编程 FAQ

Date: $Date: 2005-06-05 19:37:51 -0500 (Sun, 05 Jun 2005) $ Version: $Revision: 8226 $ Web site: http://www.python.org/

1.1. 一般问题

1.1.1. 是否有源码级的调试器， 具有breakpoint, single-stepping等功能?

是的。

pdb模块是一个简单却强大的命令行模式的python调试器。它是标准python库的一部分, 在库参考手册中有关于它的文档。作为一个例 子，你也可以使用pdb的代码编写你自己的调试器。

作为标准python发行包的一部分（通常为Tools/scripts/idle），IDLE包含了一个图形界面的调试器。在http://www.python.org/idle/doc/idle2.html#Debugger有IDLE调试器的文档。

另一个Python IDE， PythonWin包含了一个基于pdb的GUI调试器。 Pythonwin调试器对breakpoints作颜色标记，它还有一些很酷的特性，比如调试非python程序。可以参考http://www.python.org/windows/pythonwin/。最 新版本的PythonWin已作为ActivePython 发行包的一部分(见 http://www.activestate.com/Products/ActivePython/index.html)。

Boa Constructor 是一个使用wxPython的IDE和GUI builder。它提供了可视化的框架创建和操作，一个对象探查器，多种代码视图比如对象浏览器，继承架构，doc string创建的html文档，一个高级调试器，继承帮助和Zope支持。

Eric3 是基于PyQt和Scintilla editing组件的一个IDE。

Pydb是python标准调试器pdb的一个版本, 与DDD(Data Display Debugger, 一个流行的调试器图形界面)一起工作。Pydb 可以在http://packages.debian.org/unstable/devel/pydb.html找到，DDD可以在 http://www.gnu.org/software/ddd找到.

还有很多包含图形界面的商业版本Python IDE。包括:

Wing IDE (http://wingide.com)

Komodo IDE (http://www.activestate.com/Products/Komodo)

1.1.2. 是否有工具可以帮助找到bug或者做静态分析?

是的.

PyChecker是一个静态分析器，它可以找出python代码中的bug并对代码的复杂性和风格作出警告。可以在 http://pychecker.sf.net找到它.

另一个工具Pylint 检查一个模块是否满足编码规范，它还支持插件扩展。除了PyChecker能提供的bug检查外，Pylint 还提供额外的特性比如检查代码行长度，变量名是否符合代码规范，声明的接口是否都已实现等。http://www.logilab.org/projects/pylint/documentation 提供了关于Pylint特性的一个完整列表。

1.1.3. 如何由python脚本创建一个单独的二进制文件?

如果你只是希望用户运行一个单独的程序而不需要预先下载一个python的发行版，则并不需要将Python代码编译成C代码。有很多工具可以找出程序依赖的模块并将这些模块 与程序绑定在一起以产生一个单独的执行文件。

其中一种工具就是freeze tool, 它作为Tools/freeze被包含在python的代码树中。它将python字节码转换成C数组，和一个可将你所有模块嵌入到新程序中的编译器，这个编译器跟python模块链接在一起。

它根据import语句递归地扫描源代码，并查找在标准python路径中的模块和源代码目录中的模块（内建模块）。用python写的模块的字节码随后被转换成C代码（可以通过使用marshal模块转换成代码对象的数组构 造器），并产生一个可自定义的配置文件，只包含程序使用了的模块。 最后将生成的C代码编译并链接至余下的的python解释器，产生一个与你的script执行效果完全一样的单独文件。

显然，freeze需要一个C编译器。但也有一些工具并不需要。首先便是Gordon McMillan's installer，它在

http://www.mcmillan-inc.com/install1.html

它工作在Windows, Linux和至少是部分Unix变种上。

另一个便是Thomas Heller的 py2exe (只适用于Windows平台)，它在

http://starship.python.net/crew/theller/py2exe

第三个是Christian Tismer的 SQFREEZE，它将字节码附在一个特殊的python解释器后面，解释器负责找到这段代码。Python 2.4可能会引入类似的机制。

其它工具包括Fredrik Lundh的 Squeeze 和 Anthony Tuininga的 cx_Freeze.

1.1.4. 是否有关于python程序的代码标准或风格向导?

是的。标准库模块要求的代码风格被列在PEP 8.

1.1.5. 程序执行速度太慢，如何改善?

一般来说这是个复杂的问题。有很多技巧可以提升python的速度，比如可以用C重写部分代码。

在某些情况下将python转换成C或x86汇编语言是可能的，这意味着您不需要修改代码就可获得速度提升。

Pyrex 可以将稍许改动过的python码转换成C扩展，并可以在很多平台上使用。

Psyco 是一个即时编译器，可将python码转换成x86汇编语言。如果你可以使用它， Psyco 可使关键函数有明显的性能提升。

剩下的问题就是讨论各种可稍许提升python代码速度的技巧。在profile指出某个函数是一个经常执行的热点后，除非确实需要，否则不要应用任何优化措施，优化经常会使代码变得不清晰，您不应该承受这样做所带来的负担（延长的开发时间，更多可能的bug），除非优化结果确实值得你这样做。

Skip Montanaro 有一个专门关于提升python代码速度的网页，位于

http://manatee.mojam.com/~skip/python/fastpython.html。

Guido van Rossum 写了关于提升python代码速度的内容，在http://www.python.org/doc/essays/list2str.html。

还有件需要注意的事，那就是函数特别是方法的调用代价相当大；如果你设计了一个有很多小型函数的纯面向对象的接口，而这些函数所做的不过是对实例变量获取或赋值，又或是调用另一个方法，那么你应该考虑使用更直接的方式比如直接存取实例变量。也可参照profile模块（在Library Reference manual中描述)，它 能找出程序哪些部分耗费多数时间（如果你有耐性的话--profile本身会使程序数量级地变慢）。

记住很多从其它语言中学到的标准优化方法也可用于python编程。比如，在执行输出时通过使用更大块的写入来减少系统调用会加快程序速度。因此CGI脚本一次性的写入所有输出就会比写入很多次小块输出快得多。

同样的，在适当的情况下使用python的核心特性。比如，通过使用高度优化的C实现，slicing允许程序在解释器主循环的一个滴答中，切割list和其它sequence对象。因此 ，为取得同样效果，为取得以下代码的效果

```
切换行号显示 lang=en id=CA-126f809fe86276c5447b73ce84afb2788500f3b1_000 dir=ltr 1
    2 L2 = []
    3 for i in range[3]:
    4         L2.append(L1[i])
```

使用

```
切换行号显示 lang=en id=CA-824510e92283a84c8d5e068caf911c30204b48f1_001 dir=ltr 1
    2 L2 = list(L1[:3]) # "list" is redundant if L1 is a list.
```

则更短且快得多。

注意，内建函数如map(), zip(), 和friends在执行一个单独循环的任务时，可被作为一个方便的加速器。比如将两个list配成一对：

```
切换行号显示 lang=en id=CA-3e9db635d58eaf9ae0af25f8f7f9db52479e4c5c_002 dir=ltr 1
    2 >>> zip([1,2,3], [4,5,6])
    3 [(1, 4), (2, 5), (3, 6)]
```

或在执行一系列正弦值时：

```
切换行号显示 lang=en id=CA-a27db439ee995c1f0113d15a62ac4d4472f1424b_003 dir=ltr 1
    2 >>> map(math.sin, (1,2,3,4))
    3 [0.841470984808, 0.909297426826, 0.14112000806,    -0.756802495308]
```

在这些情况下，操作速度会很快。

其它的例子包括string对象的join()和split()方法。例如，如果s1..s7 是大字符串(10K+)那么join([s1,s2,s3,s4,s5,s6,s7])就会比s1+s2+s3+s4+s5+s6+s7快得多, 因为后者会计算很多次子表达式，而join()则在一次过程中完成所有的复制。对于字符串操作，对字符串对象使用replace()方

法。仅当在没有固定字符串模式时才使用正则表达式。考虑使用字符串格式化操作string % tuple和string % dictionary。

使用内建方法list.sort()来排序，参考sorting mini-HOWTO中关于较高级的使用例子。除非在极特殊的情况下，list.sort()比其它任何 方式都要好。

