# Python 3 初探，第 1 部分: Python 3 的新特性

**简介：** Python 3 是 Guido van Rossum 功能强大的通用编程语言的最新版本。它虽然打破了与 2.x 版本的向后兼容性，但却清理了某些语法方面的问题。本文是系列文章中的第一篇，介绍了影响该语言及向后兼容性的各种变化，并且还提供了新特性的几个例子。

Python 版本 3，也被称为 *Python 3000* 或 *Py3K*（仿效 Microsoft® Windows® 2000 操作系统而命名的昵称）是 Guido van Rossum 通用编程语言的最新版本。虽然新版本对该核心语言做了很多改进，但还是打破了与 2.x 版本的向后兼容性。其他一些变化则是人们期待已久的，比如：

- 真正的除法 — 例如，*1/2* 返回的是 *.5*。
- `long` 和 `int` 类型被统一为一种类型，删除了后缀 `L`。
- `True`、`False` 和 `None` 现在都是关键字。

本文 — Python 3 系列文章中的第一篇 — 的内容涵盖了新的 `print()` 函数、`input()`、输入/输出（I/O）的变化、新的 `bytes` 数据类型、字符串和字符串格式化的变化以及内置的 `dict` 类型的变化。本文面向的是那些熟悉 Python 并对新版本的变化很感兴趣但又不想费力读完所有 Python Enhancement Proposal（PEP）的编程人员。（本文后面的 参考资料 部分提供了有关这些 PEP 的链接。）

新的 `print()` 函数

如今，您将需要让手指习惯于键入 `print("hello")`，而不是原来的 `print "hello"`，这是因为 `print` 现在是一个函数，不再是一个语句。我知道，这多少有点痛苦。我认识的每个 Python 程序员 — 一旦安装了版本 3 并得到“语法不正确”错误 — 都会郁闷地大叫。我知道这两个额外的符号十分讨厌；我也知道这将会破坏向后兼容性。但是这种改变还是有好处的。

让我们考虑这样的情况，即需要将标准输出（`stdout`）重定向到一个日志。如下的例子会打开文件 `log.txt` 以便进行追加并将对象指定给 `fid`。之后，利用 `print>>` 将一个字符串重定向给文件 `fid`：

```
>>>fid = open("log.txt", "a")
>>>print>>fid, "log text"
```

另外一个例子是重定向给标准错误（sys.stderr）：

```
>>>print>>sys.stderr, "an error
occurred"
```

上述两个例子都不错，但还有更好的解决方案。新的语法只要求给 print() 函数的关键字参数 file 传递一个值就可以了。比如：

```
>>>fid = open("log.txt", "a")
>>>print("log.txt", file=fid)
```

这样的代码，语法更为清晰。另一个好处是通过向 sep 关键字参数传递一个字符串就能更改分割符（separator），通过向 end 关键字参数传递另外一个字符串就能更改结束字符串。要更改分割符，可以利用：

```
>>>print("Foo", "Bar", sep="%")
>>>Foo%Bar
```

总地来说，新的语法为：

```
print([object, ...][, sep=' '][,
end='endline_character_here'][,
file=redirect_to_here])
```

其中，方括号（[]）内的代码是可选的。默认地，若只调用 print() 自身，结果会追加一个换行符（ \n）。

从 raw_input() 到 input()

在 Python 版本 2.x 中，raw_input() 会从标准输入（sys.stdin）读取一个输入并返回一个字符串，且尾部的换行符从末尾移除。下面的这个例子使用 raw_input() 从命令提示符获取一个字符串，然后将值赋给 quest。

```
>>>quest = raw_input("What is your
quest? ")
What is your quest? To seek the holy
grail.
```

```
>>>quest
'To seek the holy grail.'
```

与之不同，Python 2.x 中的 input() 函数需要的是一个有效的 Python 表达式，比如 *3+5*。

最初，曾有人建议将 input() 和 raw_input() 从 Python 内置的名称空间一并删除，因此就需要进行导入来获得输入能力。这从方法上就不对；因为，简单键入：

```
>>>quest = input("What is your quest?")
```

将会变为：

```
>>>import sys
>>>print("What is your quest?")
>>>quest = sys.stdin.readline()
```

对于一个简单输入而言，这太过繁琐，并且对于一个新手，这未免太难理解。往往需要向他们讲述*模块* 和*导入* 究竟是怎么回事、字符串输出以及句点操作符又是如何工作的(如此麻烦的话，与 Java™ 语言就没什么差别了)。所以，在 Python 3 内，将 raw_input() 重命名为 input()，这样一来，无须导入也能从标准输入获得数据了。如果您需要保留版本 2.x 的 input() 功能，可以使用 eval(input())，效果基本相同。

