Python 语言参考手册Python Reference Manual

Release 2.3.3

Guido van Rossum
Fred L. Drake, Jr., editor
翻译团队: **见文末名单**

译文最后修改: 2004**年**2**月**18**日**

PythonLabs
Email: docs@python.org

译者声明:

本文档的翻译工作目前尚未结束, **一些工作虽已完成**, **但还很粗糙**. **尤其是某些术语很有可能存在歧义**, **由于**Python**的中文资料不多**, **可参考的材料实在有限**, **译者只能根据自身实际经验和亲身体会总结提炼**. **但译者水平有限**, **希望各位同仁多多提意见**.**使这份文档更加完善**.

更新时间和性质:

2002.03.24, **修正性更新**

修正记录:

见ChangLog.html

意见和建议:

你可以在LinuxForum**上的**Python**编程区发贴子**,**或者给我发**E-mail (1) (2).

进度:

* **正文全部完成**.

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for rapid application development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Python**是一种解释性的，面向对象的，具有动态语义的高级程序设计语言。它内建高级数据结构，配以动态类型和动态捆绑，使其在快速应用开发中非常有利，就像脚本或粘合语言一样将已存在的构件连接在一起。**Python**的简单性和句法的易学性使其代码具有优秀的可读性，因此维护程序的成本得以大大降低。**Python**具有模块和包的概念，以支持程序的模块化和代码重用。在主流平台上，**Python**的解释器和大量标准库都可以免费地以源代码形式或可执行文件形式获得，并且可以自由发布。**

This reference manual describes the syntax and "core semantics" of the language. It is terse, but attempts to be exact and complete. The semantics of non-essential built-in object types and of the built-in functions and modules are described in the *Python Library Reference*. For an informal introduction to the language, see the *Python Tutorial*. For C or C++ programmers, two additional manuals exist: *Extending and Embedding the Python Interpreter* describes the high-level picture of how to write a Python extension module, and the *Python/C API Reference Manual* describes the interfaces available to C/C++ programmers in detail.

本参考手册描述了该语言的语法和“ 核心语义 ”。手册本身是比较简洁的，但尽可能写得准确和完整。那些非基本的内置对象类型、内置函数和模块的语义在《Python **库参考》中进行描述。对于语言的浅显介绍，可以看看《**Python **入门手册》. 对于**C**和**C++**程序员, 有两个文档可供参考：《扩展和嵌入**Python**解释器》是对**Python**扩展模块设计的总体介绍；《**Python/C API **参考手册》则向**C/C++**程序员们细致地描述了可以使用的接口。**

目录

第一章 简介Introduction **1**
1.1 记法Notation 1

第二章 词法分析Lexical analysis **3**
2.1 行结构Line structure 3
2.2 其它语言符号Other tokens 7
2.3 标识符和关键字Identifiers and keywords 7
2.4 字面值Literals 8
2.5 运算符Operators 12
2.6 分隔符Delimiters 12

第三章 !!数据模型Data model **15**
3.1 对象, 值和类型Objects, values and types 15
3.2 标准类型层次The standard type hierarchy 16
3.3 特殊方法名Special method names 26

第四章 运行模型Execution model **41**
4.1 !!! 代码块, 运行结构框架和命名空间Naming and binding 41
4.2 异常Exceptions 42

第五章 表达式Expressions **45**
5.1 数值型间的转换Arithmetic conversions 45
5.2 原子Atoms 46
5.3 基元Primaries 48
5.4 幂运算符The power operator 53
5.5 一元算术运算符Unary arithmetic operations 53
5.6 二元算术运算符Binary arithmetic operations 54
5.7 移位运算符Shifting operations 55
5.8 二元位运算符Binary bit-wise operations 55
5.9 比较Comparisons 56
5.10 布尔运算Boolean operations 58
5.11 Lambda形式(lambda表达式) Lambdas 58
5.12 表达式表Expression lists 59
5.13 Evaluation order 59
5.14 总结Summary 59

第六章 简单语句Simple statements **61**
6.1 表达式语句Expression statements 61
6.2 断言语句Assert statements 62
6.3 赋值语句Assignment statements 62
6.4 pass语句The `pass` statement 65
6.5 del 语句The `del` statement 66
6.6 print语句The `print` statement 66
6.7 return语句The `return` statement 67

6.8 yield语句The `yield` statement . . . 67
6.9 raise语句The `raise` statement . . . 68
6.10 break语句The `break` statement . . . 69
6.11 continue语句The `continue` statement . . . 69
6.12 import 语句The `import` statement . . . 69
6.13 global语句The `global` statement . . . 72
6.14 exec 语句The `exec` statement . . . 72

第七章 复合语句Compound statements 75
7.1 if 语句The `if` statement . . . 76
7.2 while语句The `while` statement . . . 76
7.3 for语句The `for` statement . . . 77
7.4 try语句The `try` statement . . . 77
7.5 函数定义Function definitions . . . 79
7.6 类定义Class definitions . . . 80

第八章 顶层构件Top-level components 83
8.1 完整的Python程序Complete Python programs . . . 83
8.2 文件输入File input . . . 83
8.3 交互式输入Interactive input . . . 84
8.4 表达式输入Expression input . . . 84

附录 A History and License 85
A.1 History of the software . . . 85
A.2 Terms and conditions for accessing or otherwise using Python . . . 86

附录 B 修正记录 89

附录 C 翻译团队 91

第一章

简介Introduction

This reference manual describes the Python programming language. It is not intended as a tutorial.

这个手册描述了Python**程序设计语言**, **本文档的目的不是入门**.

While I am trying to be as precise as possible, I chose to use English rather than formal specifications for everything except syntax and lexical analysis. This should make the document more understandable to the average reader, but will leave room for ambiguities. Consequently, if you were coming from Mars and tried to re-implement Python from this document alone, you might have to guess things and in fact you would probably end up implementing quite a different language. On the other hand, if you are using Python and wonder what the precise rules about a particular area of the language are, you should definitely be able to find them here. If you would like to see a more formal definition of the language, maybe you could volunteer your time — or invent a cloning machine :-).

虽然我试图尽可能的精确, **但还是选择了英语而不是形式化规范的方法描述**, **其中句法分析和词法分析部分除外**. **这使的文档更具可读性**, **但却有可能存在歧义**. **因此**, **如果你来自火星并且想单单通过这篇文档就重新实现**Python, **你可能得猜测一些东西**.**事实上你很可能以实现一门完全不同的语言而告终**; **另一方面**, **如果你在使用**Python**并且想了解其某个细节的精确规则**, **你可以在这里明确地得到它**; **如果你想获得更多的语言形式化定义**, **可能就要自己搞了**—- **或者发明一个克隆机**:-).

It is dangerous to add too many implementation details to a language reference document — the implementation may change, and other implementations of the same language may work differently. On the other hand, there is currently only one Python implementation in widespread use (although a second one now exists!), and its particular quirks are sometimes worth being mentioned, especially where the implementation imposes additional limitations. Therefore, you'll find short "implementation notes" sprinkled throughout the text.

在语言参考文档中加入过多的实现细节是危险的事情—-**实现会改变**, **并且一个语言不同的实现可能以不同的方式工作**. **另一方面**, **当前只有一个**Python**的实现得到广泛使用**(**虽然已经存在了第二个实现**). **这里有些细节还是被提及了**, **特别是某种实现增加了限制时**, **因此你可以在本文档中找到**"**实现注意**"**的标记**.