另一个技巧就是"将循环放入函数或方法中" 。例如，假设你有个运行的很慢的程序，而且你使用profiler确定函数ff()占用了很多时间。如果你注意到ff()：

```
def ff(x):
  ..do something with x computing result... return result
```

常常是在循环中被调用，如:

切换行号显示 lang=en id=CA-68f54bad29340cfa485397501054f86a258b3c34_004 dir=ltr

```
1
2 list = map(ff, oldlist)
```

或:

切换行号显示 lang=en id=CA-fa4513b5c2fd16aaf67b924b09ab66e0fa1bbbec_005 dir=ltr

```
1
2 for x in sequence:
3     value = ff(x)
4     ...do something with value...
```

那么你可以通过重写ff()来消除函数的调用开销：

切换行号显示 lang=en id=CA-81a20abe975855f0577931c9d4777182945ff0fe_006 dir=ltr

```
1
2 def ffseq(seq):
3     resultseq = []
4     for x in seq:
5         ...do something with x computing result...
6         resultseq.append(result)
7     return resultseq
```

并重写以上两个例子：

```
list = ffseq(oldlist)
```

和

切换行号显示 lang=en id=CA-ffc57530e71f5adc90e177945e40a9b3ecc21b31_007 dir=ltr

```
1
2 for value in ffseq(sequence):
3     ...do something with value...
```

单独对ff(x)的调用被翻译成ffseq([x])[0]，几乎没有额外开销。当然这个技术并不总是合适的，还是其它的方法。

你可以通过将函数或方法的定位结果精确地存储至一个本地变量来获得一些性能提升。一个循环如：

切换行号显示 lang=en id=CA-8eb65d40dc3111930ebb17153016d8917d0cec7a_008 dir=ltr 1

```
2 for key in token:
3         dict[key] = dict.get(key, 0) + 1
```

每次循环都要定位dict.get。如果这个方法一直不变，可这样实现以获取小小的性能提升：

切换行号显示 lang=en id=CA-9fc74cfaaa7b4395017290565af4af825c2ec127_009 dir=ltr 1

```
2 dict_get = dict.get    # look up the method once
3 for key in token:
4         dict[key] = dict_get(key, 0) + 1
```

默认参数可在编译期被一次赋值，而不是在运行期。这只适用于函数或对象在程序执行期间不被改变的情况，比如替换

切换行号显示 lang=en id=CA-aff07d7a82116bcd5721b3ea4c43b3ff3aec57f7_010 dir=ltr 1

```
2 def degree_sin(deg):
3         return math.sin(deg * math.pi / 180.0)
```

为

切换行号显示 lang=en id=CA-5e3d316c887b90c025105c99e9d3521206270e0c_011 dir=ltr 1

```
2 def degree_sin(deg, factor = math.pi/180.0, sin = math.sin):
3         return sin(deg * factor)
```

因为这个技巧对常量变量使用了默认参数，因而需要保证传递给用户API时不会产生混乱。

1.2. 核心语言

1.2.1. 如何在一个函数中设置一个全局变量?

你是否做过类似的事？

切换行号显示 lang=en id=CA-67975ac570623d6e2c7b611f8037a8a7b8cfa39c_012 dir=ltr 1

```
2 x = 1 # make a global
3
4 def f():
5             print x # try to print the global
6             ...
7             for j in range(100):
8                         if q>3:
9                                     x=4
```

任何函数内赋值的变量都是这个函数的local变量。除非它专门声明为global。作为函数体最后一个语句，x被赋值，因此编译器认为x为local变量。而语句print x 试图 print一个未初始化的local变量，因而会触发NameError 异常。

解决办法是在函数的开头插入一个明确的global声明。

切换行号显示 lang=en id=CA-9ce4e3ca84cc281a0dcedca62c24f46221aca455_013 dir=ltr 1

```
2 def f():
3         global x
4         print x # try to print the global
5         ...
6         for j in range(100):
7                 if q>3:
8                     x=4
```

在这种情况下，所有对x的引用都是模块名称空间中的x。

1.2.2. python中local和global变量的规则是什么?

在Python中, 某个变量在一个函数里只是被引用，则认为这个变量是global。如果函数体中变量在某个地方会被赋值，则认为这个变量是local。如果一个global变量在函数体中 被赋予新值，这个变量就会被认为是local，除非你明确地指明其为global。

尽管有些惊讶，我们略微思考一下就会明白。一方面，对于被赋值的变量，用关键字 global是为了防止意想不到的边界效应。另一方面，如果对所有的global引用都需要关键字global，则会不停地使用global关键字。需要在每次引用内建函数或一个import的模块时都声明global。global声明是用来确定边界效应的，而这 种混乱的用法会抵消这个作用。

1.2.3. 如何在模块间共享global变量?

在一个单独程序中，各模块间共享信息的标准方法是创建一个特殊的模块（常被命名为config和cfg）。仅需要在你程序中每个模块里import这个config模块。 因为每个模块只有一个实例，对这个模块的任何改变将会影响所有的地方。例如:

config.py:

切换行号显示 lang=en id=CA-dcebffde192daf346bcb701d9b8f91e4d0018940_014 dir=ltr 1

```
2 x = 0      # Default value of the 'x' configuration setting
```

mod.py:

切换行号显示 lang=en id=CA-eb5f9da75cd113d75380e369fa0dbfc7b635ecbb_015 dir=ltr 1

```
2 import config
3 config.x = 1
```

main.py:

切换行号显示 lang=en id=CA-cb489ab8dd611ffa7002ccfe6ce49700b9e75081_016 dir=ltr 1

```
2 import config
3 import mod
4 print config.x
```

注意，由于同样的原因，使用模块也是实现Singleton设计模式的基础。

1.2.4. 什么是import模块的最好方式?

通常情况下，不要使用from modulename import * 这种格式。这样做会使引入者的namespace混乱。很多人甚至对于那些专门设计用于这种模式的模块都不采用这种方式。被设计成这种模式的模块包括Tkinter, 和threading.

在一个文件的开头引入模块。这样做使得你的你的代码需要哪些模块变得清晰，并且避免了模块名称是否存在的问题。 在每行只使用一次import使得添加和删除模块import更加容易，但每行多个import则减少屏幕空间的使用。

应该按照以下顺序import模块：

标准库模块 -- 如 sys, os, getopt 等

第三方模块(安装在python的site-packages目录下) -- 如 mx.DateTime, ZODB, PIL.Image, 等。

本地实现的模块。

不要使用相对的import。如果你在编写package.sub.m1 模块的代码并想 import package.sub.m2, 不要只是 import m2, 即使这样是合法的。用 from package.sub import m2 代替. 相对的imports会导致模块被初始化两次，并产生奇怪的bug。

有时需要将import语句移到函数或类中来防止import循环。 Gordon McMillan 说:

> 在两个模块都使用 "import <module>" 格式时是没问题的 。但若第二个模块想要获取第一个模块以外的一个名称("from module import name")且这个import语句位于最顶层时，则会产生错误 。因为这时第一个模块的名称并不处于有效状态，因为第一个模块正忙于import第二个模块。

在这种情况下，如果第二个模块只是用在一个函数中，那么可以简单地把import移入到这个函数中。当这个import被调用时，第一个模块已经完成了初始化，而第二个模块 则可以完成它的import语句了。

如果某些模块是系统相关的，那么将import移出顶层代码也是必要的。在那种情况下，甚至不可能在文件的顶层import所有的模块。在这种情况下，在对应的系统相关代码中引入这些模块则是个好的选择。

在解决诸如防止import循环或试图减少模块初始化时间等问题，且诸多模块并不需要依赖程序是如何执行的情况下，这种方法尤其有用。如果模块只是被用在某个函数中，你也可以将import移到这个函数中。注意首次import模块会花费较多的时间，但多次地import则几乎不会再花去额外的时间，而只是需要两次的字典查询操作。即使模块名称已经处在scope外，这个模块也很有可能 仍处在sys.modules中。

如果只是某个类的实例使用某个模块，则应该在类的__init__ 方法里import模块并把这个模块赋给一个实例变量以使这个模块在对象的整个生命周期内一直有效（通过这个实例变量）。注

意要使import推迟到类的实例化，必须将import放入某个方法中。在类里所有方法之外的地方放置import语句，仍然会 使模块初始化的时候执行import。

1.2.5. 如何将某个函数的选项或键值参数传递到另一个函数?