有关 bytes 的简介

新的数据类型 bytes literal 及 bytes 对象的用途是存储二进制数据。此对象是 0 到 127 的不可修改的整数序列或纯粹的 ASCII 字符。实际上，它是版本 2.5 中 bytearray 对象的不可修改版本。一个 *bytes literal* 是一个前面冠以 *b* 的字符串 — 例如，*b'byte literal'*。对 bytes literal 的计算会生成一个新的 bytes 对象。可以用 bytes() 函数创建一个新的 bytes 对象。bytes 对象的构造函数为：

```
bytes([initializer[, encoding]])
```

例如：

```
>>>b = (b'\xc3\x9f\x65\x74\x61')
>>>print(b)
b'\xc3\x83\xc2\x9feta'
```

会创建一个 bytes 对象，但这是多余的，因为通过赋值一个 byte literal 就完全可以创建 bytes 对象。（我只是想要说明这么做是可行的，但是我并不建议您这么做。）如果您想要使用 iso-8859-1 编码，可以尝试下面的做法：

```
>>>b = bytes('\xc3\x9f\x65\x74\x61',
'iso-8859-1')
>>>print(b)
b'\xc3\x83\xc2\x9feta'
```

如果初始化器（initializer）是一个字符串，那么就必须提供一种编码。如果初始化器是一个 bytes literal，则无须指定编码类型：请记住，bytes literal 并不是字符串。但是与字符串相似，可以连接多个字节：

```
>>>b'hello' b' world'
b'hello world'
```

用 bytes() 方法代表二进制数据以及被编码的文本。要将 bytes 转变为 str，bytes 对象必须要进行解码(稍后会详细介绍)。二进制数据用 decode() 方法编码。例如：

```
>>>b'\xc3\x9f\x65\x74\x61'.decode()
'ßeta'
```

也可以从文件中直接读取二进制数据。请看以下的代码：

```
>>>data = open('dat.txt', 'rb').read()
>>>print(data) # data is a string
>>># content of data.txt printed out
here
```

它的功能是打开文件以便在二进制模式内读取一个文件对象，并在整个文件内进行读取。

字符串

Python 具有单一的字符串类型 str，其功能类似于版本 2.x 的 unicode 类型。换言之，所有字符串都是 unicode 字符串。而且 — 对非拉丁文的文本用户也非常方便 — 非-ASCII 标识符现在也是允许的。例如：

```
>>>césar = ["author", "consultant"]
>>>print(césar)
['author', 'consultant']
```

在 Python 之前的版本内，repr() 方法会将 8-位字符串转变为 ASCII。例如：

```
>>>repr('é')
"'\\xc3\\xa9'"
```

现在，它会返回一个 unicode 字符串：

```
>>>repr('é')
"'é'"
```

正如我之前提到的，这个字符串是内置的字符串类型。

字符串对象和字节对象是不兼容的。如果想要得到字节的字符串表示，需要使用它的 decode() 方法。相反，如果想要从该字符串得到 bytes literal 表示，可以使用字符串对象的 encode() 方法。

字符串格式化方面的变化

很多 Python 程序员都感觉用来格式化字符串的这个内置的 % 操作符太有限了，这是因为：

- 它是一个二进制的操作符，最多只能接受两个参数。
- 除了格式化字符串参数，所有其他的参数都必须用一个元组（tuple）或是一个字典（dictionary）进行挤压。

这种格式化多少有些不灵活，所以 Python 3 引入了一种新的进行字符串格式化的方式（版本 3 保留了 % 操作符和 string.Template 模块）。字符串对象现在均具有一个方法 format()，此方法接受位置参数和关键字参数，二者均

传递到 *replacement 字段* 。Replacement 字段在字符串内由花括号（{}）标示。replacement 字段内的元素被简单称为一个*字段*。以下是一个简单的例子：

```
>>>"I love {0}, {1}, and
{2}".format("eggs", "bacon",
"sausage")
'I love eggs, bacon, and sausage'
```

字段 {0}、{1} 和 {2} 通过位置参数 eggs、 bacon 和 sausage 被传递给 format() 方法。如下的例子显示了如何使用 format() 通过关键字参数的传递来进行格式化：

```
>>>"I love {a}, {b}, and
{c}".format(a="eggs", b="bacon",
c="sausage")
'I love eggs, bacon, and sausage'
```

下面是另外一个综合了位置参数和关键字参数的例子：

```
>>>"I love {0}, {1}, and
{param}".format("eggs", "bacon",
param="sausage")
'I love eggs, bacon, and sausage'
```

