指令（这是代码对象的字节码字符串的索引）。

特殊可写属性： f_trace，如果不是None，是在代码执行期间调用各种事件的函数（由调试器使用）。通常会为每个新源代码行触发一个事件 - 可以通过将f_trace_lines设置为False来禁用此事件。
通过将f_trace_opcodes设置为True，实现可以允许请求每操作码事件。请注意，如果跟踪函数引发的异常转义为正在跟踪的函数，则可能会导致未定义的解释器行为。
f_lineno是帧的当前行号 - 从跟踪函数内写入此跳转到给定行（仅适用于最底部的帧）。调试器可以通过写入f_lineno来实现Jump命令（也称为Set Next Statement）。
帧对象提供一个函数:

frame.clear() 此方法清除对帧所持有的局部变量的所有引用。此外，如果帧属于发生器，则生成器finallized。这有助于打破涉及帧对象的引用循环（例如，捕获异常并存储其回溯以供以后使用）。

2.13.3 回溯对象Traceback objects

回溯对象表示异常的堆栈跟踪。发生异常时会隐式创建回溯对象，也可通过types.TracebaksType显式创建对象。
对于隐式创建的回溯，当搜索异常处理程序展开执行堆栈时，在每个展开的级别上，在当前回溯之前插入回溯对象。输入异常处理程序时，堆栈跟踪可供程序使用。(参加try语句一节)。它可以作为sys.exc_info()返回的元组的第三项访问，也可以作为捕获的异常的__traceback__属性访问。
当程序不包含合适的处理程序时，堆栈跟踪被写入（格式良好）到标准错误流;如果解释器是交互式的，那么它也可以作为sys.last_traceback提供给用户。
对于显式创建的回溯，由跟踪的创建者决定如何链接tb_next属性以形成完整的堆栈跟踪。

特殊只读属性： tb_frame指向当前级别的执行帧; tb_lineno给出发生异常的行号; tb_lasti表示准确的指令。如果在没有匹配的except子句或finally子句

的try语句中发生异常，则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

特殊可写属性： tb_next是堆栈跟踪中的下一个级别（朝向发生异常的帧），如果没有下一级别，则为None。
(现在可以从Python代码显式实例化回溯对象，并且可以更新现有实例的tb_next属性。)

2.13.4 切片对象Slice objects

切片对象用于表示__getitem__()方法的切片。它们也是由内置slice()函数创建。

特殊只读方法： 开始是下限;停止是上限; step是步长值;如果省略，则每个都是None。这些属性可以是任何类型。
切片对象提供的一个方法:

slice.indices(self,length) 此方法采用单个整数参数长度，并计算切片对象在应用于一系列长度项时将描述的切片信息。它返回一个由三个整数组成的元组；这些是开始和停止以及切片的步长或步幅。以与常规切片一致的方式处理丢失或越界。

2.13.5 静态方法对象

静态方法对象提供了一种破坏函数对象到上述实例方法对象的转换的方法。静态方法是对其他任何对象的包装器，通常是用户定义的方法对象。当从类或类实例中检索静态方法对象时，实际返回的对象是包装对象，该对象不受任何进一步转换的影响。静态方法对象本身不可调用，尽管它们通常包含的对象是静态方法对象由内置函数staticmethod()构造函数创建。

2.13.6 类方法对象

类方法对象(如静态方法对象)是另一个对象的包装器，它改变了从类和示例中检索该对象的方式。上面再"User-defined methods"下描述了类方法对象在这种检索时的行为。类方法对象由内置classmethod()构造函数创建。

3 特殊方法名称

类可以通过定义具有特殊名称的方法来实现由特殊语法(例如算术运算或下标或

切片)调用的某些操作。这是Python的**运算符重载**方法，允许类根据语言运算符定义自己的行为。例如：如果一个类定义了__getitem__()方法，且x是该类的一个实例，那么x[i]大致相当于type(x).__getitem__(x, i)。除非另有说明，否则在未定义适当方法时,尝试执行操作会引发异常(典型的有AttributeError或TypeError)。
将特殊方法设置为None表示相应操作不可取。例如，如果类将__iter__()设置为None，则改类不可迭代，因此在其实例上调用iter()会引发TypeError异常(不会回退到__getitem__())
在实现模拟任何内置类型的类时，重要的是只能将仿真实现到对建模对象有意义的程度。例如，一些序列可以很好的检索单个元素，但提取切片可能没有意义。(其中一个例子是W3C DOM中的NodeList接口。)