在函数的参数列表中使用 * 和 ** ；它将你的位置参数作为一个tuple，将键值参数作为一个字典。当调用另一个函数时你可以通过使用 * 和 **来传递这些参数：

切换行号显示 lang=en id=CA-69ce94899ec7e334438b0979e243e8f0d81e8d95_017 dir=ltr 1

```
2 def f(x, *tup, **kwargs):
3         ...
4         kwargs['width']='14.3c'
5         ...
6         g(x, *tup, **kwargs)
```

如果考虑到比python的2.0更老的版本的特殊情况，使用'apply':

切换行号显示 lang=en id=CA-dc716c538abd4053ab25bd030a00c78b491e2df3_018 dir=ltr 1

```
2 def f(x, *tup, **kwargs):
3         ...
4         kwargs['width']='14.3c'
5         ...
6         apply(g, (x,)+tup, kwargs)
```

1.2.6. 如何编写一个带有输出参数的函数（传引用调用）?

记住在python中参数传递是动过赋值实现的。因为赋值仅是创建一个新的对对象的引用，所以在调用者和被调用者之间没有任何的别名可以使用，因此从本质上说没有传引用调用。但你可以通过一系列的方法来实现这个效果。

对结果传递一个tuple:

切换行号显示 lang=en id=CA-8e7619362f26fbd59b4ca6f400f93aac6ee935fd_019 dir=ltr 1

```
2 def func2(a, b):
3     a = 'new-value'         # a and b are local names
4     b = b + 1                 # assigned to new objects
5     return a, b               # return new values
6
7 x, y = 'old-value', 99
8 x, y = func2(x, y)
9 print x, y                    # output: new-value 100
```

这通常是最清晰的方法。

通过使用global变量。这不是线程安全的，所以不推荐。

传递一个可变对象:

切换行号显示 lang=en id=CA-ef0bdf9a7236b062f3d4a1dfe6735a0f5e725c1e_020 dir=ltr 1

```
2 def func1(a):
3         a[0] = 'new-value'         # 'a' references a mutable list
4         a[1] = a[1] + 1            # changes a shared object
5
6 args = ['old-value', 99]
7 func1(args)
8 print args[0], args[1]         # output: new-value 100
```

传递一个可变字典：

切换行号显示 lang=en id=CA-ed932102a0340ff682791dfc70ead8edff72f198_021 dir=ltr 1

```
2 def func3(args):
3         args['a'] = 'new-value'         # args is a mutable dictionary
4         args['b'] = args['b'] + 1      # change it in-place
5
6 args = {'a':' old-value', 'b': 99}
7 func3(args)
8 print args['a'], args['b']
```

或者是将它绑定在一个类的实例中：

切换行号显示 lang=en id=CA-46bfedaa6581d05b181199c3082808cbb1100e5d_022 dir=ltr 1

```
2 class callByRef:
3         def __init__(self, **args):
4                 for (key, value) in args.items():
5                         setattr(self, key, value)
6
7 def func4(args):
8         args.a = 'new-value'              # args is a mutable callByRef
9         args.b = args.b + 1               # change object in-place
10
11 args = callByRef(a='old-value', b=99)
12 func4(args)
13 print args.a, args.b
```

但这样会使程序变得复杂，并不是一个好方法。

最好的方法还是返回一个包含多个结果的tuple。

1.2.7. 如何使用python中更高 阶的函数?

有两个选择：你可以使用内嵌的方式或使用可调用对象。比如，假设你想定义 linear(a,b)，它返回计算a*x+b 的函数f(x)。使用内嵌的方法：

切换行号显示 lang=en id=CA-1f88d7f712308ea216b7b96ea8092c7c7faf3e75_023 dir=ltr 1

```
2 def linear(a,b):
3     def result(x):
4         return a*x + b
5     return result
```

或者使用可调用的类:

切换行号显示 lang=en id=CA-9c6c42a1ffc649723a5c2993cb81ccaf2f386cb6_024 dir=ltr 1

```
2 class linear:
3     def __init__(self, a, b):
4         self.a, self.b = a,b
5     def __call__(self, x):
6         return self.a * x + self.b
```

两种方法都是：

切换行号显示 lang=en id=CA-9f5720323a60abd9b1cb9212c6731ece0f14d2f8_025 dir=ltr 1

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

给出一个可调用对象，taxes(10e6) 0.3 * 10e6 + 2。

用可调用对象的方法有个缺点，那就是这样做会慢一些且代码也会长一些。但是，注意到一系列的可调用对象可通过继承共享信号。

切换行号显示 lang=en id=CA-6b8440cbd5b914636c1bd386ee9bc93ea0704fa3_026 dir=ltr 1

```
2 class exponential(linear):
3     # __init__ inherited
4     def __call__(self, x):
5         return self.a * (x ** self.b)
```

对象可以对若干方法封装状态信息：

切换行号显示 lang=en id=CA-78dbdefe5a36b43d4c00f081720231c57c857140_027 dir=ltr 1

```
2 class counter:
```

```
3       value = 0
4       def set(self, x): self.value = x
5       def up(self): self.value=self.value+1
6       def down(self): self.value=self.value-1
7
8 count = counter()
9 inc, dec, reset = count.up, count.down, count.set
```

这里inc(), dec() 和 reset() 运性起来就像是一组共享相同计数变量的函数。

1.2.8. 如何在python中复制一个对象?

通常，使用copy.copy() 或 copy.deepcopy()。并不是所有的对象都可以被复制，但大多数是可以的。

某些对象可以被简单地多的方法复制。字典有个copy() 方法:

切换行号显示 lang=en id=CA-bdbe21df74f7ec4b05770b0ebb0f6831816d72b7_028 dir=ltr 1

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

序列可以通过slicing来复制:

切换行号显示 lang=en id=CA-6e9cc915504f54eed658041132ab91c4a9b1d30f_029 dir=ltr 1

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

1.2.9. 如何查看某个对象的方法和属性?

对于一个用户定义的类的实例x，dir(x) 返回一个按字母排序的列表，其中包含了这个实例的属性和方法，类的属性。

1.2.10. 如何在运行时查看某个对象的名称?

一般来说是不行的，因为实际上对象并没有名称。实质上，赋值经常将一个名称绑定到一个值；对于def 和 class 语句也是一样, 但在那种情况下这个变量是可调用的。考虑以下代码：

切换行号显示 lang=en id=CA-bdc0afe4417afdd88245228b4ccc42d0753e7b60_030 dir=ltr 1

```
2 class A:
3       pass
4
5 B = A
6
7 a = B()
8 b = a
```

```
 9 print b
10 <__main__.A instance at 016D07CC>
11 print a
12 <__main__.A instance at 016D07CC>
```

理论上这个类有名称：尽管它被绑定到两个名称，通过名称B进行调用，这个新创建的实例仍然被作为是类A的实例。但是，因为两个名称都被绑定到同样的值，因此说这个实例的名称到底是A还是B是不可能的。

一般来说，让你的代码知道特定对象的名称并不是必要的。除非去特意地编写一个自省程序，否则这往往意味着需要改变一下代码。

在comp.lang.python, Fredrik Lundh 曾经给出了一个极好的解答：

> 就好像在你家走廊发现一只猫，而你想知道它的名字：这只猫（对象）不会告诉你它的名字，它实际上也不在乎 —— 所以唯一的方法就是问你的邻居们（名称空间namespace）……
>
> ...如果你发现它有很多名字或根本就没有名字的话也不要惊讶！

1.2.11. 是否有类似C的 "?:" 三元操作符?