请记住，在关键字参数之后放置非关键字参数是一种语法错误。要想转义花括号，只需使用双倍的花括号，如下所示：

```
>>>"{{0}}".format("can't see me")
'{0}'
```

位置参数 can't see me 没有被输出，这是因为没有字段可以输出。请注意这不会产生错误。

新的 format() 内置函数可以格式化单个值。比如：

```
>>>print(format(10.0, "7.3g"))
      10
```

换言之，*g* 代表的是 *一般格式*，它输出的是宽度固定的值。小数点前的第一个数值指定的是最小宽度，小数点后的数值指定的是精度。format specifier 的完整语法超出了本文的讨论范围，更多信息，可以参见本文的 参考资料 小节。

内置 dict 类型的变化

3.0 内的另一个重大改变是字典内 dict.iterkeys()、

dict.itervalues() 和 dict.iteritems() 方法的删除。取而代之的

是 .keys()、 .values() 和 .items()，它们被进行了修补，可以返回轻量的、类似于集的容器对象，而不是键和值的列表。这样的好处是在不进行键和条目复制的情况下，就能在其上执行 set 操作。例如：

```
>>>d = {1:"dead", 2:"parrot"}
>>>print(d.items())
<built-in method items of dict object at
0xb7c2468c>
```

**注意：**在 Python 内，*集* 是惟一元素的无序集合。

这里，我创建了具有两个键和值的一个字典，然后输出了 d.items() 的值，

返回的是一个对象，而不是值的列表。可以像 set 对象那样测试某个元素的成员资格，比如：

```
>>>1 in d # test for membership
True
```

如下是在 dict_values 对象的条目上进行迭代的例子：

```
>>>for values in d.items():
...     print(values)
...
dead
parrot
```

不过，如果您的确想要得到值的列表，可以对所返回的 dict 对象进行强制类型转换。比如：

```
>>>keys = list(d.keys())
>>>print(keys)
[1,2]
```

新的 I/O

# 元类

Wikipedia 对元类的定义是这样的,“一个*元类* 是这样一个类,其实例也是类。”在本系列的第 2 部分我会对这个概念进行详细的介绍。

在深入研究 I/O 的新机制之前，很有必要先来看看抽象基类（ abstract base classes，ABC）。更深入的介绍将会在本系列的第 2 部分提供。

*ABC* 是一些无法被实例化的类。要使用 ABC，子类必须继承自此 ABC 并且还要覆盖其抽象方法。如果方法的前缀使用 `@abstractmethod` 修饰符（decorator），那么此方法就是一个抽象方法。新的 ABC 框架还提供了 `@abstractproperty` 修饰符以便定义抽象属性。可以通过导入标准库模块 `abc` 来访问这个新框架。清单 1 所示的是一个简单的例子。

**清单 1. 一个简单的抽象基类**

```
from abc import ABCMeta

class
SimpleAbstractClass(metaclass=ABCMeta):
    pass

SimpleAbstractClass.register(list)

assert isinstance([],
SimpleAbstractClass)
```

`register()` 方法调用接受一个类作为其参数并会让此 ABC 成为所注册类的子类。这一点可以通过在最后一行上调用 `assert` 语句进行验证。清单 2 是使用修饰符的另外一个例子。

**清单 2. 使用修饰符的一个抽象基类**

```
from abc import ABCMeta, abstractmethod

class abstract(metaclass=ABCMeta):
    @abstractmethod
    def absMeth(self):
        pass

class A(abstract):
    # must implement abstract method
    def absMeth(self):
        return 0
```

了解了 ABC 之后，我们就可以继续探究新的 I/O 系统了。之前的 Python 发布版都缺少一些重要但是出色的函数，比如用于类似于流的对象的 `seek()`。 *类似于流的对象* 是一些具有 `read()` 和 `write()` 方法的类似于文件的对象 — 比如，socket 或文件。Python 3 具有很多针对类似于流的对象的 I/O 层 — 一个原始的 I/O 层、一个被缓冲的 I/O 层以及一个文本 I/O 层 — 每层均由其自身的 ABC 及实现定义。

打开一个流还是需要使用内置的 `open(fileName)` 函数，但是也可以调用 `io.open(fileName))`。这么做会返回一个缓冲了的文本文件；`read()` 和 `readline()` 会返回字符串（请注意，Python 3 内的所有字符串都是 unicode）。您也可以使用 `open(fileName, 'b')` 打开一个缓冲了的二进制文件。在这种情况下，`read()` 会返回字节，但 `readline()` 则不能用。

此内置 `open()` 函数的构造函数是：

```
open(file,mode="r",buffering=None,encoding=None,errors=None,newline=None,closefd=True)
```