3.0.1 基本定制

object.__new__(cls[,...]) 调用其创建类cls的新实例。 __new__()是一个静态方法(特殊情况，因此你不需要声明它)，它将请求的类作为其第一个参数。其余参数传递给对象构造函数表达式(对类的调用)。__new__()返回值应该是新的实例对象(通常时cls的实例)。
典型的实现通过使用具有适当参数的super().__new__(cls[,...])调用超类的__new__()方法来创建类的新实例，然后在返回之前根据需要修改新创建的实例。
如果__new__()返回一个cls实例，那么新实例的__init__()方法将被调用，（否则将不会调用新实例__init__()方法）如__init__(self[,...]),其中self是新实例，其余参数与传入__new__()的相同。
__new__()主要用于允许不可变类型的子类(如int, str或tuple)的自定义实例创建。它也通常在自定义元类中重载，以自定义创建类。

object.__init__(self[,...]) 在创建实例(通过__new__())之后调用，但在将其返回给调用者之前调用。参数是传递给类构造函数表达式的参数。如果基类具有__init__()方法，则派生类的__init__()方法（如果有）必须显式调用它以确保正确初始化实例的基类部分；如super().__init__([args...])（注不需要self参数）。
因为__new__（）和__init__（）在构造对象时起作用（__new__（）来创建它，而__init__（）来自定义它），__init__（）不能返回非None值。这样做会导致在运行时引发TypeError。

object.__del__(self) 当实例即将被销毁时调用。这也称为终结器或（不当

地）析构函数。如果基类具有__del__（）方法，则派生类的__del__（）方法（如果有）必须显式调用它以确保正确删除实例的基类部分。
__del__（）方法可以（尽管不推荐!）通过创建对它的新引用来推迟实例的销毁。这称为物体复活*resurrection*。依赖于实现是否会在复活的对象即将被销毁时第二次调用__del__（）;当前的CPython实现只调用一次。
无法保证为解释器退出时仍然存在的对象调用__del__（）方法

Note: del x不直接调用x.__ del__（）——前者将x的引用计数递减1，而后者仅在x的引用计数达到零时调用。

CPython implementation detail: 引用循环可以防止对象引用计数变为零。在这种情况下，循环垃圾收集器稍后将检测并删除该循环。引用循环的常见原因是在局部变量中捕获到异常。然后帧的局部变量引用异常，该异常引起自己的回溯，该回溯引用回溯中捕获的所有帧的本地变量。

Warning: 由于调用__del__（）方法的不稳定情况，将忽略执行期间发生的异常，并向sys.stderr输出警告。特别是：

- 当执行任意代码时，可以调用__del__（），包括来自任意线程。如果__del__（）需要锁定或调用任何其他阻塞资源，它可能会死锁，因为执行__del__（）时被中断的代码已经占用了资源。
- 在解释器关闭期间可以执行__del__（）。因此，它需要访问的全局变量（包括其他模块）可能已被删除或设置为None。 Python保证在删除其他全局变量之前，从其模块中删除名称以单个下划线开头的全局变量;如果不存在对这样的全局变量的其他引用，这可能有助于确保在调用__del__（）方法时导入的模块仍然可用。

object.__repr__(self) 由内置函数repr()调用以计算对象的”官方”字符串表示。如果可能的话，这应该看起来像一个有效的Python表达式，可用于重新创建具有相同值的对象(给定适当环境)。如果这不可能，则应返回¡... some useful description...¿形式的字符串。返回值必须是字符串对象。如果一个类定义了__repr__（）而不是__str__（），那么当需要该类的实例的“非正式”字符串表示时，也会使用__repr__（）

object.__str__(self) 由str(object)和内置函数format()和print()调用，以计算对象的“非正式”或可良好打印的字符串表示形式。返回值必须是字符串对象。

此方法与object.__repr()不同之处在于，不期望__str__()返回有效的Python表达式:可以以使用更方便或更简洁的表示。
内置类型对象定义的默认实现调用object.__repr__()

object.__bytes__(self) 由bytes调用以计算对象的字节串表示。这应该返回一个bytes对象。

object.__format__(self,format_spec) 由format()内置函数调用，并通过扩展，计算格式化字符串字面值和str.format()方法，以生成对象的"格式化"字符串表示。format_spec参数是一个字符串，其中包含所需格式化选项的说明。format_spec参数的解释取决于实现__format__()的类型，但是大多数类会将格式化委托给其中一个内置类型，或者使用类似的格式化选项语法。
有关标准格式语法的说明，请参阅Format Specification Mini-Language。
object.__format__(x, '') is now equivalent to str(x) rather than format(str(self), '').

rich comparison富比较

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

这些是所谓的"富比较"方法。运算符和方法名称之间的对应关系如下:x¡y调用x.__lt__(y), x¡=y calls x.__le__(y), x==y calls x.__eq__(y), x!=y calls x.__ne__(y), x¿y calls x.__gt__(y), and x¿=y calls x.__ge__(y)。
如果没有为给定的参数对实现操作，那么丰富的比较方法可能会返回单例NotImplemented。按照惯例，返回False和True以进行成功比较。但是，这些方法可以返回任何值，因此如果在布尔上下文中使用比较运算符（例如，在if语句的条件下），Python将对值调用bool（）以确定结果是true还是false。