没有。在很多情况下你可以用"a and b or c"模拟 a?b:c with , 但这样做有个缺陷：如果b是zero(或 empty, 或 None -- 只要为false) 则c被选择。在很多情况下你可以查看代码以保证这种情况不会发生(例如，因为b是个常数或是一种永远不会为false的类型), 但是通常来书它的确是个问题。

Tim Peters (本人希望是Steve Majewski) 有以下建议: (a and [b] or [c])[0]. 因为 [b] 是一个永远不会为false的列表，所以错误的情况不会发生；然后对整个表达式使用 [0] 来得到想要的b或者c。很难看，但在你重写代码并且使用'if'很不方便的情况下，这种方式是有效的。

最好的方式还是用 if...else 语句。另一种方法就是用一个函数来实现 "?:" 操作符：

切换行号显示 lang=en id=CA-e9999d9147ce0586bb5069bd716ff695c2481f22_031 dir=ltr

```
 1
 2 def q(cond,on_true,on_false):
 3     if cond:
 4         if not isfunction(on_true): return on_true
 5         else: return apply(on_true)
 6     else:
 7         if not isfunction(on_false): return on_false
 8         else: return apply(on_false)
```

在大多数情况下，你会直接传递b和c: q(a,b,c)。为防止在不合适的情况下计算 b 或者 c，用一个lambda函数封装它们，例如：q(a,lambda: b, lambda: c)。

为什么python没有if-then-else表达式。有几个回答： 很多语言在没有这个的情况下也工作得很好；它会减少可读代码的数量；还没有足够多的python风格的语法；通过对标准库的搜索，发现几乎没有这种情况：通过使用 if-then-else 表达式让代码的可读性更好。

在 2002年, PEP 308 提交了若干语法建议，整个社区对此进行了一次非决定性的投票。很多人喜欢某个语法而反对另外的语法；投票结果表明，很多人宁愿没有三元操作符，也不愿意创建一种新的令人讨厌的语法，。

1.2.12. 能不能在python中编写复杂的行程序?

是的。这经常发生在将lambda嵌入到lambda的情况，根据 Ulf Bartelt，有以下三个例子:

切换行号显示 lang=en id=CA-c589ab76083bdfb074bfbcd41150ec9c9f0444b3_032 dir=ltr 1

```
2 # Primes < 1000
3 print filter(None,map(lambda y:y*reduce(lambda x,y:x*y!=0,
4 map(lambda x,y=y:y%x,range(2,int(pow(y,0.5)+1))),1),range(2,1000)))
5
6 # First 10 Fibonacci numbers
7 print map(lambda x,f=lambda x,f:(x<=1) or (f(x-1,f)+f(x-2,f)): f(x,f),
8 range(10))
9
10 # Mandelbrot set
11 print (lambda Ru,Ro,Iu,Io,IM,Sx,Sy:reduce(lambda x,y:x+y,map(lambda y,
12 Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,Sy=Sy,L=lambda yc,Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,i=IM,
13 Sx=Sx,Sy=Sy:reduce(lambda x,y:x+y,map(lambda x,xc=Ru,yc=yc,Ru=Ru,Ro=Ro,
14 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
15 >=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(
16 64+F(Ru+x*(Ro-Ru)/Sx,yc,0,0,i)),range(Sx))):L(Iu+y*(Io-Iu)/Sy),range(Sy
17 ))))(-2.1, 0.7, -1.2, 1.2, 30, 80, 24)
18 #       \___ ___    \___ ___    |     |     |__ lines on screen
19 #             V                V          |     |______ columns on screen
20 #             |                |          |___________ maximum of "iterations"
21 #             |                |_____________________ range on y axis
22 #             |______________________________________ range on x axis
```

小朋友不要在家里尝试这个!

1.3. 数字和字符串

1.3.1. 如何指定十六进制和八进制整数?

要指定一个八进制数字，在八进制之前加个0。例如，将a设置成八进制的10，输入：

切换行号显示 lang=en id=CA-c66975dd4cdd16aebc945e5721f9a85510d45d52_033 dir=ltr 1

```
2 >>> a = 010
3 >>> a
4 8
```

十六进制也很简单。在十六进制前加个0x。十六进制数可以大写也可以小写。比如，在python解释器中:

切换行号显示 lang=en id=CA-dc8cccf1867e11068f1bb344922c11a759343f94_034 dir=ltr 1

```
2 >>> a = 0xa5
3 >>> a
4 165
5 >>> b = 0XB2
6 >>> b
7 178
```

1.3.2. 为什么 -22 / 10 返回 -3?

这是因为 i%j 跟 j 为同样类型。如果你想那样，且又想：

切换行号显示 lang=en id=CA-2e9ca01693f10dc23c1f68847bc463a335a67f7e_035 dir=ltr 1

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

那么整数除法就必须返回一个浮点值。C也有这个要求，编译器截断i/j，并使i的类型和i%j一样。

在实际应用中， i%j 的j是负数的可能性很小。当j是正数时，多数情况（实际上是所有情况）下i%j >= 0是很有用的。 如果现在是10点，那么200小时以前是多少？ -190 % 12 2 是正确的，而-190 % 12 -10 则是个bug。

1.3.3. 如何将一个字符串转换成数字?

对于整数，使用内建的 int() 类型构造器, 例如 int('144') 144。类似的，float() 转换成浮点数，例如 float('144') 144.0。

默认的，这些数字被解释成十进制，所以 int('0144') 144 而 int('0x144') 则抛出ValueError异常。 int(string, base) 提供了第二个参数来指定类型，所以int('0x144', 16) 324。

如果base被指定为0，则会按python的规则来解释：开头为一个 '0' 表示八进制，而 '0x' 表示十六进制。

如果你只是将字符串转换成数字，不用使用内建函数 eval()。eval()会慢很多并 有安全风险：某人传递给python一个表达式，可能会有意想不到的边界效应。例如，某人传递 __import__('os').system("rm -rf $HOME") 会清除掉你的home目录。

eval()

也能将数字解释成为python表达式，所以 eval('09') 会给出一个语法错误，因为python将开头

为'0'的数字认为是八进制(base 8)。

1.3.4. 如何将数字转换成字符串?

比如，数字144转换成字符串 '144', 使用内建函数 str()。 如果想用八进制或十六进制表示，使用内建函数hex() 或 oct()。对于格式化，使用 [../../doc/lib/typesseq-strings.html % operator] ，比如，"%04d" % 144 为 '0144' ， "%.3f" % (1/3.0) 为 '0.333'。更多细节查看库参考手册。

1.3.5. 如何在字符串的特定位置进行修改?

不能，因为字符串是不可改的。如果你确实需要一个具有这个能力的对象，将字符串转换成列表或使用数组模块：

```
切换行号显示 lang=en id=CA-d32681a571851d034e3e919d8449f44632596eee_036 dir=ltr 1
 2 >>> s = "Hello, world"
 3 >>> a = list(s)
 4 >>> print a
 5 ['H', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd']
 6 >>> a[7:] = list("there!")
 7 >>> ''.join(a)
 8 'Hello, there!'
 9
10 >>> import array
11 >>> a = array.array('c', s)
12 >>> print a
13 array('c', 'Hello, world')
14 >>> a[0] = 'y' ; print a
15 array('c', 'yello world')
16 >>> a.tostring()
17 'yello, world'
```

1.3.6. 如何使用字符串来调用函数/方法?

这里有几种方式：

最好的一种就是使用字典将字符串映射到函数。这种方法的最大好处就是字符串不用匹配函数的名称。这也是模拟case construct的主要方式：

```
切换行号显示 lang=en id=CA-14c23bc8a1ca86ba2173522592819af60c897c0a_037 dir=ltr
1
 2 def a():
```

```
    pass

def b():
    pass

dispatch = {'go': a, 'stop': b}    # Note lack of parens for funcs

dispatch[get_input()]()    # Note trailing parens to call function
```

使用内建函数getattr():

切换行号显示 lang=en id=CA-46685622cb03e733d1b243efd2debb29622cffc2_038 dir=ltr

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

注意，getattr()可工作于任何对象，包括类，类实例，模块等。 这种方法被用在标准库中的若干地方，就像：

切换行号显示 lang=en id=CA-315bb01cab43b9d5c0148a8e88b588ad8af1f7e4_039 dir=ltr

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

    def do_bar(self):
        ...