可能的模式有：

- **r**：读
- **w**：打开供写入
- **a**：打开供追加

- **b**：二进制模式
- **t**：文本模式
- **+**：打开一个磁盘文件供更新
- **U**：通用换行模式

默认的模式是 rt，即打开供读取的文本模式。

buffering 关键字参数的期望值是以下三个整数中的一个以决定缓冲策略：

- **0**：关闭缓冲
- **1**：行缓冲
- **> 1**：完全缓冲（默认）

默认的编码方式独立于平台。关闭文件描述符或 closefd 可以是 True 或 False。如果是 False，此文件描述符会在文件关闭后保留。若文件名无法奏效的话，那么 closefd 必须设为 True。

open() 返回的对象取决于您所设置的模式。表 1 给出了返回类型。

**表 1. 针对不同打开模式的返回类型**

| 模式 | 返回对象 |
|---|---|
| **文本模式** | TextIOWrapper |
| **二进制** | BufferedReader |
| **写二进制** | BufferedWriter |
| **追加二进制** | BufferedWriter |
| **读/写模式** | BufferedRandom |

**请注意：**文本模式可以是 w、 r、wt、 rt 等。

清单 3 中所示的例子打开的是一个缓冲了的二进制流以供读取。

**清单 3. 打开一个缓冲了的二进制流以供读取**

```
>>>import io
>>>f = io.open("hashlib.pyo", "rb")  #
open for reading in binary mode
>>>f                              # f is
```

```
a BufferedReader object
<io.BufferedReader object at
0xb7c2534c>
>>>f.close()                          #
close stream
```

BufferedReader 对象可以访问很多有用的方法，比如 isatty、peek、raw、readinto、readline、readlines、seek、seekable、tell、writable、write 和 writelines。要想查看完整列表，可以在 BufferedReader 对象上运行 dir()。

结束语

Python 社区是否会接??版本 3 还尚在人们的猜测之中。打破向后兼容性意味着将要为两种版本提供支持。一些项目开发人员可能不太想迁移其项目，即便是使用版本 2 到 3 的转化器。就我个人而言，我发现从 Python 版本 2 迁移到 3 其实不过是对几个事情的重新认识：它当然不会像从 Python 迁移到 Java 或 Perl 语言那样变化强烈。很多变化是早就在人们意料中的，比如对 dict 的实质更改。执行 print() 远比执行 Java 的 System.out.println() 容易得多，学习起来也相对容易，所以的确能带来一些好处。

我猜想，blogosphere 内的一些帖子会让 Python 的支持者也会误认为其中的某些变更 — 例如对向后兼容性的打破 — 具有破坏性的影响。Lambda 本来就是准备好要删除的，只不过一直没有这么做，仍保留了其原始的格式。有关保留项目的完整列表，请访问 Python 核心开发站点。如果您具备足够的探索精神愿意深入研究所有的 PEP，那么您一定能够从中获得更深入的信息。

本系列的下一期文章将会涵盖更高级的主题，比如元类语法、ABC、修饰符、integer literal 支持、基类型和异常。

# Python 3 初探，第 2 部分：高级主题

**简介：** Python 3 是 Guido van Rossum 功能强大的通用编程语言的最新版本。它虽然打破了与 2.x 版本的向后兼容性，但却清理了某些语法方面的问题。本文是这个由两部分组成的系列文章中的第二篇，本文构建在此系列 前一期文章 的基础之上，内容涵盖了 Python 更多的新特性和更高深的一些主题，比如在抽象基类、元类和修饰符等方面的变化。

有关 Python 版本 3—，也即 *Python 3000* 或 *Py3K*— 的前一篇文章讨论了 Python 内打破向后兼容性的一些基本变化，比如新的 `print()` 函数、`bytes` 数据类型以及 `string` 类型的变化。本文是该系列文章的第 2 部分，探究了更为高深的一些主题，比如抽象基类（ABC）、元类、函数注释和修饰符（decorator）、整型数（integer literal）支持、数值类型层次结构以及抛出和捕获异常，其中的大多数特性仍然会打破与版本 2x 产品线的向后兼容性。

类修饰符

在 Python 之前的版本中，对方法的转换必须在方法定义之后进行。对于较长的方法，此要求将定义的重要组成部分与 Python Enhancement Proposal (PEP) 318（有关链接，请参见 参考资料）给出的外部接口定义分离。下面的代码片段演示了这一转换要求：

