编程常见问题

目录

一般问题

Python 有没有提供带有断点、单步调试等功能的源码级调试器？

有的。

以下介绍了一些 Python 的调试器，用内置函数 `breakpoint()` 即可切入这些调试器中。

pdb 模块是一个简单但是够用的控制台模式 Python 调试器。 它是标准 Python 库的一部分，并且已收录于库参考手册。 你也可以通过使用 pdb 代码作为样例来编写你自己的调试器。

作为标准 Python 发行版组成部分的 IDLE 交互式开发环境 (通常位于 Tools/scripts/idle3)，包括一个图形化的调试器。

PythonWin 是一种 Python IDE，其中包含了一个基于 pdb 的 GUI 调试器。PythonWin 的调试器会为断点着色，并提供了相当多的超酷特性，例如调试非 PythonWin 程序等。PythonWin 是 pywin32 项目的组成部分，也是 ActivePython 发行版的组成部分。

Eric 是一个基于 PyQt 和 Scintilla 编辑组件的 IDE。

trepan3k 是一个类似 gdb 的调试器。

Visual Studio Code 是包含了调试工具的 IDE，并集成了版本控制软件。

有许多商业 Python IDE 都包含了图形化调试器。包括：

- Wing IDE
- Komodo IDE
- PyCharm

是否有能帮助寻找漏洞或执行静态分析的工具?

有的。

Pylint 和 Pyflakes 可执行基本检查来帮助你尽早捕捉漏洞。

静态类型检查器例如 Mypy, Pyre 和 Pytype 可以检查 Python 源代码中的类型提示。

如何由 Python 脚本创建能独立运行的二进制程序?

如果只是想要一个独立的程序，以便用户不必预先安装 Python 即可下载和运行它，则不需要将 Python 编译成 C 代码。有许多工具可以检测程序所需的模块，并将这些模块与 Python 二进制程序捆绑在一起生成单个可执行文件。

一种方案是使用 freeze 工具，它以 Tools/freeze 的形式包含在 Python 源代码树中。 它可将 Python 字节码转换为 C 数组；你可以使用 C 编译器将你的所有模块嵌入到一个新程序中，再将其与标准 Python 模块进行链接。

它的工作原理是递归扫描源代码，获取两种格式的 import 语句，并在标准 Python 路径和源码目录（用于内置模块）检索这些模块。然后，把这些模块的 Python 字节码转换为 C 代码（可以利用 marshal 模块转换为代码对象的数组初始化器），并创建一个定制的配置文件，该文件仅包含程序实际用到的内置模块。然后，编译生成的 C 代码并将其与 Python 解释器的其余部分链接，形成一个自给自足的二进制文件，其功能与 Python 脚本代码完全相同。

下列包可以用于帮助创建控制台和 GUI 的可执行文件:

- Nuitka （跨平台）
- PyInstaller (跨平台)
- PyOxidizer （跨平台）
- cx_Freeze （跨平台）
- py2app （仅限 macOS）
- py2exe (仅限 Windows)

是否有 Python 编码标准或风格指南?

有的。 标准库模块所要求的编码风格记录于 **PEP 8** 之中。

语言核心内容

变量明明有值，为什么还会出现 UnboundLocalError?

当在函数内部某处添加了一条赋值语句，因而导致之前正常工作的代码报出 `UnboundLocalError` 错误，这确实有点令人惊讶。

以下代码:

```
>>> x = 10
>>> def bar():
...     print(x)
...
>>> bar()
10
```

正常工作，但是以下代码

```
>>> x = 10
>>> def foo():
...     print(x)
...     x += 1
```

在 `UnboundLocalError` 中的结果:

```
>>> foo()
Traceback (most recent call last):
  ...
UnboundLocalError: local variable 'x' referenced before assignment
```

原因就是，当对某作用域内的变量进行赋值时，该变量将成为该作用域内的局部变量，并覆盖外部作用域中的同名变量。由于 foo 的最后一条语句为 `x` 分配了一个新值，编译器会将其识别为局部变量。因此，前面的 `print(x)` 试图输出未初始化的局部变量，就会引发错误。

在上面的示例中，可以将外部作用域的变量声明为全局变量以便访问:

```
>>> x = 10
>>> def foobar():
...     global x
```

```
...     print(x)
...     x += 1
...
>>> foobar()
10
```

与类和实例变量貌似但不一样，其实以上是在修改外部作用域的变量值，为了提示这一点，这里需要显式声明一下。

```
>>> print(x)
11
```

你可以使用 `nonlocal` 关键字在嵌套作用域中执行类似的操作：

```
>>> def foo():
...     x = 10
...     def bar():
...         nonlocal x
...         print(x)
...         x += 1
...     bar()
...     print(x)
...
>>> foo()
10
11
```

Python 的局部变量和全局变量有哪些规则？

函数内部只作引用的 Python 变量隐式视为全局变量。如果在函数内部任何位置为变量赋值，则除非明确声明为全局变量，否则均将其视为局部变量。

起初尽管有点令人惊讶，不过考虑片刻即可释然。一方面，已分配的变量要求加上 `global` 可以防止意外的副作用发生。另一方面，如果所有全局引用都要加上 `global` ，那处处都得用上 `global` 了。那么每次对内置函数或导入模块中的组件进行引用时，都得声明为全局变量。这种杂乱会破坏 `global` 声明用于警示副作用的有效性。

为什么在循环中定义的参数各异的 lambda 都返回相同的结果？

假设用 for 循环来定义几个取值各异的 lambda（即便是普通函数也一样）：

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
```

以上会得到一个包含5个 lambda 函数的列表，这些函数将计算 `x**2`。大家或许期望，调用这些函数会分别返回 `0` 、 `1` 、 `4` 、 `9` 和 `16`。然而，真的试过就会发现，他们都会返回 `16` ：

```
>>> squares[2]()
16
>>> squares[4]()
16
```

这是因为 `x` 不是 lambda 函数的内部变量，而是定义于外部作用域中的，并且 `x` 是在调用 lambda 时访问的——而不是在定义时访问。循环结束时 `x` 的值是 `4` ，所以此时所有的函数都将返回 `4**2` ，即 `16` 。通过改变 `x` 的值并查看 lambda 的结果变化，也可以验证这一点。

```
>>> x = 8
>>> squares[2]()
64
```

为了避免发生上述情况，需要将值保存在 lambda 局部变量，以使其不依赖于全局 `x` 的值：

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda n=x: n**2)
```

以上 `n=x` 创建了一个新的 lambda 本地变量 `n`，并在定义 lambda 时计算其值，使其与循环当前时点的 `x` 值相同。这意味着 `n` 的值在第 1 个 lambda 中为 `0` ，在第 2 个 lambda 中为 `1` ，在第 3 个中为 `2`，依此类推。因此现在每个 lambda 都会返回正确结果：

```
>>> squares[2]()
4
>>> squares[4]()
16
```

请注意，上述表现并不是 lambda 所特有的，常规的函数也同样适用。

如何跨模块共享全局变量?

在单个程序中跨模块共享信息的规范方法是创建一个特殊模块（通常称为 config 或 cfg）。只需在应用程序的所有模块中导入该 config 模块；然后该模块就可当作全局名称使用了。因为每个模块只有一个实例，所以对该模块对象所做的任何更改将会在所有地方得以体现。 例如：

config.py：

```
x = 0   # 'x' 配置设置的默认值
```

mod.py：

```
import config
config.x = 1
```

main.py：

```
import config
import mod
print(config.x)
```

请注意，出于同样的原因，使用模块也是实现单例设计模式的基础。

导入模块的"最佳实践"是什么?