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

使用locals() 或 eval() 来获得函数名称：

切换行号显示 lang=en id=CA-582834b218e487a6dd65309450769fef13e38b2c_040 dir=ltr

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

fname = "myFunc"

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

```
 9
10 f = eval(fname)
11 f()
```

注意：使用eval() 慢而且危险。如果你对字符串的内容没有绝对控制权，其他人可能传递一个字符串而导致某个任意的函数被执行。

1.3.7. 是否有跟Perl中的chomp()类似的函数，用来去除字符串结尾处的新行符?

从Python 2.2起，可以使用S.rstrip("\r\n") 来去除字符串S结尾处的任何行符，而不会去除结尾处的其它空白符。如果字符串S并不只是一行，并在末尾有若干个空行，所有空行的行符都会被去除：

切换行号显示 lang=en id=CA-d05daec64e6efd2f816700079cec1be911efcc08_041 dir=ltr 1

```
2 >>> lines = ("line 1 \r\n"
3 ...          "\r\n"
4 ...          "\r\n")
5 >>> lines.rstrip("\n\r")
6 "line 1 "
```

当程序每次只能读取一行数据时，这样使用S.rstrip() 是很合适的。

对于老版本的Python, 分别有两个替代方法：

- 如果想去除所有的结尾空白符，使用字符串对象的 rstrip() 方法。这样会去掉所有的结尾空白符，而不只是一个新行符。
- 另外，如果在字符串S中只有一行，使用 S.splitlines()[0]。

1.3.8. 是否有实现 scanf() 或 sscanf() 功能的函数?

没有。

对于简单的输入分析，最简单的方法就是用string对象的 split() 将输入行分割成若干用空格分割的单词，然后用 int() 或float()将数字字符串转换成数字。split() 支持可选参数 "sep" 用来处理分隔符不是空格的情况。

对于更复杂的输入分析，正则表达式比C的sscanf() 更强大和更合适。

1.3.9. 'UnicodeError: ASCII [decoding,encoding] error: ordinal not in range(128)' 是什么意思?

这个错误表明python只能处理 7-bit 的 ASCII 字符串。这里有几种方法可以解决这个问题。

如果程序需要处理任意编码的数据，程序的运行环境一般都会指定它传给你的数据的编码。你需要用那个编码将输入数据转换成 Unicode 数据。例如，一个处理 email 或web输入的程序

会在 Content-Type 头里发现字符编码信息。在稍后将数据转换成Unicode时会使用到这个信息。假设通过 value 引用的字符串的编码为 UTF-8：

切换行号显示 lang=en id=CA-6e5cdf2101da67ada557e8049621353eeefbc2b3_042 dir=ltr 1

```
2 value = unicode(value, "utf-8")
```

会返回一个 Unicode 对象。如果数据没有被正确地用 UTF-8 编码，那么这个调用会触发一个 UnicodeError 异常。

如果你只是想把非 ASCII 的数据转换成 Unicode，你可以首先假定为 ASCII 编码，如果失败再产生 Unicode 对象。

切换行号显示 lang=en id=CA-8de5772defabab6f1982b7a4cdf430bebf8df0d8_043 dir=ltr 1

```
2 try:
3     x = unicode(value, "ascii")
4 except UnicodeError:
5     value = unicode(value, "utf-8")
6 else:
7     # value was valid ASCII data
8     pass
```

可以在python库中的一个叫sitecustomize.py 的文件中设定默认编码。但并不推荐这样 ，因为改变这个全局值可能会导致第三方的扩展模块出错。

注意，在 Windows 上有一种编码为 "mbcs"，它是根据你目前的locale使用编码。在很多情况下，尤其是跟 COM 一起工作的情况下，这是个合适的默认编码。

1.4. 序列 (Tuples/Lists)

1.4.1. 如何在 tuples 和 lists 之间转换?

函数 tuple(seq) 将任何序列(实际上,任何可遍历的对象)转换成一个tuple，并有着同样的元素和顺序。

例如，tuple([1, 2, 3]) 得到 (1, 2, 3)，而 tuple('abc') 得到 ('a', 'b', 'c')。 如果参数就是一个 tuple, 则不做任何复制而返回相同的对象，所以当你不确定一个对象是否是tuple时调用tuple()也没有额外的开销。

函数 list(seq) 将任何序列或任何可遍历的对象转换成一个list，并有着同样的元素和顺序。比如 list((1, 2, 3)) 得到 [1, 2, 3]，而 list('abc') 得到 ['a', 'b', 'c']。 如果参数就是一个列表，则执行一个复制操作，就像 seq[:] 一样。

1.4.2. 什么是负索引?

Python序列的索引可正可负。若使用正索引，0是第一个索引，1是第二个索引，以此类推。若使用负索引，-1 表示最后一个索引，-2表示倒数第二个，以此类推。比如seq[-n] 与

seq[len(seq)-n] 相同。

使用负索引带来很大的方便。比如 S-1 是除最后一个字符外的所有字符串，这在移除字符串结尾处的换行符时非常有用。

1.4.3. 如何反向遍历一个序列?

如果是一个列表, 最快的解决方法是

切换行号显示 lang=en id=CA-91d7694d4a25aa000e00bf16e84aa41047f7cef5_044 dir=ltr 1

```
2 list.reverse()
3 try:
4         for x in list:
5                 "do something with x"
6 finally:
7         list.reverse()
```

这样做有个缺点，就是当你在循环时，这个list被临时反转了。如果不喜欢这样，也可做一个复制。这样虽然看起来代价较大，但实际上比其它方法要快。

切换行号显示 lang=en id=CA-7cc62ed310fe7d511c8a04936c6f858632217528_045 dir=ltr 1

```
2 rev = list[:]
3 rev.reverse()
4 for x in rev:
5                 <do something with x>
```

如果它不是个列表，一个更普遍但也更慢的方法是：

切换行号显示 lang=en id=CA-d3ca9fdbc38cf23612ea20029cce18e774f25c23_046 dir=ltr 1

```
2 for i in range(len(sequence)-1, -1, -1):
3                 x = sequence[i]
4                 <do something with x>
```

还有一个更优雅的方法，就是定义一个类，使它像一个序列一样运行，并反向遍历(根据 Steve Majewski 的方法):

切换行号显示 lang=en id=CA-9162c523297eeef3001d227967bfe86d7facdd5e_047 dir=ltr 1

```
2 class Rev:
3                 def __init__(self, seq):
4                         self.forw = seq
5                 def __len__(self):
6                         return len(self.forw)
7                 def __getitem__(self, i):
```

```
8                       return self.forw[-(i + 1)]
```

你可以简单地写成:

切换行号显示 lang=en id=CA-94395f5f5d82aa0cd9f37fc24ac5c5cd9103c03b_048 dir=ltr 1

```
2 for x in Rev(list):
3         <do something with x>
```

然而，由于方法调用的开销，这是最慢的一种方法。

当使用 Python 2.3时，你可以使用一种扩展的slice语法:

切换行号显示 lang=en id=CA-5f3cfe67e45183554b90370d9f285eab4a03c775_049 dir=ltr 1

```
2 for x in sequence[::-1]:
3         <do something with x>
```

1.4.4. 怎样才能删掉一个列表中的重复元素?

Python Cookbook中有一个关于这个的较长的论述，提到了很多方法，参考：

http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52560

如果你不介意重新排列这个列表，那么对它进行排序并从list的末尾开始扫描，将重复元素删掉：

切换行号显示 lang=en id=CA-38014909ac525b18a54f2c27e2031bcd04a23634_050 dir=ltr 1

```
2 if List:
3     List.sort()
4     last = List[-1]
5     for i in range(len(List)-2, -1, -1):
6         if lastList[i]: del List[i]
7         else: last=List[i]
```

如果列表中所有的元素都可用作字典的键值（即它们都是hashable），那么通常这样更快：

切换行号显示 lang=en id=CA-aca3e5a3feac88f33ae6840d90ee190904e900bb_051 dir=ltr 1

```
2 d = {}
3 for x in List: d[x]=x
4 List = d.values()
```

1.4.5. 如何在python中使用数组?