**清单 1. Python 3 之前版本中的方法转化**

```
def myMethod(self):
    # do something

myMethod = transformMethod(myMethod)
```

为了让此类情景更易于读懂，也为了避免必须多次重用相同的方法名，在 Python 版本 2.4 中引入了方法修饰符。

*修饰符* 是一些方法，这些方法可以修改其他方法并返回一个方法或另外一个可调用对象。对它们的注释是在修饰符的名称前冠以“at”符号（@）— 类似 Java™ 注释的语法。清单 2 显示了实际应用中的修饰符。

**清单 2. 一个修饰符方法**

```
@transformMethod
```

```
def myMethod(self):
    # do something
```

修饰符是一些纯粹的语法糖（syntactic sugar）— 或者（如 Wikipedia 所言）“对计算机语言语法的补充，这些补充并不影响语言的功能，而是会让语言变得更易于被人使用。”修饰符的一种常见用法是注释静态方法。比如，清单 1 和清单 2 相当，但清单 2 更容易被人读懂。

定义修饰符与定义其他方法无异：

```
def mod(method):
    method.__name__ = "John"
    return method

@mod
def modMe():
    pass

print(modMe.__name__)
```

更棒的是 Python 3 现在不仅支持针对方法的修饰符，并且支持针对类的修饰符，所以，取代如下的用法：

```
class myClass:
    pass

myClass = doSomethingOrNotWithClass(myClass)
```

我们可以这样使用：

```
@doSomethingOrNotWithClass
class myClass:
    pass
```

元类

*元类* 是这样一些类，这些类的实例也是类。Python 3 保留了内置的、用来创建其他元类或在运行时动态创建类的 `metaclass` 类型。如下的语法仍旧有效：

```
>>>aClass = type('className',
   (object,),
   {'magicMethod': lambda cls : print("blah
blah")})
```

上述语法接受的参数包括：作为类名的字符串、被继承对象的元组（可以是一个空的元组）和一个包含可以添加的方法的字典（也可以是空的）。当然，也可以从类型继承并创建您自己的元类：

```
class meta(type):
    def __new__(cls, className, baseClasses,
dictOfMethods):
        return type.__new__(cls, className,
baseClasses, dictOfMethods)
```

# 只允许关键字的参数

Python 3 已经更改了函数参数分配给 “参数槽（parameter slot）” 的方式，可以在传递进的参数内使用星号（*）以便不接受可变长度的参数。命名的参数必须在 * 之后 — 比如，def meth(*, arg1): pass。更多信息，请参考 Python 文档或 PEP 3102（有关链接，请参见 参考资料）。

**注意：**如果上面两个例子起不到任何作用，我强烈建议您阅读 David Mertz 和 Michele Simionato 合写的有关元类的系列文章。相关链接，请参见 参考资料。

请注意，现在，在类定义中，关键字参数被允许出现在基类列表之后 — 通常来讲，即 class Foo(*bases, **kwds): pass。使用关键字参数 metaclass 将元类传递给类定义。比如：

```
>>>class aClass(baseClass1, baseClass2,
metaclass = aMetaClass): pass
```

旧的元类语法是将此元类分配给内置属性 __metaclass__：

```
class Test(object):
    __metaclass__ = type
```

而且，既然有了新的属性 — __prepare__ — 我们就可以使用此属性为新的类名称空间创建字典。在类主体被处理之前，先会调用它，如清单 3 所示。

**清单 3. 使用 the __prepare__ attribute 的一个简单元类**

```
def meth():
    print("Calling method")

class MyMeta(type):
    @classmethod
    def __prepare__(cls, name, baseClasses):
        return {'meth':meth}

    def __new__(cls, name, baseClasses,
classdict):
        return type.__new__(cls, name,
baseClasses, classdict)

class Test(metaclass = MyMeta):
    def __init__(self):
        pass

    attr = 'an attribute'

t = Test()
print(t.attr)
```

我们从 PEP 3115 节选了一个更为有趣的例子，如清单 4 所示，这个例子创建了一个具有其方法名称列表的元类，而同时又保持了类方法声明的顺序。

**清单 4. 保持了类成员顺序的一个元类**

```
# The custom dictionary
class member_table(dict):
    def __init__(self):
        self.member_names = []

    def __setitem__(self, key, value):
        # if the key is not already defined, add
to the
        # list of keys.
        if key not in self:
            self.member_names.append(key)

        # Call superclass
        dict.__setitem__(self, key, value)

# The metaclass
```

```
class OrderedClass(type):
    # The prepare function
    @classmethod
    def __prepare__(metacls, name, bases): #
No keywords in this case
        return member_table()

    # The metaclass invocation
    def __new__(cls, name, bases, classdict):
        # Note that we replace the classdict
with a regular
        # dict before passing it to the
superclass, so that we
        # don't continue to record member names
after the class
        # has been created.
        result = type.__new__(cls, name,
bases, dict(classdict))
        result.member_names =
classdict.member_names
        return result
```