默认情况，__ne__()委托给__eq__()并反转结果，除非其NotImplemented。比较运算符之间没有其他隐含的关系，例如，(x¡y or x==y)并不意味着x¡=y。要从单个根操作自动生成排序操作，请参阅functools.total_ordering()。
有关创建支持自定义比较操作的可哈希对象的一些重要说明，请参阅__hash__()段落，并可用作字典键。
这些方法没有交换参数版本(当左参数不支持操作但右参数支持时使用)；相反，__lt__()和__gt__()是彼此相反的映射，__le__()和__ge__()是彼此的映射，而__eq__()和__ne__()亦是。如果操作数的类型不同，且右操作数的类型是左操作数类型的直接或间接子类，则右操作数的映射方法具有优先级，否则左操作数的方法具有优先级。不考虑虚拟子类化。

object.__hash__(self) 由内置函数hash()调用，以及对散列集合成员的操作，包括set,frozenset 和dict.__hash__()应该返回一个整数。唯一需要的属性是比较相等的对象具有相同的哈希值；建议将对象组件的哈希值混合在一起，这些哈希值也通过将对象打包成元组并对元组进行散列来对比对象。
如果类没有定义__eq__（）方法，它也不应该定义__hash__（）操作;如果它定义了__eq__（）而不是__hash__（），它的实例将不能用作hashable集合中的项目。如果一个类定义了可变对象并实现了一个__eq__（）方法，它就不应该实现__hash__（），因为hashable集合的实现需要一个键的哈希值是不可变的（如果对象的哈希值发生变化，那么它将是错误的哈希桶）。
用户定义类默认有__eq__()和__hash__()方法；拥有它们，使得所有对象都比较不相等(除了他自己)且x.__hash__()返回一个合适的值，使得x == y意味着x is y且hash(x) == hash(y)。
覆盖__eq__（）并且未定义__hash__（）的类将__hash__（）隐式设置为None。当类的__hash__（）方法为None时，类的实例将在程序尝试检索其哈希值时引发相应的TypeError，并且在检查isinstance时也将被正确识别为不可用（obj，collections.abc.Hashable ）。
如果覆盖__eq __（）的类需要从父类保留__hash __（）的实现，则必须通过设置__hash__ = ¡ParentClass¿ .__ hash__来明确告知解释器。

object.__bool__(self) 如果未定义此方法，则调用__len__（）（如果已定义），如果对象的结果非零，则认为该对象为true。如果类既不定义__len__（）也不定义__bool__（），则其所有实例都被视为true。

3.1 自定义访问属性权限

可以定义以下方法来自定义类实例的属性访问（使用，赋值或删除x.name）的含义。

object.__getattr__(self,name) 当默认属性访问因AttributeError失败时调用(要么__getattribute__()引发AtributeError，因*name*不是实例属性或类中属性；或者*name*属性的__get__()引发AttributeError)。此方法应当返回(计算)属性值或也引发一个AttributeError异常。
注意，如果通过常规机制找到该属性，则**不会调用__getattr__()**（这是__getattr__()和__setattr__()之间的故意不对称）。这样做既出于效率原因，又因为__getattr__()无法访问实例的其他属性。注意：至少对于实例变量，你可以通过不在实例属性字典中插入任何值来伪造总控制(而是将它们插入另一个对象)。请参阅__getattribute__()方法，以获得实际获得对属性访问的完全控制方法。

object.__getattribute__(self,name) 无条件调用以实现类的实例属性访问。如果该类还定义了__getattr_()，则除非__getattribute__()显式调用它或引发AttributeError，否则不会调用后者。此方法应当返回(计算)属性值或也引发一个AttributeError异常。为了避免此方法中的无限递归，其实现应始终调用具有相同名称的基类方法来访问其所需的任何属性，例如，
object.__ getattribute__（self，name）。

object.__setattr__(self,name,value) 在尝试进行属性分配时调用。在常规机制时调用？（例如在实例字典中存储值）。*name*是属性名，*value*赋值给属性。

如果__setattr__（）想要分配给实例属性，它应该调用具有相同名称的基类方法，例如object.__ setattr__（self，name，value）。

object.__delattr__(self,name) 像__setattr__（）但是对于属性删除而不是赋值。只有当del obj.name对对象有意义时才应该实现。

object.__dir__(self) 在对象上调用dir（）时调用。必须返回一个序列。 dir（）将返回的序列转换为列表并对其进行排序。

3.1.1 自定义模块属性访问

特殊名称__getattr__和__dir__也可用于自定义对模块属性的访问。模块级别的__getattr__函数应接受一个参数，该参数是属性的名称并返回计算值或引发AttributeError。如果通过正常查找在模块对象上找不到属性，即object.__ getattribute__（），则在引发AttributeError之前，在模块__dict__中搜索__getattr__。
__dir__函数不应接受任何参数，并返回表示模块上可访问的名称的字符串列表。如果存在，此函数将覆盖模块上的标准dir（）搜索。
为了更精细地定制模块行为（设置属性，属性等），可以将模块对象的__class__属性设置为types.ModuleType的子类。