通常请勿使用 `from modulename import *` 。因为这会扰乱 importer 的命名空间，且会造成未定义名称更难以被 Linter 检查出来。

请在代码文件的首部就导入模块。这样代码所需的模块就一目了然了，也不用考虑模块名是否在作用域内的问题。每行导入一个模块则增删起来会比较容易，每行导入多个模块则更节省屏幕空间。

按如下顺序导入模块就是一种好做法：

1. 标准库模块——例如：`sys`、`os`、`argparse`、`re` 等。
2. 第三方库模块（安装于 Python site-packages 目录中的内容）——例如：`dateutil`、`requests`、`PIL.Image` 等。
3. 本地开发的模块

为了避免循环导入引发的问题，有时需要将模块导入语句移入函数或类的内部。Gordon McMillan 的说法如下：

> 当两个模块都采用 "import <module>" 的导入形式时，循环导入是没有问题的。但如果第 2 个模块想从第 1 个模块中取出一个名称（"from module import name"）并且导入处于代码的最顶层，那导入就会失败。原因是第 1 个模块中的名称还不可用，这时第 1 个模块正忙于导入第 2 个模块呢。

如果只是在一个函数中用到第 2 个模块，那这时将导入语句移入该函数内部即可。当调用到导入语句时，第 1 个模块将已经完成初始化，第 2 个模块就可以进行导入了。

如果某些模块是平台相关的，可能还需要把导入语句移出最顶级代码。这种情况下，甚至有可能无法导入文件首部的所有模块。于是在对应的平台相关代码中导入正确的模块，就是一种不错的选择。

只有为了避免循环导入问题，或有必要减少模块初始化时间时，才把导入语句移入类似函数定义内部的局部作用域。如果根据程序的执行方式，许多导入操作不是必需的，那么这种技术尤其有用。如果模块仅在某个函数中用到，可能还要将导入操作移入该函数内部。请注意，因为模块有一次初始化过程，所以第一次加载模块的代价可能会比较高，但多次加载几乎没有什么花费，代价只是进行几次字典检索而已。即使模块名超出了作用域，模块在 `sys.modules` 中也是可用的。

为什么对象之间会共享默认值？

新手程序员常常中招这类 Bug。请看以下函数：

```
def foo(mydict={}):  # 危险：所有调用共享对一个字典的引用
    ... 执行一些计算 ...
    mydict[key] = value
    return mydict
```

第一次调用此函数时， `mydict` 中只有一个数据项。第二次调用 `mydict` 则会包含两个数据项，因为 `foo()` 开始执行时， `mydict` 中已经带有一个数据项了。

大家往往希望，函数调用会为默认值创建新的对象。但事实并非如此。默认值只会在函数定义时创建一次。如果对象发生改变，就如上例中的字典那样，则后续调用该函数时将会引用这个改动的对象。

按照定义，不可变对象改动起来是安全的，诸如数字、字符串、元组和 None 之类。而可变对象的改动则可能引起困惑，例如字典、列表和类实例等。

因此，不把可变对象用作默认值是一种良好的编程做法。而应采用 None 作为默认值，然后在函数中检查参数是否为 None 并新建列表、字典或其他对象。例如，代码不应如下所示：

```
def foo(mydict={}):
    ...
```

而应这么写：

```
def foo(mydict=None):
    if mydict is None:
        mydict = {}  # 为局部命名空间新建一个字典
```

参数默认值的特性有时会很有用处。 如果有个函数的计算过程会比较耗时，有一种常见技巧是将每次函数调用的参数和结果缓存起来，并在同样的值被再次请求时返回缓存的值。这种技巧被称为“memoize”，实现代码可如下所示：

```
# 调用方只能提供两个形参并可选择以关键字形式传入 _cache
def expensive(arg1, arg2, *, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # 计算结果值
    result = ... 高耗费的计算 ...
    _cache[(arg1, arg2)] = result           # 将结果保存在缓存中
    return result
```

也可以不用参数默认值来实现，而是采用全局的字典变量；这取决于个人偏好。

如何将可选参数或关键字参数从一个函数传递到另一个函数?

请利用函数参数列表中的标识符 * 和 ** 归集实参；结果会是元组形式的位置实参和字典形式的关键字实参。然后就可利用 * 和 ** 在调用其他函数时传入这些实参：

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

形参和实参之间有什么区别?

形参 是由出现在函数定义中的名称来定义的，而 参数 则是在调用函数时实际传入的值。 形参定义了一个函数能接受什么 参数种类。 例如，对于以下函数定义:

```
def func(foo, bar=None, **kwargs):
    pass
```

foo 、 *bar* 和 *kwargs* 是 func 的形参。 不过在调用 func 时，例如：

```
func(42, bar=314, extra=somevar)
```

`42`、`314` 和 `somevar` 则是实参。

为什么修改列表 'y' 也会更改列表 'x'？

如果代码编写如下：

```
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
[10]
>>> x
[10]
```

或许大家很想知道，为什么在 y 中添加一个元素时，x 也会改变。

产生这种结果有两个因素：

1. 变量只是指向对象的一个名称。执行 `y = x` 并不会创建列表的副本——而只是创建了一个新变量 `y`，并指向 `x` 所指的同一对象。这就意味着只存在一个列表对象，`x` 和 `y` 都是对它的引用。
2. 列表属于 mutable 对象，这意味着它的内容是可以修改的。

在调用 `append()` 之后，该可变对象的内容从 `[]` 变为 `[10]`。 由于两个变量引用了同一对象，因此使用其中任意一个名称访问的都是修改后的值 `[10]`。

如果把赋给 `x` 的对象换成一个不可变对象：

```
>>> x = 5  # 整数是不可变对象
>>> y = x
>>> x = x + 1  # 5 不能被修改，在此我们会新建一个对象
>>> x
6
>>> y
5
```

可见这时 `x` 和 `y` 就不再相等了。因为整数是 immutable 对象，在执行 `x = x + 1` 时，并不会修改整数对象 `5`，给它加上 1；而是创建了一个新的对象（整数对象 `6`）并将其赋给 `x`（也就是改变了 `x` 所指向的对象）。在赋值完成后，就有了两个对象（整数对象 `6` 和 `5`）和分别指向他俩的两个变量（`x` 现在指向 `6` 而 `y` 仍然指向 `5`）。

某些操作 (例如 `y.append(10)` 和 `y.sort()`) 是改变原对象，而看上去相似的另一些操作 (例如 `y = y + [10]` 和 `sorted(y)` <sorted>) 则是创建新对象。 通常在 Python 中 (以及在标准库的所有代码中) 会改变原对象的方法将返回 ``None`() 以帮助避免混淆这两种不同类型的操作。 因此如果你错误地使用了 `y.sort()` 并期望它将返回一个经过排序的 `y` 的副本，你得到的结果将会是 `None`，这将导致你的程序产生一个容易诊断的错误。

不过还存在一类操作，用不同的类型执行相同的操作有时会发生不同的行为：即增量赋值运算符。例如，`+=` 会修改列表，但不会修改元组或整数（`a_list += [1, 2, 3]` 与 `a_list.extend([1,`

`2, 3])` 同样都会改变 `a_list`，而 `some_tuple += (1, 2, 3)` 和 `some_int += 1` 则会创建新的对象）。

换而言之：

- 对于一个可变对象（ `list` 、 `dict` 、 `set` 等等），可以利用某些特定的操作进行修改，所有引用它的变量都会反映出改动情况。
- 对于一个不可变对象（ `str` 、 `int` 、 `tuple` 等），所有引用它的变量都会给出相同的值，但所有改变其值的操作都将返回一个新的对象。

如要知道两个变量是否指向同一个对象，可以利用 `is` 运算符或内置函数 `id()`。

如何编写带有输出参数的函数（按照引用调用）？

请记住，Python 中的实参是通过赋值传递的。由于赋值只是创建了对象的引用，所以调用方和被调用方的参数名都不存在别名，本质上也就不存在按引用调用的方式。通过以下几种方式，可以得到所需的效果。

1. 返回一个元组：

```
>>> def func1(a, b):
...     a = 'new-value'        # a 和 b 是局部名称
...     b = b + 1              # 赋值为新的对象
...     return a, b            # 返回新的值
...
>>> x, y = 'old-value', 99
>>> func1(x, y)
('new-value', 100)
```

   这差不多是最明晰的解决方案了。

2. 使用全局变量。这不是线程安全的方案，不推荐使用。

3. 传递一个可变（即可原地修改的） 对象：

```
>>> def func2(a):
...     a[0] = 'new-value'     # 'a' 引用了一个可变的列表
...     a[1] = a[1] + 1        # 修改一个共享对象
...
>>> args = ['old-value', 99]
>>> func2(args)
>>> args
['new-value', 100]
```

4. 传入一个接收可变对象的字典:

```
>>> def func3(args):
...     args['a'] = 'new-value'     # args 是一个可变的字典
...     args['b'] = args['b'] + 1   # 对其进行原地修改
...
>>> args = {'a': 'old-value', 'b': 99}
>>> func3(args)
```

```
>>> args
{'a': 'new-value', 'b': 100}
```

5. 或者把值用类实例封装起来：

```
>>> class Namespace:
...     def __init__(self, /, **args):
...         for key, value in args.items():
...             setattr(self, key, value)
...
>>> def func4(args):
...     args.a = 'new-value'        # args 是一个可变的 Namespace
...     args.b = args.b + 1         # 原地修改对象
...
>>> args = Namespace(a='old-value', b=99)
>>> func4(args)
>>> vars(args)
{'a': 'new-value', 'b': 100}
```

没有什么理由要把问题搞得这么复杂。

最佳选择就是返回一个包含多个结果值的元组。

如何在 Python 中创建高阶函数?