在元类内所做的这些改变有几个原因。对象的方法一般存储于一个字典，而这个字典是没有顺序的。不过，在某些情况下，若能保持所声明的类成员的顺序将非常有用。这可以通过让此元类在信息仍旧可用时，较早地涉入类的创建得以实现— 这很有用，比如在 C 结构的创建中。借助这种新的机制还能在将来实现其他一些有趣的功能，比如在类构建期间将符号插入到所创建的类名称空间的主体以及对符号的前向引用。PEP 3115 提到更改语法还有美学方面的原因，但是对此尚存在无法用客观标准解决的争论（到 PEP 3115 的链接，请参见 参考资料）。

抽象基类

正如我在 Python 3 初探，第 1 部分：Python 3 的新特性 中提到的，ABC 是一些不能被实例化的类。Java 或 C++ 语言的程序员应该对此概念十分熟悉。Python 3 添加了一个新的框架 —abc— 它提供了对 ABC 的支持。

这个 abc 模块具有一个元类（ABCMeta）和 修饰符（@abstractmethod 和 @abstractproperty）。如果一个 ABC 具有一个 @abstractmethod 或 @abstractproperty，它就不能被实例化，但必须在一个子类内被覆盖。比如，如下代码：

```
>>>from abc import *
>>>class C(metaclass = ABCMeta): pass
>>>c = C()
```

这些代码是可以的，但是不能像下面这样编码：

```
>>>from abc import *
>>>class C(metaclass = ABCMeta):
...    @abstractmethod
...    def absMethod(self):
...       pass
>>>c = C()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class C
with abstract methods absMethod
```

更好的做法是使用如下代码：

```
>>>class B(C):
...    def absMethod(self):
...       print("Now a concrete method")
>>>b = B()
>>>b.absMethod()
Now a concrete method
```

ABCMeta 类覆盖属性 __instancecheck__ 和 __subclasscheck__，借此可以重载内置函数 isinstance() 和 issubclass()。要向 ABC 添加一个虚拟子类，可以使用 ABCMeta 提供的 register() 方法。如下所示的简单示例：

```
>>>class TestABC(metaclass=ABCMeta): pass
>>>TestABC.register(list)
>>>TestABC.__instancecheck__([])
True
```

它等同于使用 isinstance(list, TestABC)。您可能已经注意到 Python 3 使用 __instancecheck__，而非 __issubclass__，使用 __subclasscheck__，而非 __issubclass__，这看起来更为自然。若将参

数 isinstance(subclass, superclass) 反转成，比如 superclass.__isinstance__(subclass)，可能会引起混淆。可见，语法 superclass.__instancecheck__(subclass) 显然更好一点。

在 collections 模块内,可以使用几个 ABC 来测试一个类是否提供了特定的一个接口：

```
>>>from collections import Iterable
>>>issubclass(list, Iterable)
True
```

表 1 给出了这个集合框架的 ABC。

**表 1. 这个集合框架的 ABC**

| ABC | Inherits |
|---|---|
| Container | |
| Hashable | |
| Iterable | |
| Iterator | Iterable |
| Sized | |
| Callable | |
| Sequence | Sized, Iterable, Container |
| MutableSequence | Sequence |
| Set | Sized, Iterable, Container |
| MutableSet | Set |
| Mapping | Sized, Iterable, Container |
| MutableMapping | Mapping |
| MappingView | Sized |
| KeysView | MappingView,Set |
| ItemsView | MappingView,Set |
| ValuesView | MappingView |

# 集合

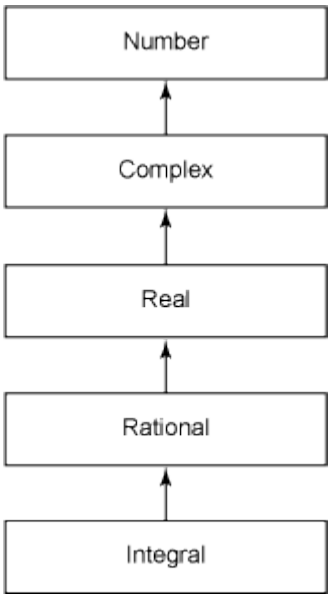
集合框架包括容器数据类型、双端队列（即 *deque*）以及一个默认的字典（即 *defaultdict* ）。一个 deque 支持从前面或后面进行追加和弹出。

defaultdict 容器是内置字典的一个子类，它（根据 Python 3 文档）“覆盖一个方法并添加一个可写的实例变量。”除此之外，它还充当一个字典。此外，集合框架还提供了一个数据类型工厂函数 namedtuple()。