3.1.2 实现描述符

以下方法仅在包含该方法的类的实例（所谓的描述符类*descriptorclass*）出现在*owner*所有者类中时才适用（该描述符必须位于拥有者的类字典中或其父类之一的字典中）。在下面的示例中，"the attribute"是指属性，其名称是拥有者类__dict__中属性的键。

object.__get__(self,instance,owner) 调用以获取所有者类（类属性访问）或该类的实例（实例属性访问）的属性。owner始终是所有者类，而instance是通过所有者访问属性时访问该属性的实例，或者是None。此方法应返回（计算）属性值或引发AttributeError异常。

object.__set__(self,instance,value) 调用以将所有者类的实例*instance*上的属性设置为新值value。

object.__delete__(self,instance) 被调用以删除所有者类的实例*instance*上的属性

object.__set_name_(self,owner,name) 在创建拥有类*owner*时调用。描述符已分配给name。

3.1.3 调用描述符

通常，描述符是具有“绑定行为”的对象属性，其属性访问权已被描述符协议中的方法覆盖：__get __（），__set__（）和__delete__（）。如果为对象定义了任何这些方法，则称其为描述符。

属性访问的默认行为是从对象的字典中获取，设置或删除属性。例如，a.x有一个以a.__dict__ ['x']开头的查找链，然后键入type(a).__dict__ ['x']，并继续通过type(a)的基类查找除了元类metaclasses。
但是，如果查找的值是定义其中一个描述符方法的对象，则Python可以覆盖默认行为并调用描述符方法。在优先级链中发生这种情况取决于定义了哪些描述符方法以及如何调用它们。
描述符调用的起点是绑定，a.x。如何组装参数取决于：

Direct Call 最简单和最不常见的调用是当用户代码直接调用描述符方法时：x .__get__（a）

Instance Binding 如果绑定到对象实例，则将a.x转换为调用：type(a).__dict__['x'].__get__(a,type(a))

Class Binding 如果绑定到类，则将A.x转换为调用：A.__dict__['x'] .__get__（None，A）。

Super Binding 如果a是super的实例，那么绑定super(B,obj).m()会在B之前的基类A中搜索obj.__class__.__mro__，然后通过调用调用描述符:A.__dict__['m'].__get__(obj, obj.__class__)。
对于实例绑定，描述符调用的优先级取决于定义的描述符方法。描述符可以定义__get__() __set__() __delete__()的任意组合。如果它没有定义__get__（），那么访问该属性将返回描述符对象本身，除非对象的实例字典中有值。如果描述符定义__set__（）和/或__delete__（），则它是一个数据描述符;如果它都不定义，则它是非数据描述符。
通常，数据描述符定义__get__（）和__set__（），而非数据描述符只有__get__（）方法。定义了__set__（）和__get__（）的数据描述符总是覆盖实例字典中的重定义。相反，非数据描述符可以被实例覆盖。
Python方法（包括staticmethod（）和classmethod（））作为非数据描述符实现。因此，实例可以重新定义和覆盖方法。这允许单个实例获取与同一类的其他实例不同的行为。
property（）函数实现为数据描述符。因此，实例不能覆盖属性的行为。

3.1.4 __slots__

__slots__允许我们显式声明数据成员（如属性）并拒绝创建__dict__和__weakref__（除非在__slots__中明确声明或在父级中可用。）
使用__dict__节省空间很重要。

object.__slots__ 可以为此类变量分配字符串，可迭代或具有实例使用的变量名称的字符串序列。 __slots__为声明的变量保留空间，并阻止为每个实例自动创建__dict__和__weakref__。
关于使用__slots__的注意事项:

- 没有__slots__的类继承时，实例的__dict__和__weakref__属性将始终可访问。

- 如果没有__dict__变量，则无法为实例分配__slots__定义中未列出的新变量。尝试分配到不公开的变量名称会引发AttributeError。如果需要动态分配新变量，则将“__dict__”添加到__slots__声明中的字符串序列中。
- 如果没有每个实例的__weakref__变量，则定义__slots__的类不支持对其实例的弱引用。如果需要弱引用支持，则将“__weakref__”添加到__slots__声明中的字符串序列。
- 通过为每个变量名创建描述符（实现描述符），在类级别实现__slots__。因此，类属性不能用于为__slots__定义的实例变量设置默认值;否则，class属性将覆盖描述符赋值。
- __slots__声明的操作不仅限于定义它的类。在父母中声明的__slots__可用于子类。但是，子类将获得__dict__和__weakref__，除非它们还定义__slots__（它应该只包含任何其他的slots）。
- 如果类定义了也在基类中定义的slot，则基类slot定义的实例变量是不可访问的（除非直接从基类检索其描述符）。这使得程序的含义未定义。将来，可能会添加一项检查以防止这种情况发生。
- 非空__slots__不适用于从“可变长度”内置类型派生的类，如int，bytes和tuple。
- 可以将任何非字符串可迭代分配给__slots__。也可以使用映射;但是，将来可以为与每个键对应的值分配特殊含义。
- __class__赋值仅在两个类具有相同的__slots__时有效