有两种选择：嵌套作用域、可调用对象。假定需要定义 `linear(a,b)`，其返回结果是一个计算出 `a*x+b` 的函数 `f(x)`。 采用嵌套作用域的方案如下：

```
def linear(a, b):
    def result(x):
        return a * x + b
    return result
```

或者可采用可调用对象：

```
class linear:

    def __init__(self, a, b):
        self.a, self.b = a, b

    def __call__(self, x):
        return self.a * x + self.b
```

采用这两种方案时：

```
taxes = linear(0.3, 2)
```

都会得到一个可调用对象，可实现 `taxes(10e6) == 0.3 * 10e6 + 2`。

可调用对象的方案有个缺点，就是速度稍慢且生成的代码略长。不过值得注意的是，同一组可调用对象能够通过继承来共享签名（类声明）：

```
class exponential(linear):
    # 继承了 __init__
```

```
    def __call__(self, x):
        return self.a * (x ** self.b)
```

对象可以为多个方法的运行状态进行封装：

```
class counter:

    value = 0

    def set(self, x):
        self.value = x

    def up(self):
        self.value = self.value + 1

    def down(self):
        self.value = self.value - 1

count = counter()
inc, dec, reset = count.up, count.down, count.set
```

以上 `inc()`、`dec()` 和 `reset()` 的表现，就如同共享了同一计数变量一样。

如何复制 Python 对象？

一般情况下，用 `copy.copy()` 或 `copy.deepcopy()` 基本就可以了。并不是所有对象都支持复制，但多数是可以的。

某些对象可以用更简便的方法进行复制。比如字典对象就提供了 `copy()` 方法：

```
newdict = olddict.copy()
```

序列可以用切片操作进行复制：

```
new_l = l[:]
```

如何找到对象的方法或属性？

对于一个用户定义类的实例 `x`，`dir(x)` 将返回一个按字母顺序排列的名称列表，其中包含实例属性及由类定义的方法和属性。

如何用代码获取对象的名称？

一般而言这是无法实现的，因为对象并不存在真正的名称。赋值本质上是把某个名称绑定到某个值上；`def` 和 `class` 语句同样如此，只是值换成了某个可调用对象。比如以下代码：

```
>>> class A:
...     pass
...
>>> B = A
>>> a = B()
>>> b = a
>>> print(b)
```

```
<__main__.A object at 0x16D07CC>
>>> print(a)
<__main__.A object at 0x16D07CC>
```

可以不太严谨地说上述类有一个名称：即使它绑定了两个名称并通过名称 `B` 唤起所创建的实例仍将被报告为类 `A` 的实例。 但是，没有办法肯定地说实例的名称是 `a` 还是 `b`，因为这两个名称都被绑定到同一个值上了。

代码一般没有必要去"知晓"某个值的名称。通常这种需求预示着还是改变方案为好，除非真的是要编写内审程序。

在 comp.lang.python 中，Fredrik Lundh 在回答这样的问题时曾经给出过一个绝佳的类比：

> 这就像要知道家门口的那只猫的名字一样：猫（对象）自己不会说出它的名字，它根本就不在乎自己叫什么——所以唯一方法就是问一遍你所有的邻居（命名空间），这是不是他们家的猫（对象）……
>
> ……并且如果你发现它有很多名字或根本没有名字，那也不必惊讶！

逗号运算符的优先级是什么？

逗号不是 Python 的运算符。 请看以下例子：

```
>>> "a" in "b", "a"
(False, 'a')
```

由于逗号不是运算符，而只是表达式之间的分隔符，因此上述代码就相当于：

```
("a" in "b"), "a"
```

而不是：

```
"a" in ("b", "a")
```

对于各种赋值运算符（`=`、`+=` 等）来说同样如此。他们并不是真正的运算符，而只是赋值语句中的语法分隔符。

是否提供等价于 C 语言 "?:" 三目运算符的东西？

有的。语法如下：

```
[on_true] if [expression] else [on_false]

x, y = 50, 25
small = x if x < y else y
```

在 Python 2.5 引入上述语法之前，通常的做法是使用逻辑运算符：

```
[expression] and [on_true] or [on_false]
```

然而这种做法并不保险，因为当 *on_true* 为布尔值"假"时，结果将会出错。所以肯定还是采用 `... if ... else ...` 形式为妙。

是否可以用 Python 编写让人眼晕的单行程序?

可以。 这一般是通过在 `lambda` 中嵌套 `lambda` 来实现的。 请参阅以下三个示例，它们是基于 Ulf Bartelt 的代码改写的:

```
from functools import reduce

# < 1000 的质数
print(list(filter(None,map(lambda y:y*reduce(lambda x,y:x*y!=0,
map(lambda x,y=y:y%x,range(2,int(pow(y,0.5)+1))),1),range(2,1000)))))

# 前 10 个斐波那契数字
print(list(map(lambda x,f=lambda x,f:(f(x-1,f)+f(x-2,f)) if x>1 else 1:
f(x,f), range(10))))

# 曼德布罗集
print((lambda Ru,Ro,Iu,Io,IM,Sx,Sy:reduce(lambda x,y:x+'\n'+y,map(lambda y,
Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,Sy=Sy,L=lambda yc,Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,i=IM,
Sx=Sx,Sy=Sy:reduce(lambda x,y:x+y,map(lambda x,xc=Ru,yc=yc,Ru=Ru,Ro=Ro,
i=i,Sx=Sx,F=lambda xc,yc,x,y,k,f=lambda xc,yc,x,y,k,f:(k<=0)or (x*x+y*y
>=4.0) or 1+f(xc,yc,x*x-y*y+xc,2.0*x*y+yc,k-1,f):f(xc,yc,x,y,k,f):chr(
64+F(Ru+x*(Ro-Ru)/Sx,yc,0,0,i)),range(Sx))):L(Iu+y*(Io-Iu)/Sy),range(Sy
))))(-2.1, 0.7, -1.2, 1.2, 30, 80, 24))
#     \___ ___/  \___ ___/  |   |   |__ 屏幕上的行
#         V          V      |   |______ 屏幕上的列
#         |          |      |_________ "迭代"的最大次数
#         |          |________________ y 轴上的取值范围
#         |___________________________ x 轴上的取值范围
```

请不要在家里尝试，骚年!

函数形参列表中的斜杠（/）是什么意思?