ABC 类型层次结构

Python 3 现支持能代表数值类的 ABC 的类型层次结构。这些 ABC 存在于 numbers 模块内并包括 Number、 Complex、Real、 Rational 和 Integral。图 1 显示了这个数值层次结构。可以使用它们来实现您自己的数值类型或其他数值 ABC。

**图 1. 数值层次结构**




# 数值塔（numerical tower）

Python 的数值层次结构的灵感来自于 Scheme 语言的数值塔。

新模块 fractions 可实现这个数值 ABC Rational。此模块提供对有理数算法的支持。若使用 dir(fractions.Fraction)，就会注意到它具有一些属

性，比如 imag、 real 和 __complex__。根据数值塔的原理分析，其原因在于 Rationals 继承自 Reals，而 Reals 继承自 Complex。

抛出和捕获异常

在 Python 3 内，except 语句已经被修改为处理语法不清的问题。之前，在 Python version 2.5 内，try . . . except 结构，比如：

```
>>>try:
...    x = float('not a number')
... except (ValueError, NameError):
...    print "can't convert type"
```

可能会被不正确地写为：

```
>>> try:
...    x = float('not a number')
... except ValueError, NameError:
...    print "can't convert type"
```

后一种格式的问题在于 ValueError 异常将永远捕获不到，因为解释器将会捕获 ValueError 并将此异常对象绑定到名称 NameError。这一点可以从下面的示例中看出：

```
>>> try:
...    x = float('blah')
... except ValueError, NameError:
...    print "NameError is ", NameError
...
NameError is invalid literal for float(): not
a number
```

所以，为了处理语法不清的问题，在想要将此异常对象与另一个名称绑定时，逗号（,）会被替换成关键字 as。如果想要捕获多个异常，必须使用括号（()）。清单 5 中的代码展示了 Python 3 内的两个合乎语法的示例。

**清单 5. Python 3 内的异常处理**

```
# bind ValueError object to local name ex
try:
    x = float('blah')
except ValueError as ex:
    print("value exception occurred ", ex)

# catch two different exceptions
simultaneously
try:
    x = float('blah')
except (ValueError, NameError):
    print("caught both types of exceptions")
```

异常处理的另一个改变是*异常链* — 隐式或显式。清单 6 给出了隐式异常链的一个示例。

**清单 6. Python 3 内的隐式异常链**

```
def divide(a, b):
    try:
        print(a/b)
    except Exception as exc:
        def log(exc):
        fid = open('logfile.txt') # missing 'w'
        print(exc, file=fid)
        fid.close()

        log(exc)

divide(1,0)
```

divide() 方法试图执行除数为零的除法，因而引发了一个异常：

ZeroDivisionError。但是，在异常语句的嵌套 log() 方法内，print(exc,

file=fid) 试图向一个尚未打开的文件进行写操作。Python 3 抛出异常，如清单 7 所示。

**清单 7. 隐式异常链示例的跟踪**

```
Traceback (most recent call last):
  File "chainExceptionExample1.py", line 3,
in divide
  print(a/b)
ZeroDivisionError: int division or modulo by
zero

During handling of the above exception,
another exception occurred:

Traceback (most recent call last):
  File "chainExceptionExample1.py", line 12,
in <module>
    divide(1,0)
  File "chainExceptionExample1.py", line 10,
in divide
    log(exc)
  File "chainExceptionExample1.py", line 7,
in log
    print(exc, file=fid)
  File "/opt/python3.0/lib/python3.0/io.py",
line 1492, in write
    self.buffer.write(b)
  File "/opt/python3.0/lib/python3.0/io.py",
line 696, in write
    self._unsupported("write")
  File "/opt/python3.0/lib/python3.0/io.py",
line 322, in _unsupported
    (self.__class__.__name__, name))
io.UnsupportedOperation:
BufferedReader.write() not supported
```

请注意，这两个异常都被处理。在 Python 的早期版本中，ZeroDivisionError 将会丢失，得不到处理。那么这是如何实现的呢？

__context__ 属性，比如 ZeroDivisionError，现在是所有异常对象的一部分。在本例中，被抛出的 IOError 的 __context__ 属性 “仍为” __context__ 属性内的 ZeroDivisionError。

除 __context__ 属性外，异常对象还有一个 __cause__ 属性，它通常被初始化为 None。这个属性的作用是以一种显式方法记录异常的原因。__cause__ 属性通过如下语法设置：

```
>>> raise EXCEPTION from CAUSE
```

它与下列代码相同：

```
>>>exception = EXCEPTION
>>>exception.__cause__ = CAUSE
>>>raise exception
```