- 可以使用具有多个slots父类的多重继承，但只允许一个父级具有由slots创建的属性（其他基类必须具有空slot布局）——违规会引发TypeError

3.2 自定义类创建

每当一个类继承自另一个类时，就会在该类上调用__init_subclass__。这样，就可以编写改变子类行为的类。这与类装饰器密切相关，但是类装饰器只影响它们应用的特定类，__ init_subclass__仅适用于定义了方法的类的未来子类。

clssmethod object.__init_subclass__(cls) 只要包含类被子类化，就会调用此方法。然后cls是新的子类。如果定义为普通实例方法，则此方法将隐式转换为类方法。
给新类的关键字参数传递给父类__init_subclass__。为了与使用__init_subclass__的其他类兼容，应该取出所需的关键字参数并将其他参数传递给基类。
默认的实现obj.__ init_subclass__不执行任何操作，但如果使用任何参数调用它，则会引发错误。
Note: 元类暗示metaclass由其他类型机制使用，并且永远不会传递给__init_subclass__实现。实际的元类（而不是显式提示）可以作为type(cls)访问。

3.2.1 Metaclasses元类

默认情况下，使用type()构造类。类主体在新的命名空间中执行，类名在本地绑定到type(name, bases,namespace)的结果。
可以通过在类定义行中传递metaclass关键字参数，或从包含此类参数的现有类继承来自定义创建类过程。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。
在类定义中指定的任何其他关键字参数都将传递给下面描述的所有元类操作。
执行类定义时，会发生以下步骤：

- MRO条目已解决
- 确定适当的元类
- 类命名空间已准备好

- 类体被执行
- 类对象已创建

3.2.2 解决MRO条目

如果类定义中出现的base不是类型的实例，则在其上搜索__mro_entries__方法。如果找到，则使用原始bases元组调用它。此方法必须返回将使用类的元组而不是此base。元组可以是空的，在这种情况下，忽略原始base。

3.2.3 确定适当的元类

- 如果没有给出base和没有明确的元类，则使用type（）
- 如果给出了显式元类并且它不是type（）的实例，那么它将直接用作元类
- 如果给出了type（）的实例作为显式元类，或者定义了bases，则使用派生最多的元类

The most derived元类是从明确指定的元类（如果有）和所有指定基类的元类（即type（cls））中选择的。最衍生的元类是所有这些候选元类的子类型。如果没有候选元类符合该标准，则类定义将因TypeError而失败。

3.2.4 准备类命名空间

一旦识别出适当的元类，就会准备类命名空间。如果元类具有__prepare__属性，则将其称为namespace = metaclass.__ prepare __（name，bases，**kwds）（其中附加关键字参数（如果有）来自类定义）。
如果元类没有__prepare__属性，则类命名空间初始化为空有序映射

3.2.5 执行类实体

类主体（大致）执行为exec（body，globals（），namespace）。与正常调用exec（）的主要区别在于，当类定义发生在函数内部时，词法作用域允许类主体（包括任何方法）引用当前和外部作用域的名称。
但是，即使类定义发生在函数内部，类中定义的方法仍然无法看到在类范围内定义的名称。必须通过实例或类方法的第一个参数，或通过下一节中描述的隐式词法范围的__class__引用来访问类变量。

3.2.6 创建类对象

通过执行类主体填充类命名空间后，通过调用metaclass（name，bases，namespace，**kwds）创建类对象（此处传递的其他关键字与传递给__prepare__的关键字相同）。
这个类对象是将由super（）的零参数形式引用的类对象。 __class__是编译器创建的隐式闭包引用，如果类主体中的任何方法引用__class__或super。这允许super（）的零参数形式正确地识别基于词法作用域定义的类，而用于进行当前调用的类或实例基于传递给该方法的第一个参数来识别。

CPython implementation detail: 在CPython 3.6及更高版本中，__ class__单元作为类名称空间中的__classcell__条目传递给元类。如果存在，则必须将其传播到type.__ new__调用，以便正确初始化类。如果不这样做，将导致Python 3.6中的DeprecationWarning和Python 3.8中的RuntimeError。
使用默认元类类型或最终调用type.__ new__的任何元类时，在创建类对象后会调用以下其他自定义步骤：