函数参数列表中的斜杠表示在它之前的形参都是仅限位置形参。 仅限位置形参没有可供外部使用的名称。 在调用接受仅限位置形参的函数时，参数将只根据其位置被映射到形参上。 例如，`divmod()` 就是一个接受仅限位置形参的函数。 它的文档说明是这样的:

```
>>> help(divmod)
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

形参列表尾部的斜杠说明，两个形参都是仅限位置形参。因此，用关键字参数调用 `divmod()` 将会引发错误:

```
>>> divmod(x=3, y=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: divmod() takes no keyword arguments
```

数字和字符串

如何给出十六进制和八进制整数?

要给出八进制数，需在八进制数值前面加上一个零和一个小写或大写字母 "o" 作为前缀。例如，要将变量 "a" 设为八进制的 "10" （十进制的 8），写法如下：

```
>>> a = 0o10
>>> a
8
```

十六进制数也很简单。只要在十六进制数前面加上一个零和一个小写或大写的字母 "x"。十六进制数中的字母可以为大写或小写。比如在 Python 解释器中输入：

```
>>> a = 0xa5
>>> a
165
>>> b = 0XB2
>>> b
178
```

为什么 -22 // 10 会返回 -3？

这主要是为了让 `i % j` 的正负与 `j` 一致，如果期望如此，且期望如下等式成立：

```
i == (i // j) * j + (i % j)
```

那么整除就必须返回向下取整的结果。C 语言同样要求保持这种一致性，于是编译器在截断 `i // j` 的结果时需要让 `i % j` 的正负与 `i` 一致。

对于 `i % j` 来说 `j` 为负值的应用场景实际上是非常少的。 而 `j` 为正值的情况则非常多，并且实际上在所有情况下让 `i % j` 的结果为 `>= 0` 会更有用处。 如果现在时间为 10 时，那么 200 小时前应是几时？ `-190 % 12 == 2` 是有用处的；`-190 % 12 == -10` 则是会导致意外的漏洞。

我如何获得 int 字面属性而不是 SyntaxError？

尝试以正式方式查找一个 `int` 字面值属性会发生 `SyntaxError` 因为句点会被当作是小数点:

```
>>> 1.__class__
  File "<stdin>", line 1
  1.__class__
   ^
SyntaxError: invalid decimal literal
```

解决办法是用空格或括号将字词与句号分开。

```
>>> 1 .__class__
<class 'int'>
>>> (1).__class__
<class 'int'>
```

如何将字符串转换为数字?

对于整数，可使用内置的 `int()` 类型构造器，例如 `int('144') == 144`。 类似地，可使用 `float()` 转换为浮点数，例如 `float('144') == 144.0`。

默认情况下，这些操作会将数字按十进制来解读，因此 `int('0144') == 144` 为真值，而 `int('0x144')` 会引发 `ValueError`。 `int(string, base)` 接受第二个可选参数指定转换的基数，例如 `int( '0x144', 16) == 324`。 如果指定基数为 0，则按 Python 规则解读数字：前缀 '0o' 表示八进制，而 '0x' 表示十六进制。

如果只是想把字符串转为数字，请不要使用内置函数 `eval()`。 `eval()` 的速度慢很多且存在安全风险：别人可能会传入带有不良副作用的 Python 表达式。比如可能会传入 `__import__('os').system("rm -rf $HOME")`，这会把 home 目录给删了。

`eval()` 还有把数字解析为 Python 表达式的后果，因此如 `eval('09')` 将会导致语法错误，因为 Python 不允许十进制数带有前导 '0'（'0' 除外）。

如何将数字转换为字符串?

例如，要把数字 `144` 转换为字符串 `'144'`，可使用内置类型构造器 `str()`。 如果你需要十六进制或八进制表示形式，可使用内置函数 `hex()` 或 `oct()`。 更复杂的格式化方式，请参阅 f-字符串 和 格式字符串语法 等章节，例如 `"{:04d}".format(144)` 将产生 `'0144'` 而 `"{:.3f}".format(1.0/3.0)` 将产生 `'0.333'`。

如何修改字符串?

无法修改，因为字符串是不可变对象。 在大多数情况下，只要将各个部分组合起来构造出一个新字符串即可。如果需要一个能原地修改 Unicode 数据的对象，可以试试 `io.StringIO` 对象或 `array` 模块：

```
>>> import io
>>> s = "Hello, world"
>>> sio = io.StringIO(s)
>>> sio.getvalue()
'Hello, world'
>>> sio.seek(7)
7
>>> sio.write("there!")
6
>>> sio.getvalue()
'Hello, there!'

>>> import array
>>> a = array.array('w', s)
>>> print(a)
array('w', 'Hello, world')
>>> a[0] = 'y'
>>> print(a)
array('w', 'yello, world')
```

```
>>> a.tounicode()
'yello, world'
```

如何使用字符串调用函数/方法?

有多种技巧可供选择。

- 最好的做法是采用一个字典，将字符串映射为函数。其主要优势就是字符串不必与函数名一样。这也是用来模拟 case 结构的主要技巧：

```
def a():
    pass

def b():
    pass

dispatch = {'go': a, 'stop': b}  # 注意函数名后不带圆括号

dispatch[get_input()]()  # 注意末尾要带圆括号以调用函数
```

- 利用内置函数 getattr() ：

```
import foo
getattr(foo, 'bar')()
```

  请注意 getattr() 可用于任何对象，包括类、类实例、模块等等。

  标准库就多次使用了这个技巧，例如：

```
class Foo:
    def do_foo(self):
        ...

    def do_bar(self):
        ...

f = getattr(foo_instance, 'do_' + opname)
f()
```

- 用 locals() 解析出函数名：

```
def myFunc():
    print("hello")

fname = "myFunc"

f = locals()[fname]
f()
```

是否有 Perl 的 chomp() 等价物用于从字符串中移除末尾换行符?

可以使用 `S.rstrip("\r\n")` 从字符串 `S` 的末尾删除所有的换行符，而不删除其他尾随空格。如果字符串 `S` 表示多行，且末尾有几个空行，则将删除所有空行的换行符：

```
>>> lines = ("line 1 \r\n"
...          "\r\n"
...          "\r\n")
>>> lines.rstrip("\n\r")
'line 1 '
```

由于通常只在一次读取一行文本时才需要这样做，所以使用 `S.rstrip()` 这种方式工作得很好。

是否有 scanf() 或 sscanf() 的等价物?

没有。

对于简单的输入解析，最简单的方法通常是使用字符串对象的 `split()` 方法将行分割为空白符分隔的单词，然后使用 `int()` 或 `float()` 将十进制字符串转换为数字值 。 `split()` 支持可选的 "sep" 形参 ，如果行中使用空白符以外的其他分隔符，可以使用该参数。

对于更复杂的输入解析，正则表达式相比 C 的 `sscanf` 更为强大也更为适合。

UnicodeDecodeError 或 UnicodeEncodeError 错误的含义是什么?

见 Unicode 指南

我能以奇数个反斜杠来结束一个原始字符串吗?

以奇数个反斜杠结尾的原始字符串将会转义用于标记字符串的引号:

```
>>> r'C:\this\will\not\work\'
  File "<stdin>", line 1
    r'C:\this\will\not\work\'
    ^
SyntaxError: unterminated string literal (detected at line 1)
```

有几种绕过此问题的办法。 其中之一是使用常规字符串以及双反斜杠:

```
>>> 'C:\\this\\will\\work\\'
'C:\\this\\will\\work\\'
```

另一种办法是将一个包含被转义反斜杠的常规字符串拼接到原始字符串上:

```
>>> r'C:\this\will\work' '\\'
'C:\\this\\will\\work\\'
```

在 Windows 上还可以使用 `os.path.join()` 来添加反斜杠:

```
>>> os.path.join(r'C:\this\will\work', '')
'C:\\this\\will\\work\\'
```

请注意虽然在确定原始字符串的结束位置时反斜杠会对引号进行“转义”，但在解析原始字符串的值时并不会发生转义。 也就是说，反斜杠会被保留在原始字符串的值中:

```
>>> r'backslash\'preserved'
"backslash\\'preserved"
```

另请参阅 语言参考 中的规范说明。

性能

我的程序太慢了。该如何加快速度？

总的来说，这是个棘手的问题。在进一步讨论之前，首先应该记住以下几件事：

- 不同的 Python 实现具有不同的性能特点。 本 FAQ 着重解答的是 CPython。
- 不同操作系统可能会有不同表现，尤其是涉及 I/O 和多线程时。
- 在尝试优化代码 *之前*，务必要先找出程序中的热点（请参阅 `profile` 模块）。
- 编写基准测试脚本，在寻求性能提升的过程中就能实现快速迭代（请参阅 `timeit` 模块）。
- 强烈建议首先要保证足够高的代码测试覆盖率（通过单元测试或其他技术），因为复杂的优化有可能会导致代码回退。

话虽如此，Python 代码的提速还是有很多技巧的。以下列出了一些普适性的原则，对于让性能达到可接受的水平会有很大帮助：

- 相较于试图对全部代码铺开做微观优化，优化算法（或换用更快的算法）可以产出更大的收益。
- 使用正确的数据结构。参考 内置类型 和 `collections` 模块的文档。
- 如果标准库已为某些操作提供了基础函数，则可能（当然不能保证）比所有自编的函数都要快。对于用 C 语言编写的基础函数则更是如此，比如内置函数和一些扩展类型。例如，一定要用内置方法 `list.sort()` 或 `sorted()` 函数进行排序（某些高级用法的示例请参阅 排序的技术 ）。
- 抽象往往会造成中间层，并会迫使解释器执行更多的操作。如果抽象出来的中间层级太多，工作量超过了要完成的有效任务，那么程序就会被拖慢。应该避免过度的抽象，而且往往也会对可读性产生不利影响，特别是当函数或方法比较小的时候。

如果你已经达到纯 Python 允许的限制，那么有一些工具可以让你走得更远。 例如，Cython 可以将稍加修改的 Python 代码版本编译为 C 扩展，并能在许多不同的平台上使用。 Cython 可以利用编译（和可选的类型标注）来让你的代码显著快于解释运行时的速度。 如果你对自己的 C 编程技能有信心，还可以自行 编写 C 扩展模块。

> **参见：** 专门介绍 性能提示 的wiki页面。

将多个字符串连接在一起的最有效方法是什么？

`str` 和 `bytes` 对象是不可变的，因此连接多个字符串的效率会很低，因为每次连接都会创建一个新的对象。一般情况下，总耗时与字符串总长是二次方的关系。

如果要连接多个 `str` 对象，通常推荐的方案是先全部放入列表，最后再调用 `str.join()`：

```
chunks = []
for s in my_strings:
    chunks.append(s)
result = ''.join(chunks)
```

（还有一种合理高效的习惯做法，就是利用 `io.StringIO` ）

如果要连接多个 `bytes` 对象，推荐做法是用 `bytearray` 对象的原地连接操作（ `+=` 运算符）追加数据：

```
result = bytearray()
for b in my_bytes_objects:
    result += b
```

序列（元组/列表）

如何在元组和列表之间进行转换？

类型构造器 `tuple(seq)` 可将任意序列（实际上是任意可迭代对象）转换为数据项和顺序均不变的元组。

例如，`tuple([1, 2, 3])` 会生成 `(1, 2, 3)` ， `tuple('abc')` 则会生成 `('a', 'b', 'c')` 。 如果参数就是元组，则不会创建副本而是返回同一对象，因此如果无法确定某个对象是否为元组时，直接调用 `tuple()` 也没什么代价。

类型构造器 `list(seq)` 可将任意序列或可迭代对象转换为数据项和顺序均不变的列表。例如，`list((1, 2, 3))` 会生成 `[1, 2, 3]` 而 `list('abc')` 则会生成 `['a', 'b', 'c']`。如果参数即为列表，则会像 `seq[:]` 那样创建一个副本。

什么是负数索引？

Python 序列的索引可以是正数或负数。索引为正数时，0 是第一个索引值， 1 为第二个，依此类推。索引为负数时，-1 为倒数第一个索引值，-2 为倒数第二个，依此类推。可以认为 `seq[-n]` 就相当于 `seq[len(seq)-n]`。

使用负数序号有时会很方便。 例如 `S[:-1]` 就是原字符串去掉最后一个字符，这可以用来移除某个字符串末尾的换行符。

序列如何以逆序遍历？

使用内置函数 `reversed()` ：

```
for x in reversed(sequence):
    ...  # 对 x 执行某些操作 ...
```

原序列不会变化，而是构建一个逆序的新副本以供遍历。

如何从列表中删除重复项？

许多完成此操作的的详细介绍，可参阅 Python Cookbook：

https://code.activestate.com/recipes/52560/

如果列表允许重新排序，不妨先对其排序，然后从列表末尾开始扫描，依次删除重复项：

```
if mylist:
    mylist.sort()
    last = mylist[-1]
    for i in range(len(mylist)-2, -1, -1):
        if last == mylist[i]:
            del mylist[i]
        else:
            last = mylist[i]
```

如果列表的所有元素都能用作集合的键（即都是 hashable ），以下做法速度往往更快：

```
mylist = list(set(mylist))
```

以上操作会将列表转换为集合，从而删除重复项，然后返回成列表。

如何从列表中删除多个项?

类似于删除重复项，一种做法是反向遍历并根据条件删除。不过更简单快速的做法就是切片替换操作，采用隐式或显式的正向迭代遍历。以下是三种变体写法：

```
mylist[:] = filter(keep_function, mylist)
mylist[:] = (x for x in mylist if keep_condition)
mylist[:] = [x for x in mylist if keep_condition]
```

列表推导式可能是最快的。

如何在 Python 中创建数组?

用列表：

```
["this", 1, "is", "an", "array"]
```

列表在时间复杂度方面相当于 C 或 Pascal 的数组；主要区别在于，Python 列表可以包含多种不同类型的对象。

`array` 模块也提供了一些创建具有紧凑表示形式的固定类型数据的方法，但其索引速度要比列表慢。 还可关注 NumPy 和其他一些第三方包也定义了一些各具特色的数组类结构体。

要获得 Lisp 风格的列表，可以使用元组来模拟 *cons 单元*:

```
lisp_list = ("like",  ("this",  ("example", None) ) )
```

如果需要可变特性，你可以用列表来代替元组。 在这里模拟 Lisp *car* 的是 `lisp_list[0]` 而模拟 *cdr* 的是 `lisp_list[1]`。 只有在你确定真有需要时才这样做，因为这通常会比使用 Python 列表要慢上许多。

如何创建多维列表?

多维数组或许会用以下方式建立：

```
>>> A = [[None] * 2] * 3
```

打印出来貌似没错：

```
>>> A
[[None, None], [None, None], [None, None]]
```

但如果给某一项赋值，结果会同时在多个位置体现出来：

```
>>> A[0][0] = 5
>>> A
[[5, None], [5, None], [5, None]]
```

原因在于用 `*` 对列表执行重复操作并不会创建副本，而只是创建现有对象的引用。 `*3` 创建的是包含 3 个引用的列表，每个引用指向的是同一个长度为 2 的列表。1 处改动会体现在所有地方，这一定不是应有的方案。

推荐做法是先创建一个所需长度的列表，然后将每个元素都填充为一个新建列表。

```
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

以上生成了一个包含 3 个列表的列表，每个子列表的长度为 2。也可以采用列表推导式：

```
w, h = 2, 3
A = [[None] * w for i in range(h)]
```

或者，你也可以使用提供矩阵数据类型的扩展；其中最著名的是 NumPy。

我如何将一个方法或函数应用于由对象组成的序列？

要调用一个方法或函数并将返回值累积到一个列表中，list comprehension 是一种优雅的解决方案：

```
result = [obj.method() for obj in mylist]

result = [function(obj) for obj in mylist]
```

如果只需运行方法或函数而不保存返回值，那么一个简单的 `for` 循环就足够了：

```
for obj in mylist:
    obj.method()

for obj in mylist:
    function(obj)
```

为什么 a_tuple[i] += ['item'] 会引发异常？

这是由两个因素共同导致的，一是增强赋值运算符属于 *赋值* 运算符，二是 Python 可变和不可变对象之间的差别。

只要元组的元素指向可变对象，这时对元素进行增强赋值，那么这里介绍的内容都是适用的。在此只以 `list` 和 `+=` 举例。

如果你写成这样:

```
>>> a_tuple = (1, 2)
>>> a_tuple[0] += 1
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

触发异常的原因显而易见： `1` 会与指向 `(1)` 的对象 `a_tuple[0]` 相加，生成结果对象 `2`，但在试图将运算结果 `2` 赋值给元组的 `0` 号元素时就会报错，因为元组元素的指向无法更改。

其实在幕后，上述增强赋值语句的执行过程大致如下：

```
>>> result = a_tuple[0] + 1
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

由于元组是不可变的，因此赋值这步会引发错误。

如果写成以下这样：

```
>>> a_tuple = (['foo'], 'bar')
>>> a_tuple[0] += ['item']
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

这时触发异常会令人略感惊讶，更让人吃惊的是虽有报错，但加法操作却生效了：