Every Python implementation comes with a number of built-in and standard modules. These are not documented here, but in the separate *Python Library Reference* document. A few built-in modules are mentioned when they interact in a significant way with the language definition.

每个Python**实现都提供了大量的内建和标准模块**, **它们不在这个文档的介绍范围之内**, **但可以在另一个叫做**Python **库参考手册的文档中找到**. **少量与语言定义密切相关的内建模块也在这被介绍了**.

1.1 记法Notation

The descriptions of lexical analysis and syntax use a modified BNF grammar notation. This uses the following style of definition:

在描述词法和句法分析时候, **我们使用不甚严格的**BNF, **通常是以下的定义方式**:

```
name:           lc_letter (lc_letter | "_")*
lc_letter:      "a"..."z"
```

The first line says that a `name` is an `lc_letter` followed by a sequence of zero or more `lc_letter`s and underscores. An `lc_letter` in turn is any of the single characters '`a`' through '`z`'. (This rule is actually adhered to for the names defined in lexical and grammar rules in this document.)

第一行说明name**为**lc_letter**后跟随零个以上**(**包括零个**)lc_letter**或下划线的序列**. lc_letter**是**"a"**至**"z"**中任意一个字符**.(**实际上**, **这个**"**名字**"**的定义贯穿于本文档的整个词法和语法规则中**)

Each rule begins with a name (which is the name defined by the rule) and a colon. A vertical bar (`|`) is used to separate alternatives; it is the least binding operator in this notation. A star (`*`) means zero or more repetitions of the preceding item; likewise, a plus (`+`) means one or more repetitions, and a phrase enclosed in square brackets (`[ ]`) means zero or one occurrences (in other words, the enclosed phrase is optional). The `*` and `+` operators bind as tightly as possible; parentheses are used for grouping. Literal strings are enclosed in quotes. White space is only meaningful to separate tokens. Rules are normally contained on a single line; rules with many alternatives may be formatted alternatively with each line after the first beginning with a vertical bar.

每个规则以一个名字(**为所定义的规则的名字**)**和一个冒号为开始**. **竖线**(—)**用于分隔可选项**.**这是记法中结合性最弱的符号**.**星号**(*)**意味着前一项的零次或多次的重复**; **同样**, **加号**(+)**意味着一次或多次的重复**. **在方括号**([])**中的内容意味着它可以出现零次或一次**(**也就是说它是可选的**).**星号和加号与前面的项尽可能地紧密的结合**, **小括号用于分组**.**字符串的字面值用引号括住**.**空白字符仅仅在分隔语言符号**(token)**时有用**.**通常规则被包含在一行之中**, **有很多可选项的规则可能会被格式化成多行的形式**, **后续行都以一个竖线开始**.

In lexical definitions (as the example above), two more conventions are used: Two literal characters separated by three dots mean a choice of any single character in the given (inclusive) range of ASCII characters. A phrase between angular brackets (`<...>`) gives an informal description of the symbol defined; e.g., this could be used to describe the notion of 'control character' if needed.

在词法定义中(**如上例**), **有两个习惯比较常用**: **以三个句点分隔的一对串字面值意味着在给定（包括）的**ASCII**字符范围内任选一个字符。在尖括号**(¡¿)**中的短语给出了非正式的说明**, **例如**, **这用在了需要说明**"**控制字符**"**记法的时候**.

Even though the notation used is almost the same, there is a big difference between the meaning of lexical and syntactic definitions: a lexical definition operates on the individual characters of the input source, while a syntax definition operates on the stream of tokens generated by the lexical analysis. All uses of BNF in the next chapter ("Lexical Analysis") are lexical definitions; uses in subsequent chapters are syntactic definitions.

即使在句法和词法定义中使用的记号几乎相同, **但它们之间在含义上还是有着的很大不同**: **词法定义是在输入源的一个个字符上进行操作，而句法定义是在由词法分析所生成的语言符号流上进行操作。在下节**("**词法分析**")**中使用的**BNF**都是词法定义**, **以后的章节是句法定义**.

第二章

词法分析Lexical analysis

A Python program is read by a *parser*. Input to the parser is a stream of *tokens*, generated by the *lexical analyzer*. This chapter describes how the lexical analyzer breaks a file into tokens.

一个Python**程序由解析器读入**, **输入解析器的是一个语言符号流**, **由词法分析器生成**.**本章讨论词法分析器是如何把文件分隔成语言符号的**.

Python uses the 7-bit ASCII character set for program text. New in version 2.3: An encoding declaration can be used to indicate that string literals and comments use an encoding different from ASCII.. For compatibility with older versions, Python only warns if it finds 8-bit characters; those warnings should be corrected by either declaring an explicit encoding, or using escape sequences if those bytes are binary data, instead of characters.

Python**使用**7**比特长的**ASCII**字符集作为程序文本和串字面值**. 8**比特长的字符的也可以作串字面值和注释**, **但它们的解释是依赖于平台的**, **在串中插入八比特字符的正确方法是使用八进制数和十六进制数的转义字符**.

The run-time character set depends on the I/O devices connected to the program but is generally a superset of ASCII.

运行时字符集依赖于连接到程序的I/O**设备**, **但通常是**ASCII**的超集**.

Future compatibility note: It may be tempting to assume that the character set for 8-bit characters is ISO Latin-1 (an ASCII superset that covers most western languages that use the Latin alphabet), but it is possible that in the future Unicode text editors will become common. These generally use the UTF-8 encoding, which is also an ASCII superset, but with very different use for the characters with ordinals 128-255. While there is no consensus on this subject yet, it is unwise to assume either Latin-1 or UTF-8, even though the current implementation appears to favor Latin-1. This applies both to the source character set and the run-time character set.

向后兼容性备忘: **假定**8**位字符集是**ISO Latin-1(**一种**ASCII**码的超集，它覆盖了大部分使用拉丁字母的西方语言**.)**看起来是个不错的做法**, **但是未来可能是支持**Unicode**的编辑器更流行一些**, **通常使用**UTF-8(**另一种**ASCII**码的超集**)**编码**, **但是对于顺序在**128**到**255**之间的字符用法两者存在很大的区别。然而关于这点还没有一致的意见，假定为**Latin-1**或**UTF-8**都是不明智的，尽管当前的实现偏向于**Latin-1, **这一点对于源程序字符集和运行字符集都是适用的。**

2.1 行结构Line structure

A Python program is divided into a number of *logical lines*.

一个Python**程序被分成多个逻辑行**.

2.1.1 逻辑行Logical lines

The end of a logical line is represented by the token NEWLINE. Statements cannot cross logical line boundaries except where NEWLINE is allowed by the syntax (e.g., between statements in compound statements). A logical line is constructed from one or more *physical lines* by following the explicit or implicit *line joining* rules.

逻辑行以一个NEWLINE(**新行**)**语言符号结束**, **语句不能跨多个逻辑行**, **除非语法上允许**NEWLINE(**例如**, **在复合语句的中的语句序列**).**一个逻辑行由一个物理行**, **或者以显式**/**隐式行连接规则连接的多个物理行构成**.