- 首先，type.__ new__收集类命名空间中定义__set_name__（）方法的所有描述符;
- 第二，所有这些__set_name__方法都是在定义的类和特定描述符的指定名称的情况下调用的;和
- 最后，在方法解析顺序中，在新类的直接父级上调用__init_subclass__（）钩子。

创建类对象后，将其传递给类定义中包含的类装饰器（如果有），并将结果对象作为定义的类绑定在本地名称空间中。
当通过type.__new__创建新类时，作为命名空间参数提供的对象将复制到新的有序映射，并丢弃原始对象。新副本包装在只读代理中，该代理成为类对象的__dict__属性。

3.2.7 metaclasses的使用

元类的潜在用途是无限的。已探索的一些想法包括枚举，日志记录，接口检查，自动委托，自动属性创建，代理，框架和自动资源锁定/同步。

3.3 自定义实例和子类检查

以下方法用于覆盖isinstance（）和issubclass（）内置函数的默认行为。

略...前述未懂

3.4 模拟泛型类型

可以通过定义特殊方法来实现PEP 484（例如List [int]）指定的泛型类语法。

classmethod object.__class_getitem__(cls,key) 通过key中的类型参数返回表示泛型类特化的对象。
这个方法是在类对象本身上查找的，当在类体中定义时，该方法隐式地是一个类方法。注意，此机制主要用于静态类型提示，不鼓励使用其他用法。

3.5 模拟可调用对象

object.__call__(self[,args...]) 当实例作为函数是"called"时调用;x(arg1. arg2, ...)为x.__call__(arg1,arg2,...)的速记。

3.6 模拟容器类型

可以定义以下方法来实现容器对象。容器通常是序列（如列表或元组）或映射（如字典），但也可以代表其他容器。第一组方法用于模拟序列或模拟映射;不同之处在于，对于序列，允许键应该是整数k，其中0 ¡= k ¡N，其中N是序列的长度，或切片对象，它们定义了一系列条目。还建议映射提供方法keys（），values（），items（），get（），clear（），setdefault（），pop（），popitem（），copy（）和update（）行为类似于Python的标准字典对象。
collections.abc模块提供了一个MutableMapping抽象基类，以帮助从__getitem__（），__ setitem__（），__ delitem__（）和keys（）的基本集创建这些方法。可变序列应提供方法append（），count（），index（），extend（），insert（），pop（），remove（），reverse（）和sort（），如Python标准列表对象。最后，序列类型应该通过定义下面描述的方法__add__（），__radd__（），__iadd__（），__mul__（），__rmul__（）和__imul__（）来实现加法（意思是连接）和乘法（意思是重复）;他们不应该定义其他数字运算符。建议映射和序列都实现__contains__（）方法以允许有效使用in运算符;对于映射，应该搜索映射的键;对于序列，它应该搜索值。进一步建议映射和序列都实现__iter__（）方法，以允许有效迭代容器;对于映射，__user__（）应与keys（）相同;对于序列，它应该迭代值。

object.__len__(self) 被调用来实现内置函数len（）。应返回对象的长度，整数¿ = 0.此外，未定义__bool __（）方法且__len __（）方法返回零的对象在布尔上下文中被视为false。

CPython implementation detail: 在CPython中，长度必须至多为sys.maxsize。如果长度大于sys.maxsize，则某些功能（例如len（））可能会引发OverflowError。为了防止通过真值测试引发OverflowError，对象必须定义__bool__（）方法。

object.__length_hint__(self) 被调用以实现operator.length_hint（）。应返回对象的估计长度（可能大于或小于实际长度）。长度必须是¿ = 0的整数。此方法纯粹是一种优化，并且永远不需要正确性。

object.__getitem__(self,key) 调用以实现self[key]的等价。对于序列类型，接受的键应该是整数和切片对象。请注意，负索引的特殊解释（如果类希望模拟序列类型）取决于__getitem __()方法。如果key是不合适的类型，则可能引发TypeError;如果序列的索引集之外的值（在对负值进行任何特殊解释之后），则应引发IndexError。对于映射类型，如果缺少键（不在容器中），则应引发KeyError。

object.__setitem__(self,key,nalue) 被调用以实现对self[key]的分配。与__getitem __（）相同。如果对象支持对键值的更改，或者是否可以添加新键，或者对于可以替换元素的序列，则只应对映射实现此操作。对于__getitem __（）方法，应针对不正确的键值引发相同的异常。

object.__setitem__(self,key,value) 实现self[key]赋值。同上

object.__delitem__(self,key,value) 删除self[key]。同上

object.__missing__(self,key) 当字符不在字典中时，被dict.__getitem__()调用以实现dict子类的self[key]

object.__iter__(self) 当容器需要迭代器时，将调用此方法。此方法应返回一个新的迭代器对象，该对象可以迭代容器中的所有对象。对于映射，它应该迭代容器的键。