```
>>> a_tuple[0]
['foo', 'item']
```

要明白为什么会这样，你需要知道 (a) 如果一个对象实现了 `__iadd__()` 魔术方法，那么它就会在执行 `+=` 增强赋值时被调用，并且其返回值将在赋值语句中被使用；(b) 对于列表而言，`__iadd__()` 等价于在列表上调用 `extend()` 并返回该列表。 所以对于列表我们可以这样说，`+=` 就是 `list.extend()` 的“快捷方式”:

```
>>> a_list = []
>>> a_list += [1]
>>> a_list
[1]
```

这相当于：

```
>>> result = a_list.__iadd__([1])
>>> a_list = result
```

a_list 所引用的对象已被修改，而引用被修改对象的指针又重新被赋值给 `a_list`。 赋值的最终结果没有变化，因为它是引用 `a_list` 之前所引用的同一对象的指针，但仍然发生了赋值操作。

因此，在此元组示例中，发生的事情等同于：

```
>>> result = a_tuple[0].__iadd__(['item'])
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

`__iadd__()` 执行成功，因此列表得到了扩充，但是即使 `result` 是指向 `a_tuple[0]` 所指向的同一个对象，最后的赋值仍然会导致错误，因为元组是不可变的。

我想做一个复杂的排序：能用 Python 进行施瓦茨变换吗？

归功于 Perl 社区的 Randal Schwartz，该技术根据度量值对列表进行排序，该度量值将每个元素映射为“顺序值”。在 Python 中，请利用 `list.sort()` 方法的 `key` 参数：

```
Isorted = L[:]
Isorted.sort(key=lambda s: int(s[10:15]))
```

如何根据另一个列表的值对某列表进行排序？

将它们合并到元组的迭代器中，对结果列表进行排序，然后选择所需的元素。

```
>>> list1 = ["what", "I'm", "sorting", "by"]
>>> list2 = ["something", "else", "to", "sort"]
>>> pairs = zip(list1, list2)
>>> pairs = sorted(pairs)
>>> pairs
[("I'm", 'else'), ('by', 'sort'), ('sorting', 'to'), ('what', 'something')]
>>> result = [x[1] for x in pairs]
>>> result
['else', 'sort', 'to', 'something']
```

对象

什么是类？

类是通过执行 class 语句创建的某种对象的类型。创建实例对象时，用 Class 对象作为模板，实例对象既包含了数据（属性），又包含了数据类型特有的代码（方法）。

类可以基于一个或多个其他类（称之为基类）进行创建。基类的属性和方法都得以继承。这样对象模型就可以通过继承不断地进行细化。比如通用的 `Mailbox` 类提供了邮箱的基本访问方法.，它的子类 `MboxMailbox`、 `MaildirMailbox`、 `OutlookMailbox` 则能够处理各种特定的邮箱格式。

什么是方法？

方法是属于对象的函数，对于对象 `x` ，通常以 `x.name(arguments...)` 的形式调用。方法以函数的形式给出定义，位于类的定义内：

```
class C:
    def meth(self, arg):
        return arg * 2 + self.attribute
```

什么是 self？

Self 只是方法的第一个参数的习惯性名称。假定某个类中有个方法定义为 `meth(self, a, b, c)`，则其实例 `x` 应以 `x.meth(a, b, c)` 的形式进行调用；而被调用的方法则应视其为做了 `meth(x, a, b, c)` 形式的调用。

另请参阅 为什么必须在方法定义和调用中显式使用“self”？ 。

如何检查对象是否为给定类或其子类的一个实例？

使用内置函数 `isinstance(obj, cls)`。 你可以检测对象是否属于多个类中的某一个的实例，只要提供一个元组而非单个类即可，如 `isinstance(obj, (class1, class2, ...))`，还可以检测对象是否属于 Python 的某个内置类型，如 `isinstance(obj, str)` 或 `isinstance(obj, (int, float, complex))`。

请注意 `isinstance()` 还会检测派生自 abstract base class 的虚继承。 因此对于已注册的类，即便没有直接或间接继承自抽象基类，对抽象基类的检测都将返回 `True` 。要想检测“真正的继承”，请扫描类的 MRO:

```
from collections.abc import Mapping

class P:
     pass

class C(P):
    pass

Mapping.register(P)
```

```
>>> c = C()
>>> isinstance(c, C)        # 直接
True
>>> isinstance(c, P)        # 间接
True
>>> isinstance(c, Mapping)  # 虚拟
True

# 实际的继承链
>>> type(c).__mro__
(<class 'C'>, <class 'P'>, <class 'object'>)

# 测试“真正的继承”
>>> Mapping in type(c).__mro__
False
```

请注意，大多数程序不会经常用 `isinstance()` 对用户自定义类进行检测。 如果是自己开发的类，更合适的面向对象编程风格应该是在类中定义多种方法，以封装特定的行为，而不是检查对象属于什么类再据此干不同的事。假定有如下执行某些操作的函数:

```
def search(obj):
    if isinstance(obj, Mailbox):
        ...  # 搜索邮箱的代码
    elif isinstance(obj, Document):
        ...  # 搜索文档的代码
    elif ...
```

更好的方法是在所有类上定义一个 `search()` 方法，然后调用它：

```
class Mailbox:
    def search(self):
        ...  # 搜索邮箱的代码

class Document:
    def search(self):
        ...  # 搜索文档的代码

obj.search()
```

什么是委托?

委托是一种面向对象的技术（也称为设计模式）。假设对象 `x` 已经存在，现在想要改变其某个方法的行为。可以创建一个新类，其中提供了所需修改方法的新实现，而将所有其他方法都委托给 `x` 的对应方法。

Python 程序员可以轻松实现委托。比如以下实现了一个类似于文件的类，只是会把所有写入的数据转换为大写：

```
class UpperOut:

    def __init__(self, outfile):
        self._outfile = outfile

    def write(self, s):
        self._outfile.write(s.upper())

    def __getattr__(self, name):
        return getattr(self._outfile, name)
```

这里 `UpperOut` 类重新定义了 `write()` 方法，在调用下层的 `self._outfile.write()` 方法之前将参数字符串转换为大写形式。 所有其他方法都被委托给下层的 `self._outfile` 对象。 委托是通过 `__getattr__()` 方法完成的；请参阅 语言参考 了解有关控制属性访问的更多信息。

请注意在更一般的情况下委托可能会变得比较棘手。 当属性即需要被设置又需要被提取时，类还必须定义 `__setattr__()` 方法，而这样做必须十分小心。 `__setattr__()` 的基本实现大致如下所示:

```
class X:
    ...
    def __setattr__(self, name, value):
        self.__dict__[name] = value
    ...
```

许多 __setattr__() 实现都会调用 object.__setattr__() 在 self 上设置属性，而不会导致无限递归:

```
class X:
    def __setattr__(self, name, value):
        # 这里添加自定义的逻辑...
        object.__setattr__(self, name, value)
```

另外，也可以通过直接在 self.__dict__ 中插入条目来设置属性。

如何在扩展基类的派生类中调用基类中定义的方法?

使用内置的 super() 函数:

```
class Derived(Base):
    def meth(self):
        super().meth()  # 调用 Base.meth
```

在下面的例子中，super() 将自动根据它的调用方 (self 值) 来确定实例对象，使用 type(self).__mro__ 查找 method resolution order (MRO)，并返回 MRO 中位于 Derived 之后的项: Base。

如何让代码更容易对基类进行修改?

可以为基类赋一个别名并基于该别名进行派生。这样只要修改赋给该别名的值即可。顺便提一下，如要动态地确定（例如根据可用的资源）该使用哪个基类，这个技巧也非常方便。例如:

```
class Base:
    ...

BaseAlias = Base

class Derived(BaseAlias):
    ...
```

如何创建静态类数据和静态类方法?