使用列表:

切换行号显示 lang=en id=CA-6cce2d31e5d6a371ba0f548f717a18a9e9d61541_052 dir=ltr 1

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

列表对等于C或Pascal中的数组；最大的不同是python的列表可以包含很多不同的数据类型。

array 模块也可以提供方法来创建紧凑表示的固定类型数组，但是它的索引会比 列表慢。也要注意可定义类似数组且拥有各种特性的Numeric扩展和其它方式。

要获得Lisp风格链接的列表，你可以通过使用tuple来模拟cons cells。

切换行号显示 lang=en id=CA-149c76dcfefb5c1404f7af06304f0b89d9b497ab_053 dir=ltr 1

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

如果需要在运行时可更改，可以使用列表代替tuple。这里类似lisp car 的是 lisp_list[0] ，而类似 cdr 的是 lisp_list[1]。 仅当你确定需要时使用它，因为这样比使用python列表慢很多。

1.4.6. 如何使用多维列表?

你很有可能用这种方式来产生一个多维数组：

切换行号显示 lang=en id=CA-488268424998cafaa2e2557a2eb188bc5fef8b78_054 dir=ltr 1

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

如果你pirnt它的话似乎是正确的：

切换行号显示 lang=en id=CA-1e97a681fa9a019c5798eee3e6e9eb92cb022cc2_055 dir=ltr 1

2 >>> A

3 [[None, None], [None, None], [None, None]]

但是当你赋一个值时，它会出现在好几个地方：

切换行号显示 lang=en id=CA-7ebdf13f0638173af52ba1ba980492093657c286_056 dir=ltr 1

2 >>> A[0][0] = 5

3 >>> A

4 [[5, None], [5, None], [5, None]]

这是因为用 * 来复制时，只是创建了对这个对象的引用，而不是真正的创建了它。 *3 创建了一个包含三个引用的列表，这三个引用都指向同一个长度为2的列表。其中一个行的改变会显示在所有行中，这当然不是你想要的。

建议创建一个特定长度的list，然后用新的list填充每个元素：

切换行号显示 lang=en id=CA-a47e6a169f8e9e2bbeb6ee66d8f79d67bcee849a_057 dir=ltr 1

2 A = [None]*3

3 for i in range(3):

4 A[i] = [None] * 2

这样创建了一个包含三个不同的长度为2的列表。你也可以使用list comprehension：

切换行号显示 lang=en id=CA-680f829fb2c1d189934363c9a9aac701a43102de_058 dir=ltr 1

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

或者，你可以使用一个扩展来提供矩阵数据类型；Numeric Python 是最有名的。

1.4.7. 如何对一系列的对象应用方法?

使用list comprehension:

切换行号显示 lang=en id=CA-6eb973ddf6254edfa5615222ecbb5f7758bd01d3_059 dir=ltr 1

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

更一般的，可以使用以下函数：

切换行号显示 lang=en id=CA-553e6bdab2fcef77cf8f211ce433ad5bd80f5860_060 dir=ltr 1

```
2 def method_map(objects, method, arguments):
3         """method_map([a,b], "meth", (1,2)) gives [a.meth(1,2), b.meth(1,2)]"""
4         nobjects = len(objects)
5         methods = map(getattr, objects, [method]*nobjects)
6         return map(apply, methods, [arguments]*nobjects)
```

1.5. 字典

1.5.1. 如何按特定的顺序显示一个字典?

不能这样。字典按不可预测的顺序存储数据，所以字典的显示顺序也是不可预测的。

或许你正想保存一个可打印版本到一个文件，做某些更改后将其与其它显示的字典比较，这个回答会使你感到沮丧。在这种情况下，使用pprint模块来pretty-print字典，这样元素会按键值排序。

另一个复杂的多的方法就是继承UserDict.UserDict 并创建类SortedDict ，以一个可预知的顺序显示出来。这里有一个示例：

切换行号显示 lang=en id=CA-168ae34dd13ba93db1574008d58d78fade31eaf2_061 dir=ltr 1

```
2 import UserDict, string
3
4 class SortedDict(UserDict.UserDict):
5     def __repr__(self):
```

```
    result = []
    append = result.append
    keys = self.data.keys()
    keys.sort()
    for k in keys:
        append("%s: %s" % (`k`, `self.data[k]`))
    return "{%s}" % string.join(result, ", ")

___str__ = __repr__
```

虽然这不是个完美的解决方案，但它可以在你遇到的很多情况下工作良好，最大的缺陷就是，如果字典中某个值也是字典，那么将不会以任何特定顺序显示值。

1.5.2. 我想做一个复杂的排序，可以在python中完成一个Schwartzian变换吗?

可以，通过使用list comprehensions会非常简单。

根据Perl社区的Randal Schwartz的方法，创建一个矩阵，这个矩阵将列表的每个元素都映射到相应的“排序值”上，通过这个矩阵对列表进行排序。有一个字符串列表，用字符串的大写字母值排序：

切换行号显示 lang=en id=CA-b521cdc8456e718b08a8aeedc9d634a7b761fdd2_062 dir=ltr 1

```
2 tmp1 = [ (x.upper(), x) for x in L ] # Schwartzian transform
3 tmp1.sort()
4 Usorted = [ x[1] for x in tmp1 ]
```

对每个字符串的10-15位置的子域扩展的整数值进行排序：

切换行号显示 lang=en id=CA-e300ee5c67e762053fb40b65155f5de477a7e79f_063 dir=ltr 1

```
2 tmp2 = [ (int(s[10:15]), s) for s in L ] # Schwartzian transform
3 tmp2.sort()
4 Isorted = [ x[1] for x in tmp2 ]
```

注意到 Isorted 也能被这样计算：

切换行号显示 lang=en id=CA-8fca1fc70adfa340685ba8e5a9779063a0e35dc2_064 dir=ltr 1

```
2 def intfield(s):
3     return int(s[10:15])
4
5 def Icmp(s1, s2):
6     return cmp(intfield(s1), intfield(s2))
```

```
7
8 Isorted = L[:]
9 Isorted.sort(Icmp)
```

但是因为这个方法对L的每个元素调用intfield()多次，所以要比Schwartzian变换慢。

1.5.3. 如何根据一个list的值来对另一个list排序?

将它们合并成一个包含若干tuple的列表，对列表排序，然后选取你想要的元素。

切换行号显示 lang=en id=CA-138ea1098d3c248883f90a7d032477b5b3c40955_065 dir=ltr 1

```
2 >>> list1 = ["what", "I'm", "sorting", "by"]
3 >>> list2 = ["something", "else", "to", "sort"]
4 >>> pairs = zip(list1, list2)
5 >>> pairs
6 [('what', 'something'), ("I'm", 'else'), ('sorting', 'to'), ('by', 'sort')]
7 >>> pairs.sort()
8 >>> result = [ x[1] for x in pairs ]
9 >>> result
10 ['else', 'sort', 'to', 'something']
```

对于最后一步，一个替代方法是：

切换行号显示 lang=en id=CA-499024d94e2b60a346fe7d0fa1e872f7e789985e_066 dir=ltr 1

```
2 result = []
3 for p in pairs: result.append(p[1])
```

如果你发现这样做更清晰，你也许会使用这个替代方法。但是，对于长列表，它几乎会花去大约两倍的时间。为什么？首先，append() 操作需要重新分配内存，虽然它使用了一些技巧用来防止每次操作时都这样做，它的消耗依然很大。第二，表达式 "result.append" 需要一个额外的属性定位，第三，所有的函数调用也会减慢速度。

1.6. 对象

1.6.1. 什么是类?

类是在执行类语句时创建的特殊对象。类对象被用来作为模板创建实例对象，它包含了针对某个数据类型的数据（属性）和代码（方法）。

一个类可以继承一个或多个被称为基类的类。其继承了基类的属性和方法。这就允许通过继承来对类进行重定义。假设有一个通用的Mailbox类提供基本的邮箱存取操作，那么它的子类比如 MboxMailbox, MaildirMailbox, OutlookMailbox 可以处理各种特定的邮箱格式。

1.6.2. 什么是method（方法）?

method就是某个在类x中的函数，一般调用格式为 x.name(arguments...)。 方法在类定义中被定义成函数:

切换行号显示 lang=en id=CA-47cd552e9c893ec69e4b96532655c44a940d4de5_067 dir=ltr

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

1.6.3. 什么是self?