Iterator对象也需要实现此方法;他们被要求返回自己。有关迭代器对象的更多信息，请参阅迭代器类型。

object.__reversed__(self) 由reverse（）内置调用（如果存在）以实现反向迭代。它应该返回一个新的迭代器对象，它以相反的顺序迭代容器中的所有对象。
如果未提供__reversed __（）方法，则reverse（）内置将回退到使用序列协议（__len __（）和__getitem __（））。支持序列协议的对象应该只提供__reversed __（），如果它们可以提供比reverse（）提供的更有效的实现。
成员资格测试运算符（in和not in）通常实现为序列的迭代。但是，容器对象可以使用更高效的实现提供以下特殊方法，这也不要求对象是序列。

object.__contains__(self,item) 被调用以实现成员测试操作。如果item在self中，则返回true，否则返回false。对于映射对象，这应该考虑映射的key而不是value或key-item对。(zx:this is for 'in' operator)

3.7 模拟数字类型

可以定义以下方法来模拟数字对象。对应于所实现的特定种类数不支持的操作的方法（例如，非整数的按位操作）应保持未定义。

- 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二进制算术运算

$$+, -, \star, @, /, //, \%, divmod(), pow(), \star\star, \ll, \gg, \&, \wedge, \|$$

例如，要计算表达式x + y，其中x是具有__add__（）方法的类的实例，则调用x.__ add__（y）。 __divmod__（）方法应该等同于使用__floordiv__（）和__mod__（）;它不应该与__truediv__（）相关。请注意，如果要支持内置pow（）函数的三元版本，则应定义__pow__（）以接受可选的第三个参数。如果其中一个方法不支持使用提供的参数进行操作，则应返回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)
object.__rlshift__(self, other)
object.__rrshift__(self, other)
object.__rand__(self, other)
object.__rxor__(self, other)
object.__ror__(self, other)
仅当左操作数不支持相应的操作[3]并且操作数具有不同类型时，才调用这些函数。 [4]例如，要计算表达式x-y，其中y是具有__rsub__（）方法的类的实例，如果x.__sub__（y）返回NotImplemented，则调用y.__ rsub__（x）。
请注意，三元pow（）不会尝试调用__rpow__（）（强制规则会变得太复杂）。

NOTE 如果右操作数的类型是左操作数类型的子类，并且该子类提供了操作的反射方法，则此方法将在左操作数的非反射方法之前调用。此行为允许子类覆盖其祖先的操作。

- 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

$$+, =, -, \star =, @ =, / =, // =, \% =, \star\star =, \ll =, \gg =, \& =, \wedge =, | =$$

这些方法应该尝试就地执行操作（修改self）并返回结果（可能是，但不一定是self）。如果未定义特定方法，则扩充分配将回退到常规方法。例如，如果x是具有__iadd__（）方法的类的实例，则x += y等效于x = x.__ iadd__（y）。否则，与x + y的评估一样，考虑x.__ add__（y）和y.__ radd__（x）。在某些情况下，增强赋值可能会导致意外错误（请参阅为什么a_tuple[i] +=['item']在添加时会引发异常？），但这种行为实际上是数据模型的一部分。

- object.__neg__(self) 负号
- object.__pos__(self) 加号
- object.__abs__(self) abs()
- object.__invert__(self) 取反
- object.__complex__(self) complex()

- object.__int__(self) int()
- object.__float__(self) float()
- object.__index__(self)

调用实现operator.index（），并且只要Python需要无损地将数字对象转换为整数对象（例如在切片中，或在内置bin（），hex（）和oct（）函数中）。此方法的存在表明数字对象是整数类型。必须返回一个整数。

NOTE 为了得到一个连贯的整数类型，当定义__index __（）时，也应该定义__int__（），并且两者都应该返回相同的值

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

调用实现内置函数round（）和数学函数trunc（），floor（）和ceil（）。除非将ndigits传递给__round__（），否则所有这些方法都应该将截断的对象的值返回到Integral（通常是int）。
如果未定义__int__（），则内置函数int（）将回退到__trunc__（）。

3.8 With语句上下文管理器

上下文管理器是一个对象，它定义在执行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语句调用上下文管理器（在with语句一节中描述），但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途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。
有关上下文管理器的更多信息，请参阅Context Manager Types。

object.__enter__(self) 输入与此对象相关的运行时上下文。 with语句将此方法的返回值绑定到语句的as子句中指定的目标（如果有）。

object.__exit__(self,exc_type,exc_value,traceback) 退出与此对象相关的运行时上下文。参数描述导致退出上下文的异常。如果在没有异常的情况下退出上下文，则所有三个参数都将为None。
如果提供了异常，并且该方法希望抑制异常（即，防止它被传播），则它应该返回一个真值。否则，在退出此方法时将正常处理异常。