2.1.2 物理行Physical lines

A physical line ends in whatever the current platform's convention is for terminating lines. On UNIX, this is the ASCII LF (linefeed) character. On Windows, it is the ASCII sequence CR LF (return followed by linefeed). On Macintosh, it is the ASCII CR (return) character.

一个物理行由所在平台的断行符号结束.**在**Unix**上**, **是**ASCII LF(**换行**)**字符**; **在**DOS/Windows**上**, **是**ASCII**字符序列**CR LF(**回车加换行**); **在**Macintosh**上**, **是**ASCII CR(**回车**)**字符**.

2.1.3 注释Comments

A comment starts with a hash character (#) that is not part of a string literal, and ends at the end of the physical line. A comment signifies the end of the logical line unless the implicit line joining rules are invoked. Comments are ignored by the syntax; they are not tokens.

一个注释以#**字符（此时**, **它不能是串字面值的一部分）开始，结束于该物理行的结尾**.**如果没有隐式的行连接**, **那么注释就意味着该逻辑行的终止**.**注释为句法分析所忽略**, **它们不记作语言符号**.

2.1.4 Encoding declarations

If a comment in the first or second line of the Python script matches the regular expression 「`coding[=:]\s*([\w-_.]+)`」, this comment is processed as an encoding declaration; the first group of this expression names the encoding of the source code file. The recommended forms of this expression are

```
# -*- coding: <encoding-name> -*-
```

which is recognized also by GNU Emacs, and

```
# vim:fileencoding=<encoding-name>
```

which is recognized by Bram Moolenar's VIM. In addition, if the first bytes of the file are the UTF-8 byte-order mark (`'\xef\xbb\xbf'`), the declared file encoding is UTF-8 (this is supported, among others, by Microsoft's **notepad**).

If an encoding is declared, the encoding name must be recognized by Python. The encoding is used for all lexical analysis, in particular to find the end of a string, and to interpret the contents of Unicode literals. String literals are converted to Unicode for syntactical analysis, then converted back to their original encoding before interpretation starts. The encoding declaration must appear on a line of its own.

2.1.5 显式行连接Explicit line joining

Two or more physical lines may be joined into logical lines using backslash characters (\), as follows: when a physical line ends in a backslash that is not part of a string literal or comment, it is joined with the following forming a single logical line, deleting the backslash and the following end-of-line character. For example:

两个或更多物理行可以使用反斜线字符(\)**合并成一个逻辑行**, **具体地说**: **当一个物理行结束在一个反斜线（此时**, **这个反斜线不能是串字面值或注释的一部分）处时，它就同下面的物理行合并成一个逻辑行，同时将反斜线和跟着的行结束符删除。**

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # Looks like a valid date
        return 1
```

A line ending in a backslash cannot carry a comment. A backslash does not continue a comment. A backslash does not continue a token except for string literals (i.e., tokens other than string literals cannot be split across physical lines using a backslash). A backslash is illegal elsewhere on a line outside a string literal.

以反斜线结尾的行不能在其后加注释, 反斜线不能续注释行. 除了串字面值, 也不能续语言符号(也就是说,其它不是串字面值的语言符号不能通过反斜线横跨数个物理行). 在串字面值之外其它地方出现的反斜线都是非法的.

2.1.6 隐式行连接Implicit line joining

Expressions in parentheses, square brackets or curly braces can be split over more than one physical line without using backslashes. For example:

在小括号, 中括号, 大括号中的表达式可以跨越多个物理行, 而不用反斜线, 例如:

```
month_names = ['Januari', 'Februari', 'Maart',      # These are the
               'April',   'Mei',      'Juni',       # Dutch names
               'Juli',    'Augustus', 'September',  # for the months
               'Oktober', 'November', 'December']   # of the year
```

Implicitly continued lines can carry comments. The indentation of the continuation lines is not important. Blank continuation lines are allowed. There is no NEWLINE token between implicit continuation lines. Implicitly continued lines can also occur within triple-quoted strings (see below); in that case they cannot carry comments.

隐式的行连接可以尾随注释.接续行的缩进可以不考虑.空的接续行是不允许的.在隐式的接续行中是不存在NEWLINE语言符号的, 隐式的行连接在三重引用串(后述)中也是合法的, 那种情况下不能加注释.

2.1.7 空行Blank lines

A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored (i.e., no NEWLINE token is generated). During interactive input of statements, handling of a blank line may differ depending on the implementation of the read-eval-print loop. In the standard implementation, an entirely blank logical line (i.e. one containing not even whitespace or a comment) terminates a multi-line statement.

一个仅包括空白, 制表符, 换页符和可能的注释的逻辑行, 是被忽略的(就是说没有NEWLINE语言符号产生). 在交互式输入语句时, 空行的处理可能不同, 依赖于输入-计算-输出(read-eval-print)循环的实现方式. 在标准实现中, 一个纯粹的空行(即不包括任何东西,甚至注释和空白)才结束多行语句.

2.1.8 缩进Indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

逻辑行的前导空白(空格和制表符)用于计算行的缩进层次, 层次然后用于语句的分组.

First, tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight (this is intended to be the same rule as used by UNIX). The total number of spaces preceding the first non-blank character then determines the line's indentation. Indentation cannot be split over multiple physical lines using backslashes; the whitespace up to the first backslash determines the indentation.

首先, 制表符被转换成(从左到右)一至八个空格, 这样直到包括替换部分的字符总数是八的倍数(这个规则也适用于UNIX)第一个非空白字符前的空格总数用于检测代码的缩进, 缩进不能用反斜线在多个物理行间接续; 反斜线之前的空白字符用于检测缩进.

Cross-platform compatibility note: because of the nature of text editors on non-UNIX platforms, it is unwise to use a mixture of spaces and tabs for the indentation in a single source file. It should also be noted that different platforms may explicitly limit the maximum indentation level.

跨平台兼容性注意：由于在非UNIX平台上的文本编辑器特性，在单个源文件里使用混合空格和制表符的缩进是不明智的。

A formfeed character may be present at the start of the line; it will be ignored for the indentation calculations above. Formfeed characters occurring elsewhere in the leading whitespace have an undefined effect (for instance, they may reset the space count to zero).

在行首可以出现换页符, 但它在以上的缩进计算中被忽略.出现在其它位置的换页符的作用是不确定的.(例如,它可能重置空格数为零)

The indentation levels of consecutive lines are used to generate INDENT and DEDENT tokens, using a stack, as follows.

连续行的缩进层次用于生成语言符号INDENT和DEDENT, 这个过程中使用了堆栈, 如下述:

Before the first line of the file is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one INDENT token is generated. If it is smaller, it *must* be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a DEDENT token is generated. At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero.

在文件第一行未读入之前, 一个零被压栈. 它以后也不会被弹出来. 堆栈中的数字都从底部向顶部增长, 在每个逻辑行的开头处, 行的缩进层次与栈顶比较, 如果相等则什么也不会发生; 如果大于栈顶, 将其压入栈中, 并产生一个INDENT语言符号;如果小于栈顶, 那么它应该是堆栈中已存在的数字中的一个, 堆栈中所有大于它的数都将被弹出, 并且每个都产生一个DEDENT语言符号, 到达文件尾时, 堆栈中大于零的数字都被弹出, 每弹出一个数字也产生一个DEDENT语言符号.

Here is an example of a correctly (though confusingly) indented piece of Python code:

这是一个有着正确缩进格式的Python代码的例子(虽然有点乱):