Python 支持静态数据和静态方法（以 C++ 或 Java 的定义而言）。

静态数据只需定义一个类属性即可。若要为属性赋新值，则必须在赋值时显式使用类名:

```
class C:
    count = 0   # C.__init__ 被调用的次数

    def __init__(self):
        C.count = C.count + 1

    def getcount(self):
        return C.count  # 或返回 self.count
```

对于所有符合 isinstance(c, C) 的 c， c.count 也同样指向 C.count，除非被 c 自身或者被从 c.__class__ 回溯到基类 C 的搜索路径上的某个类所覆盖。

注意：在 C 的某个方法内部，像 `self.count = 42` 这样的赋值将在 `self` 自身的字典中新建一个名为 "count" 的不相关实例。 想要重新绑定类静态数据名称就必须总是指明类名，无论是在方法内部还是外部:

```
C.count = 314
```

Python 支持静态方法：

```
class C:
    @staticmethod
    def static(arg1, arg2, arg3):
        # 没有 'self' 形参!
        ...
```

不过为了获得静态方法的效果，还有一种做法直接得多，也即使用模块级函数即可：

```
def getcount():
    return C.count
```

如果代码的结构化比较充分，每个模块只定义了一个类（或者多个类的层次关系密切相关），那就具备了应有的封装。

在 Python 中如何重载构造函数（或方法）？

这个答案实际上适用于所有方法，但问题通常首先出现于构造函数的应用场景中。

在 C++ 中，代码会如下所示：

```
class C {
    C() { cout << "No arguments\n"; }
    C(int i) { cout << "Argument is " << i << "\n"; }
}
```

在 Python 中，只能编写一个构造函数，并用默认参数捕获所有情况。例如：

```
class C:
    def __init__(self, i=None):
        if i is None:
            print("No arguments")
        else:
            print("Argument is", i)
```

这不完全等同，但在实践中足够接近。

也可以试试采用变长参数列表，例如：

```
def __init__(self, *args):
    ...
```

上述做法同样适用于所有方法定义。

在用 __spam 的时候得到一个类似 _SomeClassName__spam 的错误信息。

以双下划线打头的变量名会被“破坏”，以便以一种简单高效的方式定义类私有变量。任何形式为 `__spam` 的标识符（至少前缀两个下划线，至多后缀一个下划线）文本均会被替换为 `_classname__spam`，其中 `classname` 为去除了全部前缀下划线的当前类名称。

标识符可以在类的内部不加改变地使用，但要在类的外部访问它，就必须使用被混淆的名称：

```
class A:
    def __one(self):
        return 1
    def two(self):
        return 2 * self.__one()

class B(A):
    def three(self):
        return 3 * self._A__one()

four = 4 * A()._A__one()
```

需要特别指出，这并不能保证私密性因为外部用户仍然可以有意地访问私有属性；许多 Python 程序员根本就不屑于使用私有变量名。

参见： 私有名称调整规范说明 了解相关详情和特例。

类定义了 __del__ 方法，但是删除对象时没有调用它。

这有几个可能的原因。

`del` 语句不一定要调用 `__del__()` -- 它只是减少对象的引用计数，如果计数达到零才会调用 `__del__()`。

如果你的数据结构包含循环链接（如树每个子节点都带有父节点的引用，而每个父节点也带有子节点的列表），引用计数永远不会回零。 尽管 Python 偶尔会用某种算法检测这种循环引用，但在数据结构的最后一条引用消失之后，垃圾收集器可能还要过段时间才会运行，因此 `__del__()` 方法可能会在不方便或随机的时刻被调用。 这对于重现一个问题是非常不方便的。 更糟糕的是，各个对象的 `__del__()` 方法是以随机顺序执行的。 虽然你可以运行 `gc.collect()` 来强制执行垃圾回收操作，但 *仍会存在* 一些对象永远不会被回收的失控情况。

尽管有垃圾回收器，但当对象使用完毕时在要调用的对象上定义显式的 `close()` 方法仍然是个好主意。 `close()` 方法可以随后移除引用子对象的属性。 请不要直接调用 `__del__()` -- `__del__()` 应当调用 `close()` 并且 `close()` 应当确保被可以被同一对象多次调用。

另一种避免循环引用的做法是利用 `weakref` 模块，该模块允许指向对象但不增加其引用计数。例如，树状数据结构应该对父节点和同级节点使用弱引用（如果真要用的话！）

最后，如果你的 `__del__()` 方法引发了异常，会将警告消息打印到 `sys.stderr`。

如何获取给定类的所有实例的列表？

Python 不会记录类（或内置类型）的实例。可以在类的构造函数中编写代码，通过保留每个实例的弱引用列表来跟踪所有实例。

为什么 id() 的结果看起来不是唯一的?

id() 返回一个整数，该整数在对象的生命周期内保证是唯一的。 因为在 CPython 中，这是对象的内存地址，所以经常发生在从内存中删除对象之后，下一个新创建的对象被分配在内存中的相同位置。 这个例子说明了这一点：

```
>>> id(1000)
13901272
>>> id(2000)
13901272
```

这两个 id 属于不同的整数对象，之前先创建了对象，执行 `id()` 调用后又立即被删除了。若要确保检测 id 时的对象仍处于活动状态，请再创建一个对该对象的引用：

```
>>> a = 1000; b = 2000
>>> id(a)
13901272
>>> id(b)
13891296
```

什么情况下可以依靠 *is* 运算符进行对象的身份相等性测试?

`is` 运算符可用于测试对象的身份相等性。`a is b` 等价于 `id(a) == id(b)`。

身份相等性最重要的特性就是对象总是等同于自身，`a is a` 一定返回 `True`。身份相等性测试的速度通常比相等性测试要快。而且与相等性测试不一样，身份相等性测试会确保返回布尔值 `True` 或 `False`。

但是，身份相等性测试 *只能* 在对象身份确定的场景下才可替代相等性测试。一般来说，有以下3种情况对象身份是可以确定的：

1. 赋值操作将创建新的名称但不会改变对象标识号。 在赋值操作 `new = old` 之后，可以保证 `new is old`。
2. 将对象放入存储对象引用的容器不会改变对象的标识号。 在列表赋值操作 `s[0] = x` 之后，将可保证 `s[0] is x`。
3. 如果一个对象是单例，则意味着该对象只能存在一个实例。 在赋值操作 `a = None` 和 `b = None` 之后，可以保证 `a is b` 因为 `None` 是单例对象。

其他大多数情况下，都不建议使用身份相等性测试，而应采用相等性测试。尤其是不应将身份相等性测试用于检测常量值，例如 `int` 和 `str`，因为他们并不一定是单例对象：

```
>>> a = 1000
>>> b = 500
>>> c = b + 500
>>> a is c
False
```

```
>>> a = 'Python'
>>> b = 'Py'
>>> c = b + 'thon'
>>> a is c
False
```

同样地，可变容器的新实例，对象身份一定不同：

```
>>> a = []
>>> b = []
>>> a is b
False
```

在标准库代码中，给出了一些正确使用对象身份测试的常见模式：

1. 正如 **PEP 8** 所建议的，标识测试是检查 `None` 的推荐方式。 这样的代码读起来就像直白的英语并可避免与具有结果为假的布尔值的对象相混淆。

2. 当 `None` 是一个有效的输入值时检查可选参数会有点麻烦。 在这些情况下，你可以创建一个保证与其他对象不同的单例哨兵对象。 例如，以下代码演示了如何实现一个行为与 `dict.pop()` 类似的方法：

   ```
   _sentinel = object()

   def pop(self, key, default=_sentinel):
       if key in self:
           value = self[key]
           del self[key]
           return value
       if default is _sentinel:
           raise KeyError(key)
       return default
   ```

3. 容器的实现有时需要用标识测试来增强相等性测试。 这样可以防止代码被 `float('NaN')` 这类不等于自身的对象所干扰。

例如，以下是 `collections.abc.Sequence.__contains__()` 的实现代码：

```
def __contains__(self, value):
    for v in self:
        if v is value or v == value:
            return True
    return False
```

一个子类如何控制哪些数据被存储在一个不可变的实例中？

当子类化一个不可变类型时，请重写 `__new__()` 方法而不是 `__init__()` 方法。 后者只在一个实例被创建 *之后* 运行，这对于改变不可变实例中的数据来说太晚了。

所有这些不可变的类都有一个与它们的父类不同的签名：

```
from datetime import date
```

```
class FirstOfMonthDate(date):
    "Always choose the first day of the month"
    def __new__(cls, year, month, day):
        return super().__new__(cls, year, month, 1)

class NamedInt(int):
    "Allow text names for some numbers"
    xlat = {'zero': 0, 'one': 1, 'ten': 10}
    def __new__(cls, value):
        value = cls.xlat.get(value, value)
        return super().__new__(cls, value)

class TitleStr(str):
    "Convert str to name suitable for a URL path"
    def __new__(cls, s):
        s = s.lower().replace(' ', '-')
        s = ''.join([c for c in s if c.isalnum() or c == '-'])
        return super().__new__(cls, s)
```

这些类可以这样使用:

```
>>> FirstOfMonthDate(2012, 2, 14)
FirstOfMonthDate(2012, 2, 1)
>>> NamedInt('ten')
10
>>> NamedInt(20)
20
>>> TitleStr('Blog: Why Python Rocks')
'blog-why-python-rocks'
```

我该如何缓存方法调用?