3.9 特殊方法查询

对于自定义类，只有在对象的类型上定义，而不是在对象的实例字典中，才能保证特殊方法的隐式调用正常工作。

- class C:pass
- c = C()
- c.__len__ = lambad: 5
- len(c) # ERROR

这种行为背后的基本原理在于许多特殊方法，例如__hash __（）和__repr __（），它们由所有对象实现，包括类型对象。如果这些方法的隐式查找使用传统的查找过程，则在类型对象本身上调用它们时会失败：

- 1.__hash__() == hash(1) True
- int.__hash__() == hash(int) ERROR

以这种方式错误地尝试调用类的未绑定方法有时被称为“(metaclass confusion)元类混淆”，并且在查找特殊方法时绕过实例可以避免：

- type(1).__hash(1) == hash(1) True
- type(int).__hash(int)__ == hash(int) True

除了为了正确性而绕过任何实例属性之外，隐式特殊方法查找通常也会绕过__getattribute__（）方法，甚至是对象的元类：

metaclass:

```
>>> class Meta(type):
...     def __getattribute__(*args):
...         print("Metaclass getattribute invoked")
...         return type.__getattribute__(*args)
...
>>> class C(object, metaclass=Meta):
...     def __len__(self):
...         return 10
...     def __getattribute__(*args):
...         print("Class getattribute invoked")
...         return object.__getattribute__(*args)
...
>>> c = C()
>>> c.__len__()                 # Explicit lookup via instance
Class getattribute invoked
10
>>> type(c).__len__(c)          # Explicit lookup via type
Metaclass getattribute invoked
10
>>> len(c)                      # Implicit lookup
10
```

以这种方式绕过__getattribute__()机制为解释器中的速度优化提供了很大的空间，代价是在处理特殊方法时的灵活性（必须在类对象本身上设置特殊方法才能一致地调用翻译器）。

4 协同程序(Coroutines)

4.1 Awaitable Objects

等待对象通常实现__await__()方法。从异步def函数返回的协程对象是等待的。

NOTE: 从types.coroutine()或asyncio.coroutine()修饰的生成器返回的生成器迭代器对象也是等待的，但它们没有实现__await__()。

object.__await__(self) 必须返回迭代器。应该用来实现等待对象。例如，asyncio.Future实现此方法以与await表达式兼容。

4.2 Coroutine Objects

协程对象是等待的(awaitable:可以在await表达式中使用的对象。可以是一个coroutine或具有__await__（）方法的对象)对象。可以通过调用__await__（）并迭代结果来控制协程的执行。当协程完成执行并返回时，迭代器引发StopIteration，异常的value属性保存返回值。如果协同程序引发异常，它将由迭代器传播。协程不应直接引发未处理的StopIteration异常。
协程也有下面列出的方法，类似于生成器的方法（参见Generator-iterator方法）。但是，与生成器不同，协同程序不直接支持迭代。

coroutine.send(value) 开始或恢复协程的执行。如果value为None，则这相当于推进由__await__（）返回的迭代器。如果value不是None，则此方法委托给迭代器的send（）方法，该方法导致协程挂起。结果（返回值，StopIteration或其他异常）与迭代__await__（）返回值时的结果相同，如上所述。

coroutine.throw(type[,value[,traceback]])) 在协程中引发指定的异常。如果它有这样的方法，则此方法委托给迭代器的throw（）方法，该方法导致协程挂起。否则，在暂停点处引发异常。结果（返回值，StopIteration或其他异常）与迭代__await__（）返回值时的结果相同，如上所述。如果协程中没有捕获异常，它会传播回调用方。

coroutine.close() 使协程自行清理并退出。如果协程被挂起，则此方法首先委托迭代器的close（）方法，该方法导致协程挂起(如果它有这样的方法)。然后它在挂起点引发GeneratorExit，使协同程序立即自行清理。最后，协程标记为已完成执行，即使它从未启动过。
当它们即将被销毁时，使用上述过程自动关闭协同程序对象。

4.3 异步迭代器(Asynchronout Iterators)

异步迭代器可以在__anext__方法中调用异步代码。
异步迭代器可以在async for语句中使用。

object.__aiter__(self) 必须返回异步迭代器对象。

object.__anext__(self) 必须返回可等待的对象，产生迭代器的下一个值。迭代结束时应该引发StopAsyncIteration错误。

从Python 3.7开始,__aiter__必须返回异步迭代器对象。返回任何其他内容将导致TypeError错误

4.4 异步上下文管理器

异步上下文管理器是一个context manager，它能够在其__aenter__和__aexit__方法中暂停执行。可以在async with语句中使用。

object.__anext__(self) 此方法在语义上类似于__enter__（），唯一的区别是它必须返回awaitable。

object.__aexit__(self,exc_type,exc_value,traceback) 这个方法在语义上类似于__exit__（），唯一的区别是它必须返回一个awaitable。