```
def perm(l):
        # Compute the list of all permutations of l
    if len(l) <= 1:
                  return [l]
    r = []
    for i in range(len(l)):
             s = l[:i] + l[i+1:]
             p = perm(s)
             for x in p:
              r.append(l[i:i+1] + x)
    return r
```

The following example shows various indentation errors:

下面的例子展示了不同的缩进错误:

```
 def perm(l):                       # error: first line indented
for i in range(len(l)):             # error: not indented
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # error: unexpected indent
        for x in p:
                r.append(l[i:i+1] + x)
            return r                # error: inconsistent dedent
```

(Actually, the first three errors are detected by the parser; only the last error is found by the lexical analyzer — the indentation of `return r` does not match a level popped off the stack.)

(**事实上**, **前三个错误是由解析器发现的**, **仅仅最后一个错误是由词法分析器找到的**—- return r **的缩进层次不与弹出堆栈中的数匹配**.)

2.1.9 符号间的空白Whitespace between tokens

Except at the beginning of a logical line or in string literals, the whitespace characters space, tab and formfeed can be used interchangeably to separate tokens. Whitespace is needed between two tokens only if their concatenation could otherwise be interpreted as a different token (e.g., ab is one token, but a b is two tokens).

除非位于逻辑行起首或者字符串当中，空格，制表符和换页符这些空白字符可以等同地用于分隔词法项(token)**。空白仅当两个符号在连接时会有其它含义时才使用**(**例如**, ab**是一个符号**, **但**a b**是两个符号**).

2.2 其它语言符号Other tokens

Besides NEWLINE, INDENT and DEDENT, the following categories of tokens exist: *identifiers*, *keywords*, *literals*, *operators*, and *delimiters*. Whitespace characters (other than line terminators, discussed earlier) are not tokens, but serve to delimit tokens. Where ambiguity exists, a token comprises the longest possible string that forms a legal token, when read from left to right.

除了NEWLINE, INDENT**和**DEDENT, **还有以下几类语言符号**:**标识符**, **关键字**, **字面值**, **运算符和分隔符**.**空白不是语言符号**(**除了断行符**, **前述**), **但可以用于分隔语言符号**.**当解释某语言符号有歧义时**, **该语言符号被看作是由一个尽可能长的合法词法项组成**(**自左至右**).

2.3 标识符和关键字Identifiers and keywords

Identifiers (also referred to as *names*) are described by the following lexical definitions:

标识符(**也叫做名字**)**由以下词法定义描述**:

```
identifier  ::=  (letter|"_") (letter | digit | "_")*
letter      ::=  lowercase | uppercase
lowercase   ::=  "a"..."z"
uppercase   ::=  "A"..."Z"
digit       ::=  "0"..."9"
```

Identifiers are unlimited in length. Case is significant.

标识符不限长, **区分大小写**.

2.3.1 关键字Keywords

The following identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

以下标识符用作保留字, 或者叫做语言的关键字, 并且不能作为普通标识符使用, 而且它们必须像下面的那样严格拼写.

```
and       del       for       is        raise
assert    elif      from      lambda    return
break     else      global    not       try
class     except    if        or        while
continue  exec      import    pass      yield
def       finally   in        print
```

Note that although the identifier `as` can be used as part of the syntax of `import` statements, it is not currently a reserved word.

In some future version of Python, the identifiers `as` and `None` will both become keywords.

2.3.2 保留的标识符类型Reserved classes of identifiers

Certain classes of identifiers (besides keywords) have special meanings. These classes are identified by the patterns of leading and trailing underscore characters:

（关键字以外的）某些类型的标识符具有特殊的含义, 如下:

`_*` Not imported by '`from` *module* `import *`'. The special identifier '_' is used in the interactive interpreter to store the result of the last evaluation; it is stored in the `__builtin__` module. When not in interactive mode, '_' has no special meaning and is not defined. See section 6.12, "The `import` statement."

不能通过" from moduls import *"**导入。特定标识符**"_"**在交互式解释器用于存储上次估值的结果. 它位于**__buildin__**模块之中. 在非交互方式时**,"_"**没有特殊含义, 且是未定义的**

Note: The name '_' is often used in conjunction with internationalization; refer to the documentation for the `gettext` module for more information on this convention.

`__*__` System-defined names. These names are defined by the interpreter and it's implementation (including the standard library); applications should not expect to define additional names using this convention. The set of names of this class defined by Python may be extended in future versions. See section 3.3, "Special method names."

系统定义的名字

`__*` Class-private names. Names in this category, when used within the context of a class definition, are re-written to use a mangled for to help avoid name clashes between "private" attributes of base and derived classes. See section 5.2.1, "Identifiers (Names)."

类私有名字

2.4 字面值Literals

Literals are notations for constant values of some built-in types.

字面值是某些内建类型常值的表示法.

2.4.1 串字面值String literals

String literals are described by the following lexical definitions:

串字面值由以下词法定义描述:

```
stringliteral    ::=  [stringprefix](shortstring | longstring)
stringprefix     ::=  "r" | "u" | "ur" | "R" | "U" | "UR" | "Ur" | "uR"
shortstring      ::=  "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring       ::=  "'''" longstringitem* "'''"
                      | '"""' longstringitem* '"""'
shortstringitem  ::=  shortstringchar | escapeseq
longstringitem   ::=  longstringchar | escapeseq
shortstringchar  ::=  <any ASCII character except "\" or newline or the quote>
longstringchar   ::=  <any ASCII character except "\">
escapeseq        ::=  "\" <any ASCII character>
```

One syntactic restriction not indicated by these productions is that whitespace is not allowed between the `stringprefix` and the rest of the string literal.

上面没有表示出来的一个句法限制是在stringprefix**和串字面值之间不允许有空白**.

In plain English: String literals can be enclosed in matching single quotes (`'`) or double quotes (`"`). They can also be enclosed in matching groups of three single or double quotes (these are generally referred to as *triple-quoted strings*). The backslash (`\`) character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character. String literals may optionally be prefixed with a letter '`r`' or '`R`'; such strings are called *raw strings* and use different rules for interpreting backslash escape sequences. A prefix of '`u`' or '`U`' makes the string a Unicode string. Unicode strings use the Unicode character set as defined by the Unicode Consortium and ISO 10646. Some additional escape sequences, described below, are available in Unicode strings. The two prefix characters may be combined; in this case, '`u`' must appear before '`r`'.

以英语的方式描述:串是以单引号(')**或双引号**("), **它们也可以用成对的三个单引号和双引号**(**这叫做三重引用串**), **反斜线**\可以用于引用其它有特殊含义的字符, **例如新行**, **反斜线本身**, **或者引用字符**.**串字面值可选地可以以**'u'**和**'U'**开头**, **这样它就是一个**"**原始串**"**了**, **它在解释反斜线时有着不同的规则**, **前缀有**'u'**和**'U'**的串是**Unicode**串**, Unicode**使用**Unicode**协会和**ISO 10646**定义的**Unicode**字符集**. **其它一些在**Unicode**中有效的转义字符一会儿会提到**. **这两个前缀可以组合使用**, **但**'u'**必须在**'r'**之前**.

In triple-quoted strings, unescaped newlines and quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the string. (A "quote" is the character used to open the string, i.e. either `'` or `"`.)

在三重引用串中, **未转义的新行和引用字符是允许的**(**并被保留**),**除非三个连续的引用字符中断了该串**.(**引用字符是用于引用字符串的字符**, **如**'**和**")

Unless an '`r`' or '`R`' prefix is present, escape sequences in strings are interpreted according to rules similar to those used by Standard C. The recognized escape sequences are:

如果一个'r'**或**'R'**给出**, **那么其含义就像标准**C**中的规则类似地解释**, **承认的转义的字符如下**:

Escape Sequence	Meaning	Notes
`\`*newline*	Ignored	
`\\`	Backslash (`\`)	
`\'`	Single quote (`'`)	
`\"`	Double quote (`"`)	
`\a`	ASCII Bell (BEL)	
`\b`	ASCII Backspace (BS)	
`\f`	ASCII Formfeed (FF)	
`\n`	ASCII Linefeed (LF)	
`\N{`*name*`}`	Character named *name* in the Unicode database (Unicode only)	
`\r`	ASCII Carriage Return (CR)	
`\t`	ASCII Horizontal Tab (TAB)	
`\u`*xxxx*	Character with 16-bit hex value *xxxx* (Unicode only)	(1)
`\U`*xxxxxxxx*	Character with 32-bit hex value *xxxxxxxx* (Unicode only)	(2)
`\v`	ASCII Vertical Tab (VT)	
`\`*ooo*	ASCII character with octal value *ooo*	(3)
`\x`*hh*	ASCII character with hex value *hh*	(4)

Notes:

(1) Individual code units which form parts of a surrogate pair can be encoded using this escape sequence.

(2) Any Unicode character can be encoded this way, but characters outside the Basic Multilingual Plane (BMP) will be encoded using a surrogate pair if Python is compiled to use 16-bit code units (the default). Individual code units which form parts of a surrogate pair can be encoded using this escape sequence.

(3) As in Standard C, up to three octal digits are accepted.

(4) Unlike in Standard C, at most two hex digits are accepted.

Unlike Standard C, all unrecognized escape sequences are left in the string unchanged, i.e., *the backslash is left in the string*. (This behavior is useful when debugging: if an escape sequence is mistyped, the resulting output is more easily recognized as broken.) It is also important to note that the escape sequences marked as "(Unicode only)" in the table above fall into the category of unrecognized escapes for non-Unicode string literals.

不像标准C, **所有不能被解释的转义序列留在串不作改变**, **即反斜线留在串中**(**这个行为在调试中有用**: **如果输入出错**, **这样可以很容易地判断出错**), **也要注意**, **上面仅仅在**Unicode**中才有效的转义序列**,**在非**Unicode**字面值中是无效的**.

When an 'r' or 'R' prefix is present, a character following a backslash is included in the string without change, and *all backslashes are left in the string*. For example, the string literal `r"\n"` consists of two characters: a backslash and a lowercase 'n'. String quotes can be escaped with a backslash, but the backslash remains in the string; for example, `r"\""` is a valid string literal consisting of two characters: a backslash and a double quote; `r"\"` is not a valid string literal (even a raw string cannot end in an odd number of backslashes). Specifically, *a raw string cannot end in a single backslash* (since the backslash would escape the following quote character). Note also that a single backslash followed by a newline is interpreted as those two characters as part of the string, *not* as a line continuation.

当给出'r'**或**'R'**时**, **跟随反斜线后面的字符不被改变**, **并且所有制的反斜线字符都会留在串中**.**例如**,**串**`r"\n"`**由两个字符组成**:**一个反斜线的一个小写的**'n'.**引用字符可以用反斜线引用**, **但反斜线会留在串中**.**比如**`r"\""`**是一个有效的串字面值**(**即使原始串不能以连续的奇数个反斜线结束**). **另外**, **原始不能以一个反斜线结束**(**因为反斜线会把后面的引用字符转义**), **也要注意新行号前的反斜线是解释为串中的两个字符**, **而不是作为续行处理**.

When an 'r' or 'R' prefix is used in conjunction with a 'u' or 'U' prefix, then the `\uXXXX` escape sequence is processed while *all other backslashes are left in the string*. For example, the string literal `ur"\u0062\n"` consists of three Unicode characters: 'LATIN SMALL LETTER B', 'REVERSE SOLIDUS', and 'LATIN SMALL LETTER N'. Backslashes can be escaped with a preceding backslash; however, both remain in the string. As a result, `\uXXXX` escape sequences are only recognized when there are an odd number of backslashes.

2.4.2 串字面值的连接String literal concatenation

Multiple adjacent string literals (delimited by whitespace), possibly using different quoting conventions, are allowed, and their meaning is the same as their concatenation. Thus, `"hello" 'world'` is equivalent to `"helloworld"`. This feature can be used to reduce the number of backslashes needed, to split long strings conveniently across long lines, or even to add comments to parts of strings, for example:

多个相邻的串字面值(**由空白分隔**), **可能使用不同的引用习惯**, **是允许的**, **并且它的含义在连接时是一样的行**.**因此**, "hello""world"**等价于**"helloworld".**这个待征可以用来减少原本要使用的反斜线的数目**, **可以把一个长串分隔在多行上**,**下班甚至在串的某个部分加上注释**, **例如**:

```
re.compile("[A-Za-z_]"       # letter or underscore
           "[A-Za-z0-9_]*"   # letter, digit or underscore
          )
```

Note that this feature is defined at the syntactical level, but implemented at compile time. The '+' operator must be used to concatenate string expressions at run time. Also note that literal concatenation can use different quoting styles for each component (even mixing raw strings and triple quoted strings).

注意这个功能是定义在句法层次上的, **但是是在编译时实现的**.**在运行时连接串必须使用**"+"**运算符**. **并且不同的引用字符可以混用**, **甚至可以将原始串与三重引用串混着用**.

2.4.3 数值型的字面值Numeric literals

There are four types of numeric literals: plain integers, long integers, floating point numbers, and imaginary numbers. There are no complex literals (complex numbers can be formed by adding a real number and an imaginary number).

存在有四种类型的数值型的字面值:**普通整数**,**长整数**, **浮点数和虚数**.**没有复数字面值**(**复数可以以一个实数加上一个虚数的形式给出**)

Note that numeric literals do not include a sign; a phrase like `-1` is actually an expression composed of the unary operator '`-`' and the literal `1`.

注意数值型的字面值不包括符号(**译注**:**正负号**), **像**-1**实际上是个组合了一元运算符**"-"**和字面值**1**的表达式**.

2.4.4 整数和长整数型的字面值Integer and long integer literals

Integer and long integer literals are described by the following lexical definitions:

整数和长整数型字面值可以用以下词法定义:

```
longinteger     ::=  integer ("l" | "L")
integer         ::=  decimalinteger | octinteger | hexinteger
decimalinteger  ::=  nonzerodigit digit* | "0"
octinteger      ::=  "0" octdigit+
hexinteger      ::=  "0" ("x" | "X") hexdigit+
nonzerodigit    ::=  "1"..."9"
octdigit        ::=  "0"..."7"
hexdigit        ::=  digit | "a"..."f" | "A"..."F"
```

Although both lower case '`l`' and upper case '`L`' are allowed as suffix for long integers, it is strongly recommended to always use '`L`', since the letter '`l`' looks too much like the digit '`1`'.

虽然小写的"l"**的大写的**"L"**都可以作为长整数的后缀**, **但强烈火推荐使用**"L", **因为字母**"l"**看来太像数字**1**了**.

Plain integer literals that are above the largest representable plain integer (e.g., 2147483647 when using 32-bit arithmetic) are accepted as if they were long integers instead.[1] There is no limit for long integer literals apart from what can be stored in available memory.

普通十进制整数最大可以为2147483647(**也就是使用**32**位比特数字的最大值**),**普通的八进制和十六进制数可以**4294967295, **但大于**2147483647**的数就通过减**4294967295**变为负数了**.**长整数的大小是没有限制的**, **仅仅受制于可用的内存容量**.

Some examples of plain integer literals (first row) and long integer literals (second and third rows):

下面是一些普通和长整数的例子:

```
7     2147483647                        0177
3L    79228162514264337593543950336L    0377L    0x100000000L
      79228162514264337593543950336              0xdeadbeef
```

2.4.5 浮点型的字面值Floating point literals

Floating point literals are described by the following lexical definitions:

浮点型的字面值可以用以下词法定义描述:

[1]In versions of Python prior to 2.4, octal and hexadecimal literals in the range just above the largest representable plain integer but below the largest unsigned 32-bit number (on a machine using 32-bit arithmetic), 4294967296, were taken as the negative plain integer obtained by subtracting 4294967296 from their unsigned value.

```
floatnumber    ::=  pointfloat | exponentfloat
pointfloat     ::=  [intpart] fraction | intpart "."
exponentfloat  ::=  (intpart | pointfloat) exponent
intpart        ::=  digit+
fraction       ::=  "." digit+
exponent       ::=  ("e" | "E") ["+" | "-"] digit+
```

Note that the integer and exponent parts of floating point numbers can look like octal integers, but are interpreted using radix 10. For example, '077e010' is legal, and denotes the same number as '77e10'. The allowed range of floating point literals is implementation-dependent. Some examples of floating point literals:

注意浮点数的整数部分的指数部分可以看起来像是个八进制数, 但实际上仍作为十进制处理, 例如,"077e010"**是合法的, 它等价于**"77e10".**浮点数的允许范围是依赖于实现的**,**确良以是一些浮点数的例子**:

```
3.14    10.    .001    1e100    3.14e-10    0e0
```

Note that numeric literals do not include a sign; a phrase like -1 is actually an expression composed of the operator - and the literal 1.

注意数值型的字面值不包括符号(**译注**:**正负号**), **像**-1**实际上是个组合了一元运算符**"-"**和字面值**1**的表达式**.

2.4.6 虚数的字面值Imaginary literals

Imaginary literals are described by the following lexical definitions:

虚数的字面值可以用下面的词法定义描述:

```
imagnumber  ::=  (floatnumber | intpart) ("j" | "J")
```

An imaginary literal yields a complex number with a real part of 0.0. Complex numbers are represented as a pair of floating point numbers and have the same restrictions on their range. To create a complex number with a nonzero real part, add a floating point number to it, e.g., (3+4j). Some examples of imaginary literals:

虚数是实部为零的复数, 复数表达成一对有着相同取值范围的浮点数对.**为了创建一个有着非零实部的复数**,**可以对它增加一个浮点数**,**例如**,(3+4j). **下面是一些例子**:

```
3.14j   10.j    10j     .001j   1e100j  3.14e-10j
```

2.5 运算符Operators

The following tokens are operators:

以下符号是运算符:

```
+       -       *       **      /       //      %
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=      <>
```

The comparison operators <> and != are alternate spellings of the same operator. != is the preferred spelling; <> is obsolescent.

比较运算符¡¿**和**!=**是可选的两个具有相同含义的符号**. **但推荐使用**!=, ¡¿**是过时的写法**.

2.6 分隔符Delimiters

The following tokens serve as delimiters in the grammar:

以下符号用作语法上的分隔符:

```
(       )       [       ]       {       }
,       :       .       `       =       ;
+=      -=      *=      /=      //=     %=
&=      |=      ^=      >>=     <<=     **=
```

The period can also occur in floating-point and imaginary literals. A sequence of three periods has a special meaning as an ellipsis in slices. The second half of the list, the augmented assignment operators, serve lexically as delimiters, but also perform an operation.

句号可以在浮点数和虚数字面值中出现, **一个连续的三个句号的序列具有特殊的含义**, **它指出这是一个片断中的省略写法**.**在这个表格的后半部分**, **即参数化赋值运算符**, **在词法上是作为分隔符处理**, **但也执行运算**.

The following printing ASCII characters have special meaning as part of other tokens or are otherwise significant to the lexical analyzer:

以下可显示ASCII**字符具有作为其它符号的一部分有着特殊的含义**, **或者对于词法分析器具有重要意义**:

```
'       "       #       \
```

The following printing ASCII characters are not used in Python. Their occurrence outside string literals and comments is an unconditional error:

以下可显示ASCII**字符不在**Python**中使用**, **当它们出现在注释和字面值之外时就是非法的**:

```
@       $       ?
```

第三章

!!数据模型Data model

3.1 对象, 值和类型Objects, values and types

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer," code is also represented by objects.)

Python中,对象是数据的抽象,所有Python程序中的数据都表述成对象,或与对象有关的形式. (实际上, 和"冯–诺依曼的"存储程序计算机"模型一致, 连代码也是对象)

Every object has an identity, a type and a value. An object's *identity* never changes once it has been created; you may think of it as the object's address in memory. The '`is`' operator compares the identity of two objects; the `id()` function returns an integer representing its identity (currently implemented as its address). An object's *type* is also unchangeable.[1] An object's type determines the operations that the object supports (e.g., "does it have a length?") and also defines the possible values for objects of that type. The `type()` function returns an object's type (which is an object itself). The *value* of some objects can change. Objects whose value can change are said to be *mutable*; objects whose value is unchangeable once they are created are called *immutable*. (The value of an immutable container object that contains a reference to a mutable object can change when the latter's value is changed; however the container is still considered immutable, because the collection of objects it contains cannot be changed. So, immutability is not strictly the same as having an unchangeable value, it is more subtle.) An object's mutability is determined by its type; for instance, numbers, strings and tuples are immutable, while dictionaries and lists are mutable.

每个对象都有一个标识, 一个类型和一个值. 一旦对象建立了, 它的标识就不能改变了; 你可以认为它是内存中的地址."is"运算符可以用来比较两个对象的标识; id()函数可以获得一个整数形式的对象标识(当前是是作为地址实现的). 对象的类型也是不可变的, 它用于检测对象支持的操作(例如,""有长度吗?), 也定义了该种类型的几个值.type()函数返回对象的类型(类型本身也是一个对象). 某些值的对象可以改变, 值可以改变的对象称为是可变的, 一旦建立后其值就不可变的对象称为是不可变的.(属于不可变对象的包容器在包括有可变对象的引用时, 当可变对象改变了时, 其实是可变的, 但仍被看作是可变对象, 因为它所包含的对象集合是不可变的, 所以不可变对象与值不可变是不完全一样的, 它更加微妙)一个对象的可变性由它的类型决定, 例如. 数值, 串和元组是不可变的, 但字典和列表是可变的.

Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected. An implementation is allowed to postpone garbage collection or omit it altogether — it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable. (Implementation note: the current implementation uses a reference-counting scheme with (optional) delayed detection of cyclically linked garbage, which collects most objects as soon as they become unreachable, but is not guaranteed to collect garbage containing circular references. See the *Python Library Reference* for information on controlling the collection of cyclic garbage.)

对象不用显式地释放它们; 但是当它们不可用时它就可能会被回收.实现是允许推迟回收或完全忽略它

[1] Since Python 2.2, a gradual merging of types and classes has been started that makes this and a few other assertions made in this manual not 100% accurate and complete: for example, it *is* now possible in some cases to change an object's type, under certain controlled conditions. Until this manual undergoes extensive revision, it must now be taken as authoritative only regarding "classic classes", that are still the default, for compatibility purposes, in Python 2.2 and 2.3.

的—**这是回收机制如何实现的质量问题**. **只要求尚能访问的对象不被回收**.(**实现注意**:**当前实现使用一个引用计数机制和一个可选的循环垃圾延时检测机制**.**只要对象不可用了**, **就会收回大部分这样的对象**,**但不能保证回收中包含有循环引用**.**对于循环回收的详细控制情况**, **见**Python **库参考**)

Note that the use of the implementation's tracing or debugging facilities may keep objects alive that would normally be collectable. Also note that catching an exception with a '`try`...`except`' statement may keep objects alive.

注意使用实现的跟踪和调试工具时可能会保留本该回收的对象, **也要注意使用语句**"try ... except"**也可能保留本该回收的对象**.

Some objects contain references to "external" resources such as open files or windows. It is understood that these resources are freed when the object is garbage-collected, but since garbage collection is not guaranteed to happen, such objects also provide an explicit way to release the external resource, usually a `close()` method. Programs are strongly recommended to explicitly close such objects. The '`try`...`finally`' statement provides a convenient way to do this.

有些对象包含有"**外部**"**资源**, **像文件或窗口**. **可以相信在垃圾回收时这些资源也被释放**. **但因为垃圾回收不保证一定发生**, **这样的对象提供了显式的方法释放这些资源**. **通常是用**close()**方法**.**高度推荐使用这种方法释放包含外部资源的对象**."try ... finally"**提供了这样的一个途径**.

Some objects contain references to other objects; these are called *containers*. Examples of containers are tuples, lists and dictionaries. The references are part of a container's value. In most cases, when we talk about the value of a container, we imply the values, not the identities of the contained objects; however, when we talk about the mutability of a container, only the identities of the immediately contained objects are implied. So, if an immutable container (like a tuple) contains a reference to a mutable object, its value changes if that mutable object is changed.

有些对象包含了其它对象的引用; **它们叫做容器**, **容器的例子有元组**, **列表和字典**.**引用作为容器值的一部分**.**大多数情况下**, **当我们谈及一个容器的值时**, **我们只是涉及了这个值**, **而不是所包含的对象**; **但是**, **当我们谈及容器对象的可变性的时候**, **就只是涉及被直接包含的对象的标识了**. **因此**,**如果一个不可变对象**(**如元组**)**包含了一个可变对象**, **那么只要这个可变对象的值变了则它的值就也改变了**.

Types affect almost all aspects of object behavior. Even the importance of object identity is affected in some sense: for immutable types, operations that compute new values may actually return a reference to any existing object with the same type and value, while for mutable objects this is not allowed. E.g., after '`a = 1; b = 1`', `a` and `b` may or may not refer to the same object with the value one, depending on the implementation, but after '`c = []; d = []`', `c` and `d` are guaranteed to refer to two different, unique, newly created empty lists. (Note that '`c = d = []`' assigns the same object to both `c` and `d`.)

类型对对象的大多数行为都有影响, **甚至在某种程度上对对象的标识也有重要的影响**:**对于不可变对象**, **计算新值的运算符可能实际上返回的是一个已存在的具有相同类型和值的对象的引用**, **对于可变对象**,**这是不允许的**.**例如**: **在**"a = 1; b = 1"**之后**, a**和**b**有可能指向一个具有值为**1**的对象**, **这依赖于实现**.**但**"c = []; d = []"**之后**,c**和**d**可以保证是保存了两个不同的**, **独立的**, **新创建的空列表**. (**注意**"c = d = []"**是把相同的对象赋给了**c**和**d.)

3.2 标准类型层次The standard type hierarchy

Below is a list of the types that are built into Python. Extension modules (written in C, Java, or other languages, depending on the implementation) can define additional types. Future versions of Python may add types to the type hierarchy (e.g., rational numbers, efficiently stored arrays of integers, etc.).

以下是Python**内建类型的一个列表**, **以**C**语言写的扩展模块可以定义其它类型**. **以后版本的**Python**可能会在这个类型层次中增加其它类型**(**例如**: **有理数**, **高效率存储的整数数组**, **等等**)

Some of the type descriptions below contain a paragraph listing 'special attributes.' These are attributes that provide access to the implementation and are not intended for general use. Their definition may change in the future.

有些类型描述中包括了一个列出"**特殊属性**"**的段落**, **它们提供了一些访问实现的方法而不是作为一般目的使用**.**这些定义可能会在未来改变**.

无None This type has a single value. There is a single object with this value. This object is accessed through the built-in name `None`. It is used to signify the absence of a value in many situations, e.g., it is returned from functions that don't explicitly return anything. Its truth value is false.

这个类型具有单一值, 也只有一个对象有这个值, 这个对象可以通过内建名字None访问它在许多场合表示无值. 例如它在那些没有返回值的函数中返回. 其真值为假.

未实现NotImplemented This type has a single value. There is a single object with this value. This object is accessed through the built-in name `NotImplemented`. Numeric methods and rich comparison methods may return this value if they do not implement the operation for the operands provided. (The interpreter will then try the reflected operation, or some other fallback, depending on the operator.) Its truth value is true.