但更为优雅。清单 8 中所示的示例展示了显式异常链。

**清单 8. Python 3 中的显式异常链**

```
class CustomError(Exception):
    pass

try:
    fid = open("aFile.txt") # missing 'w' again
    print("blah blah blah", file=fid)
except IOError as exc:
    raise CustomError('something went wrong')
from exc
```

正如之前的例子所示，print() 函数抛出了一个异常，原因是文件尚未打开以供写入。清单 9 给出了相应的跟踪。

**清单 9. 异常跟踪**

```
Traceback (most recent call last):
  File "chainExceptionExample2.py", line 5,
in <module>
    fid = open("aFile.txt")
  File "/opt/python3.0/lib/python3.0/io.py",
line 278, in __new__
    return open(*args, **kwargs)
  File "/opt/python3.0/lib/python3.0/io.py",
line 222, in open
```

```
    closefd)
 File "/opt/python3.0/lib/python3.0/io.py",
line 615, in __init__
    _fileio._FileIO.__init__(self, name,
mode, closefd)
IOError: [Errno 2] No such file or directory:
'aFile.txt'

The above exception was the direct cause of the
following exception:

Traceback (most recent call last):
File "chainExceptionExample2.py", line 8, in
<modulei>
  raise CustomError('something went wrong')
from exc
__main__.CustomError: something went wrong
```

请注意，在异常跟踪中的一行 “The above exception was the direct cause of the following exception,” 之后的是另一个对导致 CustomError “something went wrong” 异常的跟踪。

添加给异常对象的另一个属性是 __traceback__。如果被捕获的异常不具备其 __traceback__ 属性，那么新的跟踪就会被设置。如下是一个简单的例子：

```
from traceback import format_tb

try:
    1/0
except Exception as exc:
    print(format_tb(exc.__traceback__)[0])
```

请注意，format_tb 返回的是一个列表，而且此列表中只有一个异常。

# with 语句

从 Python 版本 2.6 开始，with 已经成为了一个关键字，而且这一属性无需再通过 __future__ 导入。

## 整数支持和语法

Python 支持不同进制的整型字符串文本 — 八进制、十进制（最明显的！）和十六进制 — 而现在还加入了二进制。八进制数的表示已经改变：八进制数现在均以一个前缀 `0o` 或 `0O`（即，数字零后跟一个大写或小写的字母 *o*）开头。比如，八进制的 13 或十进制的 11 分别如下表示：

```
>>>0o13
11
```

新的二进制数则以前缀 `0b` 或 `0B`（即，数字零后跟一个大写或小写的字母 *b*）开头。十进制数 21 用二进制表示应为：

```
>>>0b010101
21
```

`oct()` 和 `hex()` 方法已被删除。

## 函数注释

*函数注释* 会在编译时将表述与函数的某些部分（比如参数）相关联。就其本身而言，函数注释是无意义的 — 即，除非第三方库对之进行处理，否则它们不会被处理。函数注释的作用是为了标准化函数参数或返回值被注释的方式。函数注释语法为：

```
def methodName(param1: expression1, ...,
paramN:
expressionN)->ExpressionForReturnType:
    ...
```

例如，如下所示的是针对某函数的参数的注释：

```
def execute(program:"name of program to be
executed", error:"if something goes wrong"):
    ...
```

如下的示例则注释了某函数的返回值。这对于检查某函数的返回类型非常有用：

```
def getName() -> "isString":
```

...

函数注释的完整语法可以在 PEP 3107（相关链接，请参见 参考资料）内找到。

结束语

# 意外收获

在我看来，为 Python 3 添加的最棒的一个模块是 antigravity。启动 Python，

然后在命令行键入 import antigravity。您一定不会失望。

Python 3 最终的发布版在 2008 年 12 月份初就已经发布。自那之后，我曾经查阅过一些博客，试图了解人们对向后不兼容性问题的反应。虽然，我不能断言社区在某些程度上已经达成了官方的共识，但我阅读过的博客所呈现的观点呈两极分化。Linux® 开发社区似乎不太喜欢转变到版本 3，因为需要移植大量代码。相比之下，因为新版本对 unicode 支持方面的改进，很多 Web 开发人员则欢迎转变。

我想要说明的一点是，在您做决定是否要移植到新版本之前，应该仔细阅读相关的 PEP 和开发邮件列表。这些 PEP 解释了各项更改的初衷、带来的益处及其实现。不难看出，这些更改的做出经过了深思熟虑和激烈讨论。本系列所展示的这些主题旨在让普通的 Python 程序员无需遍阅所有的 PEP 就能立即对这些更改有大致的概念。