Self仅仅是method的第一个常规参数。一个method定义为 meth(self, a, b, c)，

对于定义这个method的类的某个实例x，调用时为 x.meth(a, b, c) ，而实际上是 meth(x, a, b, c)。

参 考 [../draft/general.html#why-must-self-be-used-explicitly-in-method-definitions-and-calls Why must 'self' be used explicitly in method definitions and calls?]

1.6.4. 如何确定某个对象是指定的类或子类的实例?

使用内建函数 isinstance(obj, cls)。 你可以通过一个tuple来检查某个对象是否是一系列类的实例，例如 isinstance(obj, (class1, class2, ...)), 并检查某个对象是否是python的内建类型，例如 isinstance(obj, str) or isinstance(obj, (int, long, float, complex))。

注意多数程序并不经常使用 isinstance() 来检查用户定义的类，如果你自己在编写某个类，一个更好的面向对象风格的方法就是定义一个封装特定功能的method，而不是检查对象所属的类然后根据这个来调用函数。例如，如果你有某个函数：

切换行号显示 lang=en id=CA-1a66badf6c5573acb1682d9417f5193deaf7e136_068 dir=ltr

```
2 def search (obj):
3     if isinstance(obj, Mailbox):
4         # ... code to search a mailbox
5     elif isinstance(obj, Document):
6         # ... code to search a document
7     elif ...
```

一个更好的方法就是对所有的类都定义一个search() 方法：

切换行号显示 lang=en id=CA-ed932c4aba965495c8897eec9bfac84ec4c79e74_069 dir=ltr

```
2 class Mailbox:
3     def search(self):
4         # ... code to search a mailbox
```

```
5
6 class Document:
7         def search(self):
8                 # ... code to search a document
9
10 obj.search()
```

1.6.5. 什么是delegation?

Delegation是一个面向对象技术（也被称为一种设计模式）。假设你有个类x并想改变它的某个方法method。 你可以创建一个新类，提供这个method的一个全新实现，然后将其它method都delegate到x中相应的method。

Python程序员可以轻易地实现delegation。比如，下面这个类像一个文件一样使用，但它将所有的数据都转换成大写：

切换行号显示 lang=en id=CA-85944f63739891706922c727e0f1411f862cf255_070 dir=ltr

```
2 class UpperOut:
3         def __init__(self, outfile):
4                 self.__outfile = outfile
5         def write(self, s):
6                 self.__outfile.write(s.upper())
7         def __getattr__(self, name):
8                 return getattr(self.__outfile, name)
```

在这里类UpperOut 重新定义了write() 方法，在调用self.outfile.write()方法之前，将字符串参数都转换成大写。所有其它的method都delegate到self.outfile 相应的method。这个delegation通过 __getattr__ 方法来完成；关于控制属性存取的更多信息，参考 [../../doc/ref/attribute-access.html the language reference] 。

注意到更多的情况下，delegation使人产生疑惑。 若需要修改属性，还需要在类中定义 __settattr__ 方法，并应小心操作。__setattr__ 的实现基本与以下一致：

切换行号显示 lang=en id=CA-adac8d9fd29182f62de79941371721857277a918_071 dir=ltr

```
2 class X:
3         ...
4         def __setattr__(self, name, value):
5                 self.__dict__[name] = value
6         ...
```

大多数__setattr__实现必须修改self.__dict__，用来存储自身的本地状态信息以防止无限递归。

1.6.6. 如果继承类里的某个方法覆盖了基类中的定义，如何从继承类

中调用基类的这个方法?

如果你使用的是新风格的类，使用内建函数 super():

切换行号显示 lang=en id=CA-76db8b2ba0145ca3dfc37e5055a11ce8c2ec14bb_072 dir=ltr

```
1
2 class Derived(Base):
3     def meth (self):
4         super(Derived, self).meth()
```

如果你使用的是经典风格的类：对于一个定义为 class Derived(Base): ... 的类 ，你可以调用 Base（或Base某个基类）的方法 meth()，例如 Base.meth(self, arguments...). 这里，Base.meth 是个未绑定的方法，你可以使用 self 参数。

1.6.7. 如何组织我的代码以使改变基类更容易?

你可以为基类定义一个别名，在你的类定义之前将真正的基类赋给它，并在你的整个类里使用别名。那么所有需要改变的就是赋给别名的值。另外，当你想动态决定使用哪个基类的情况（例如，根据资源的有效性），这个技巧也很方便。例如：

切换行号显示 lang=en id=CA-aee9d6f026b84f81d7ccb599b7028e750df1b629_073 dir=ltr

```
1
2 BaseAlias = <real base class>
3 class Derived(BaseAlias):
4     def meth(self):
5         BaseAlias.meth(self)
6         ...
```

1.6.8. 怎样创建静态类数据和静态类方法?

创建静态数据(C++ 或 Java中的说法)很简单；但不直接支持静态方法(同样是 C++ 或 Java中的说法)的创建。

对于静态数据，简单地定义一个类属性。当给这个属性赋予新值时，需要明确地使用类名称。

切换行号显示 lang=en id=CA-f6c32f38a4bb3b976d2d3252b67648c510e356b9_074 dir=ltr

```
1
2 class C:
3     count = 0     # number of times C.__init__ called
4
5     def __init__(self):
6         C.count = C.count + 1
7
8     def getcount(self):
9         return C.count    # or return self.count
```

对于isinstance(c, C)，c.count 同样引用 C.count， 除非被c本身重载，或是被基类搜索路径中从c.__class__到C上的某个类重载。

注意：在C的method中，类似 self.count = 42 的操作创建一个新的不相关实例，它在self本身的dict中被命名为 "count"。 重新邦定一个类静态数据必须指定类，无论是在method里面还是外面：

切换行号显示 lang=en id=CA-2be972d3c1cab97ae7e6afca1143b98b42746aa9_075 dir=ltr 1

```
2 C.count = 314
```

当你使用新类型的类时，便有可能创建静态方法:

切换行号显示 lang=en id=CA-8baf3c59750d72bf72cc190b85ec31e013ba342a_076 dir=ltr 1

```
2 class C:
3         def static(arg1, arg2, arg3):
4                 # No 'self' parameter!
5                 ...
6         static = staticmethod(static)
```

但是，创建静态方法的另一个更直接的方式是使用一个简单的模块级别的函数：

切换行号显示 lang=en id=CA-bbe774358ad42264dab0e12a8a9ee199c5adf60a_077 dir=ltr 1

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

如果每个模块可以定义一个类（或紧密相关的类结构），就可提供相应的封装。

1.6.9. 在python中如何重载构造函数?

这个答案对于所有的method都适用，但是问题一般都先出现在构造函数上。

在C++中你编写

切换行号显示 lang=en id=CA-e9957e8a819cacaac931811918c693177972015a_078 dir=ltr 1

```
2 class C {
3         C() { cout << "No arguments\n"; }
4         C(int i) { cout << "Argument is " << i << "\n"; }
5 }
```

在python中，你只能编写一个构造函数，通过使用默认参数来处理所有的情况。例如：

切换行号显示 lang=en id=CA-0ea2270e334816e772300bea19e1eda9805be6d9_079 dir=ltr 1

```
2 class C:
3         def __init__(self, i=None):
```

```
4           if i is None:
5               print "No arguments"
6           else:
7               print "Argument is", i
```

这与C++并不相同，但在实际应用中已相当接近。

你也可以使用变长参数列表，例如

切换行号显示 lang=en id=CA-c9aa1b61a7940a4d21a65251a19f4ed939edaffe_080 dir=ltr 1

```
2 def __init__(self, *args):
3     ....
```

同样的方法适用于所有的method定义。

1.6.10. 我想使用 __spam 却得到一个错误_SomeClassName__spam.

有两个前置下划线的变量提供了一个简单却有效的定义类私有变量的方法。任何spam (至少两个前置下划线，最多一个后置下划线)格式的标志符都会被替换为 _classnamespam, 其中 classname 是目前的类名称，并去掉了所有的前置下划线。

这并不能保证私有性：一个外部的用户可以直接连接到"_classnamespam" 属性，且所有的私有变量在对象的 __dict__中都是可见的。很多Python程序员从来不为私有变量名称烦恼。

1.6.11. 我的类定义了`__del__` 但在删除对象时并没有被调用.