缓存方法的两个主要工具是 `functools.cached_property()` 和 `functools.lru_cache()`。 前者在实例层级上存储结果而后者在类层级上存储结果。

cached_property 方式仅适用于不接受任何参数的方法。 它不会创建对实例的引用。 被缓存的方法结果将仅在实例的生存期内被保留。

其优点是，当一个实例不再被使用时，缓存的方法结果将被立即释放。缺点是，如果实例累积起来，累积的方法结果也会增加。它们可以无限制地增长。

lru_cache 方式适用于具有 hashable 参数的方法。 它会创建对实例的引用，除非特别设置了传入弱引用。

最少近期使用算法的优点是缓存会受指定的 *maxsize* 限制。 它的缺点是实例会保持存活，直到其达到生存期或者缓存被清空。

这个例子演示了几种不同的方式:

```
class Weather:
    "Lookup weather information on a government website"

    def __init__(self, station_id):
        self._station_id = station_id
        # The _station_id is private and immutable
```

```
    def current_temperature(self):
        "Latest hourly observation"
        # Do not cache this because old results
        # can be out of date.

    @cached_property
    def location(self):
        "Return the longitude/latitude coordinates of the station"
        # Result only depends on the station_id

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='mm'):
        "Rainfall on a given date"
        # 取决于 station_id、date 和 unit
```

上面的例子假定 *station_id* 从不改变。 如果相关实例属性是可变对象，则 *cached_property* 方式就不再适用，因为它无法检测到属性的改变。

要让 *lru_cache* 方式在 *station_id* 可变时仍然适用，类需要定义 `__eq__()` 和 `__hash__()` 方法以便缓存能检测到相关属性的更新:

```
class Weather:
    "Example with a mutable station identifier"

    def __init__(self, station_id):
        self.station_id = station_id

    def change_station(self, station_id):
        self.station_id = station_id

    def __eq__(self, other):
        return self.station_id == other.station_id

    def __hash__(self):
        return hash(self.station_id)

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='cm'):
        'Rainfall on a given date'
        # 取决于 station_id、date 和 unit
```

模块

如何创建 .pyc 文件?

当首次导入模块时（或当前已编译文件创建之后源文件发生了改动），在 `.py` 文件所在目录的 `__pycache__` 子目录下会创建一个包含已编译代码的 `.pyc` 文件。该 `.pyc` 文件的名称开头部分将与 `.py` 文件名相同，并以 `.pyc` 为后缀，中间部分则依据创建它的 `python` 版本而各不相同。（详见 **PEP 3147**。）

`.pyc` 文件有可能会无法创建，原因之一是源码文件所在的目录存在权限问题，这样就无法创建 `__pycache__` 子目录。假如以某个用户开发程序而以另一用户运行程序，就有可能发生权限问题，测试 Web 服务器就属于这种情况。

除非设置了 PYTHONDONTWRITEBYTECODE 环境变量，否则导入模块并且 Python 能够创建 __pycache__ 子目录并把已编译模块写入该子目录（权限、存储空间等等）时，.pyc 文件就将自动创建。

在最高层级运行的 Python 脚本不会被视为经过了导入操作，因此不会创建 .pyc 文件。假定有一个最高层级的模块文件 foo.py，它导入了另一个模块 xyz.py，当运行 foo 模块（通过输入 shell 命令 python foo.py ），则会为 xyz 创建一个 .pyc，因为 xyz 是被导入的，但不会为 foo 创建 .pyc 文件，因为 foo.py 不是被导入的。

若要为 foo 创建 .pyc 文件 —— 即为未做导入的模块创建 .pyc 文件 —— 可以利用 py_compile 和 compileall 模块。

py_compile 模块能够手动编译任意模块。 一种做法是交互式地使用该模块中的 compile() 函数:

```
>>> import py_compile
>>> py_compile.compile('foo.py')
```

这将会将 .pyc 文件写入与 foo.py 相同位置下的 __pycache__ 子目录（或者你也可以通过可选参数 cfile 来重写该行为）。

还可以用 compileall 模块自动编译一个或多个目录下的所有文件。只要在命令行提示符中运行 compileall.py 并给出要编译的 Python 文件所在目录路径即可：

```
python -m compileall .
```

如何找到当前模块名称?

模块可以查看预定义的全局变量 __name__ 获悉自己的名称。如其值为 '__main__' ，程序将作为脚本运行。通常，许多通过导入使用的模块同时也提供命令行接口或自检代码，这些代码只在检测到处于 __name__ 之后才会执行：

```
def main():
    print('Running test...')
    ...

if __name__ == '__main__':
    main()
```

如何让模块相互导入?

假设有以下模块：

foo.py:

```
from bar import bar_var
foo_var = 1
```

bar.py:

```
from foo import foo_var
bar_var = 2
```

问题是解释器将执行以下步骤：

- 首先导入 `foo`
- 为 `foo` 创建空的全局变量
- 编译 `foo` 并开始执行
- `foo` 导入 `bar`
- 为 `bar` 创建空的全局变量
- 编译 `bar` 并开始执行
- `bar` 导入 `foo` (该步骤无操作，因为已经有一个名为 `foo` 的模块)。
- 导入机制尝试从 `foo_var` 全局变量读取 `foo`，用来设置 `bar.foo_var = foo.foo_var`

最后一步失败了，因为 Python 还没有完成对 foo 的解释，foo 的全局符号字典仍然是空的。

当你使用 `import foo` ，然后尝试在全局代码中访问 `foo.foo_var` 时，会发生同样的事情。

这个问题有（至少）三种可能的解决方法。

Guido van Rossum 建议完全避免使用 `from <module> import ...` ，并将所有代码放在函数中。全局变量和类变量的初始化只应使用常量或内置函数。这意味着导入模块中的所有内容都以 `<module>.<name>` 的形式引用。

Jim Roskind 建议每个模块都应遵循以下顺序：

- 导出（全局变量、函数和不需要导入基类的类）
- `import` 语句
- 本模块的功能代码（包括根据导入值进行初始化的全局变量）。

Van Rossum 不太喜欢这种方法，因为import出现在一个奇怪的地方，但它确实有效。

Matthias Urlichs 建议对代码进行重构，使得递归导入根本就没必要发生。

这些解决方案并不相互排斥。

__import__('x.y.z') 返回的是 <module 'x'>；该如何得到 z 呢？

不妨考虑换用 `importlib` 中的函数 `import_module()` ：

```
z = importlib.import_module('x.y.z')
```

对已导入的模块进行了编辑并重新导入，但变动没有得以体现。这是为什么？

出于效率和一致性的原因，Python 仅在第一次导入模块时读取模块文件。否则，在一个多模块的程序中，每个模块都会导入相同的基础模块，那么基础模块将会被一而再、再而三地解析。如果要强行重新读取已更改的模块，请执行以下操作：

```
import importlib
import modname
importlib.reload(modname)
```

警告：这种技术并非万无一失。尤其是模块包含了以下语句时：

```
from modname import some_objects
```

仍将继续使用前一版的导入对象。如果模块包含了类的定义，并 *不会* 用新的类定义更新现有的类实例。这样可能会导致以下矛盾的行为：

```
>>> import importlib
>>> import cls
>>> c = cls.C()                # Create an instance of C
>>> importlib.reload(cls)
<module 'cls' from 'cls.py'>
>>> isinstance(c, cls.C)       # isinstance is false?!?
False
```

只要把类对象的 id 打印出来，问题的性质就会一目了然：

```
>>> hex(id(c.__class__))
'0x7352a0'
>>> hex(id(cls.C))
'0x4198d0'
```