有几个可能的原因。

del 语句并不一定要调用 __del__ -- 它只是简单地减少对象的引用计数，如果已为零便调用 __del__ 。

如果你的数据结构包含 循环链接（例如，在一个树种，每个child都有一个parent引用而每个parent都有一系列的child），那么引用计数永远不会返回至零。一旦python运行一个算法来检测这种循环，但可能在你的数据结构的最后一个引用结束后过一段时间才会运行垃圾收集，所以你的 __del__ 方法会在一个随机的时间被调用。当你想reproduce错误时，这是很不方便的。更糟的是，对象的 __del__ 方法以任意顺序执行。你可以运行 gc.collect() 来强制进行收集，但是也可能会有对象永远不会被收集的情况。

尽管有循环收集，为对象明确地定义一个 close() 用来在完成任务后被调用，依然是一个好主意。那么close() 方法会删除引用subobject的属性。不要直接调用 __del__ -- __del__ 应该调用 close() 而 close() 应该确定对于相同的对象它可以不止一次地被调用。

另一个防止循环引用的方法就是使用 "weakref" 模块，它允许你指向对象而不会增加引用计数。距离来说，对于树数据结构，应该对它们的parent和sibling（如果需要！）使用weak引用。

如果一个对象曾作为一个函数的local变量，而这个函数在一个except语句中捕获一个表达式，那么情况有所变化，对这个对象的引用在那个函数的stack frame中且被stack trace包含，即，这个对象引用仍然存在。一般的，调用 sys.exc_clear() 会清除最后一次记录的异常，以解决这个

问题。

最后，如果你的 __del__ 方法抛出一个异常，一个警告信息被输出到 sys.stderr。

1.6.12. 我如何得到一个给定类的所有实例 的列表?

Python并不跟踪某个类（或内建数据类型）的所有实例。你可以通过在类的构造函数中维护一个列表来跟踪所有的实例。

1.7. 模块

1.7.1. 如何创建一个 .pyc 文件?

当一个模块被首次import（或者是源代码文件比目前已编译好的文件更新）时，一个包含了编译好的代码的 .pyc 文件在这个 .py 文件所在的目录中被创建。

.pyc文件创建失败的一个可能原因是目录的许可权限。举例来说，你正在测试一个web服务器，编程时为某用户，但运行时又为另一用户，就会发生这种情况。如果你import一个模块，且python有这个能力（权限，剩余空间等）将编译好的模块写回目录，那么一个.pyc文件就会被自动创建。

在脚本的顶层运行python时，则不会认为引入了模块，.pyc文件也不会被创建。例如，如果你有一个顶层的模块 abc.py ，而它import了另一个模块 xyz.py, 当你运行abc时, xyz.pyc 会在xyz被import时被创建，但是abc.pyc不会被创建，因为它没有被import。

如果你需要创建 abc.pyc -- 即，为一个并没import的模块创建 .pyc 文件 -- 你可以使用 py_compile 和 compileall 模块。

py_compile 模块可以手动地编译任何模块。一种方式是使用这个模块的compile() 函数。

切换行号显示 lang=en id=CA-dca3054019e141c3cd21c26986212335014ab669_081 dir=ltr 1

```
2 >>> import py_compile
3 >>> py_compile.compile('abc.py')
```

这会将.pyc 文件写入abc.py 所在的目录(或者可以通过可选参数cfile 改变它）。

你也可以使用compileall 模块自动编译一个或若干目录中的所有文件。你可以在命令行里运行 compileall.py 并指定要编译的python文件的目录。

切换行号显示 lang=en id=CA-6c1e61180b0635d7e7cc58549bac52290b5063f1_082 dir=ltr 1

```
2 python compileall.py .
```

1.7.2. 如何查到目前这个模块的名称?

通过global变量__name__ 某个模块可以得知它的名称。如果它的值为 '__main__'，这个程序正作为一个脚本在运行。 有很多模块常通过import来使用，这些模块也提供了一个命令行界面或自测试功能，这些代码只在检查了 __name__ 后运行:

切换行号显示 lang=en id=CA-9f08421048af96a9c05f1828dcffc13d5adba17e_083 dir=ltr 1

```
2 def main():
3         print 'Running test...'
4         ...
5
6 if __name__    '__main__':
7         main()
```

1.7.3. 如何让模块互相import?

假设你有以下模块:

foo.py:

切换行号显示 lang=en id=CA-2623cfda33e4b23e5c7c5114d0894c32819c303f_084 dir=ltr 1

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

bar.py:

切换行号显示 lang=en id=CA-2bf021eb6da662df5e749e0572cb638a19b05889_085 dir=ltr 1

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

解释器按以下步骤执行：

main imports foo

Empty globals for foo are created
foo is compiled and starts executing
foo imports bar
Empty globals for bar are created
bar is compiled and starts executing
bar imports foo (which is a no-op since there already is a module named foo)
bar.foo_var = foo.foo_var

最后一步会失败，因为python还没有解释完 foo，而且foo的global symbol dictionary也是空的。

当你import foo, 然后在global代码中试图连接 foo.foo_var 时也会发生同样的事情。

至少有三个方法可解决这个问题。

Guido van Rossum 建议避免所有的from <module> import ...的用法，并将所有的代码都移到函数中。global变量和类的初始化应该只使用常量或内建函数。这意味着对所有import的模块的引用都使用 <module>.<name> 的方式。

Jim Roskind 建议在每个模块中采用以下步骤：

exports (globals, functions, and classes that don't need imported base classes)

import statements

active code (including globals that are initialized from imported values).

van Rossum 并不是很喜欢这种方法，因为import出现在一个奇怪的地方，但这样确实可以工作。

Matthias Urlichs 建议i重写你的代码，使你不必在开头就递归地import。

这些方法互相之间并不排斥。

1.7.4. `__import__`('x.y.z') 返回 <module 'x'>; 如何得到 z?

用:

切换行号显示 lang=en id=CA-2181c67e1670c8297f436bffe701fdcffa1ab053_086 dir=ltr 1

```
2 __import__('x.y.z').y.z
```

对于更现实的情况，你可能需要这样做：

切换行号显示 lang=en id=CA-c0827717aee9d07a726e82c72e4d7438b3423a9a_087 dir=ltr 1

```
2 m = __import__(s)
3 for i in s.split(".")[1:]:
4         m = getattr(m, i)
```

1.7.5. 当我对import的模块修改并重新import后却没有出现应有的改变，为什么?

处于效率和连续性的原因，python只在模块第一次被import时读取模块文件。如果不这样，在一个包含很多模块的程序中，若每个模块都import另一个相同的模块，会导致这个模块被多次读取。若要强行重读某个模块，这样做：

切换行号显示 lang=en id=CA-9373dd7c5b0f8d1cb10f0c0c2c73ba3d8e2d86d8_088 dir=ltr 1

```
2 import modname
3 reload(modname)
```

警告：这种方法并不是100%有效。特别地，模块包含以下语句

切换行号显示 lang=en id=CA-b4beb8ff9b0a55f2a7703c8c6083ac458cceb76f_089 dir=ltr 1

```
2 from modname import some_objects
```

仍会使用旧版本的对象。如果模块包含类定义，已存在的类实例也不会更新成新的定义。这会导致以下荒谬的结果：

切换行号显示 lang=en id=CA-be829f78612ab2eb6ebf72c0587fb2cd6097540f_090 dir=ltr 1

```
2 >>> import cls
3 >>> c = cls.C()                        # Create an instance of C
4 >>> reload(cls)
```

```
5 <module 'cls' from 'cls.pyc'>
6 >>> isinstance(c, cls.C)            # isinstance is false?!?
7 False
```

如果你print这个类对象的话，就会搞清楚这个问题的实质了：

切换行号显示 lang=en id=CA-59afce7a4725de206751012ef811d809f3e8e75a_091 dir=ltr 1

```
2 >>> c.__class__
3 <class cls.C at 0x7352a0>
4 >>> cls.C
5 <class cls.C at 0x4198d0>
```