这个类型具有单一值, 也只有一个对象有这个值, 这个对象可以通过内建名字NotImplemented访问. 如果数值方法和许多比较方法的操作数未提供其实现, 它们就可能返回这个值. (解释器会试图扩展成其它操作，或者其它退化的操作，依赖于运算符)它的真值为真.

省略Ellipsis This type has a single value. There is a single object with this value. This object is accessed through the built-in name `Ellipsis`. It is used to indicate the presence of the '`...`' syntax in a slice. Its truth value is true.

这个类型具有单一值, 也只有一个对象有这个值, 这个对象可以通过内建名字NotImplemented访问. 它用于指出使用在片断上的"..." 句法. 其真值为真.

数值型Numbers These are created by numeric literals and returned as results by arithmetic operators and arithmetic built-in functions. Numeric objects are immutable; once created their value never changes. Python numbers are of course strongly related to mathematical numbers, but subject to the limitations of numerical representation in computers.

它们由数值型字面值生成，或由算术运算和内置算术函数作为值返回。数值型对象是不可变的.一旦创建其值就不可改变. Python的数值型和数学上的数字关系非常密切，但要受到计算机的数值表达能力的限制.

Python distinguishes between integers, floating point numbers, and complex numbers:

Python **对整数，浮点数和复数区别对待：**

整数Integers These represent elements from the mathematical set of whole numbers.

描述了数学上的整数集.

There are three types of integers:

有三种整数类型：

普通整数Plain integers These represent numbers in the range -2147483648 through 2147483647. (The range may be larger on machines with a larger natural word size, but not smaller.) When the result of an operation would fall outside this range, the result is normally returned as a long integer (in some cases, the exception `OverflowError` is raised instead). For the purpose of shift and mask operations, integers are assumed to have a binary, 2's complement notation using 32 or more bits, and hiding no bits from the user (i.e., all 4294967296 different bit patterns correspond to different values).

描述了在-2147483648 至2147483647 范围之内的数. (这个范围可能会在本地机器字较大的机器更大些，但绝不会小.)当某个运算的结果超出了这个范围，异常OverflowError 会被抛出. 对于以移位和屏蔽为目的的运算, 整数被认为是使用32位或更多位的二进制的补码形式. 并且不为用户隐藏任何位.(就是说, 所有位的组合对应于4294967296个不同的值)

长整数Long integers These represent numbers in an unlimited range, subject to available (virtual) memory only. For the purpose of shift and mask operations, a binary representation is assumed, and negative numbers are represented in a variant of 2's complement which gives the illusion of an infinite string of sign bits extending to the left.

长整数的表示范围在语言上是无限制的, 只受限于你的内存(虚拟内存). 对于以移位和屏蔽为目的的运算, 长整数被认为是二进制的形式, 如果是负数,那么就被转换成二进制补码形式, 符号位向左扩展.

Booleans These represent the truth values False and True. The two objects representing the values False and True are the only Boolean objects. The Boolean type is a subtype of plain integers, and Boolean values behave like the values 0 and 1, respectively, in almost all contexts, the exception being that when converted to a string, the strings `"False"` or `"True"` are returned, respectively.

The rules for integer representation are intended to give the most meaningful interpretation of shift and mask operations involving negative integers and the least surprises when switching between the plain and long integer domains. Any operation except left shift, if it yields a result in the plain integer domain without causing overflow, will yield the same result in the long integer domain or when using mixed operands.

整数表示法的这个规则为了使对负数操作时尽量有实际意义,至少也在普通整数和长整数之间转换时所导致的结果不至于太过奇怪. 对于除了左移之外的操作, 只要这些数的计算结果在整数域之中, 没有溢出, 那么它在长整数域或混合运算中也能得到相同结果.

浮点数Floating point numbers These represent machine-level double precision floating point numbers. You are at the mercy of the underlying machine architecture (and C or Java implementation) for the accepted range and handling of overflow. Python does not support single-precision floating point numbers; the savings in processor and memory usage that are usually the reason for using these is dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating point numbers.

本类型表示了机器级的双精度浮点数. 是机器的体系结构和C实现的浮点数范围和控制溢出方法屏蔽你不需知道的细节. Python不支持单精度浮点数, 使用它的原因通常是减少CPU负荷和节省内存, 但是这一节省被对象在Python中的使用开销所抵消, 因此没有必要支持两种浮点数使语言变的复杂.

复数Complex numbers These represent complex numbers as a pair of machine-level double precision floating point numbers. The same caveats apply as for floating point numbers. The real and imaginary parts of a complex number `z` can be retrieved through the read-only attributes `z.real` and `z.imag`.

本类型描述了一对机器级的双精度浮点数, 在浮点数中的警告内容也适用于复数类型. 复数z的实部和虚部可以通过属性z.real和z.imag获得.

有序类型Sequences These represent finite ordered sets indexed by non-negative numbers. The built-in function `len()` returns the number of items of a sequence. When the length of a sequence is *n*, the index set contains the numbers 0, 1, . . . , *n*-1. Item *i* of sequence *a* is selected by *a*`[`*i*`]`.

本类型描述了用非负数索引的有限的有序集合. 内建函数len()返回有序类型数据中的项数当有序类型数据的长为N时, 索引号为0,1, ..., N-1.有序类型数据A中的项I, 可以以A[I]表示.

Sequences also support slicing: *a*`[`*i*`:`*j*`]` selects all items with index *k* such that *i* <= *k* < *j*. When used as an expression, a slice is a sequence of the same type. This implies that the index set is renumbered so that it starts at 0.

有序类型也支持片断:a[i:j]可以表示满足i¡=k¡j的所有项a[k]. 在使用这个表达式时, 它具有相同的有序类型, 这也隐含着索引重新从零开始计.

Some sequences also support "extended slicing" with a third "step" parameter: *a*`[`*i*`:`*j*`:`*k*`]` selects all items of *a* with index *x* where *x* `=` *i* `+` *n*`*`*k*, *n* `>=` `0` and *i* <= *x* < *j*.

Sequences are distinguished according to their mutability:

有序类型按照可变性分两类:

不可变类型Immutable sequences An object of an immutable sequence type cannot change once it is created. (If the object contains references to other objects, these other objects may be mutable and may be changed; however, the collection of objects directly referenced by an immutable object cannot change.)

一个不可变对象一旦建立其值不可能再改变(如果这个对象包括其它对象的引用, 这个被引用的对象可以是可变对象并且其值可以变化;但为该不可变对象所直接引用的对象集合是不能变的.)

The following types are immutable sequences:

以下类型是不可变类型:

字符串Strings The items of a string are characters. There is no separate character type; a character is represented by a string of one item. Characters represent (at least) 8-bit bytes. The built-in functions `chr()` and `ord()` convert between characters and nonnegative integers representing the byte values. Bytes with the values 0-127 usually represent the corresponding ASCII values, but the interpretation of values is up to the program. The string data type is also used to represent arrays of bytes, e.g., to hold data read from a file.

字符串的项是字符. 没有单独的字符类型, 一个字符可以表达成只有一个项的串.字符描述成八比特(至少)字节. 内建函数chr()**和**ord()**在字符和非负整数字节值表示之间切换. 在值**0-127**之间的字符通常描述成**ASCII**值, 但值的解释依赖于应用程序.字符串类型也用于表示字节数组, 例如, 保存从文件读出的内容.**

(On systems whose native character set is not ASCII, strings may use EBCDIC in their internal representation, provided the functions `chr()` and `ord()` implement a mapping between ASCII and EBCDIC, and string comparison preserves the ASCII order. Or perhaps someone can propose a better rule?)

(**在那些本地字符集不是**ASCII**的系统上,字符串可能使用**EBCDIC**作为其内部表示方案. 已提供了函数**chr()**和**ord()**在**ASCII**和**EBCDIC**之间进行互换, 并且串的比较是按照**ASCII**序的, 或者你有更好的主意**?)

Unicode Unicode The items of a Unicode object are Unicode code units. A Unicode code unit is represented by a Unicode object of one item and can hold either a 16-bit or 32-bit value representing a Unicode ordinal (the maximum value for the ordinal is given in `sys.maxunicode`, and depends on how Python is configured at compile time). Surrogate pairs may be present in the Unicode object, and will be reported as two separate items. The built-in functions `unichr()` and `ord()` convert between code units and nonnegative integers representing the Unicode ordinals as defined in the Unicode Standard 3.0. Conversion from and to other encodings are possible through the Unicode method `encode` and the built-in function `unicode()`.

每个Unicode**对象的项是**Unicode**字符, 一个**Unicode**字符表示成一个只有一项的**Unicode**对象, 并且它是一个**16**比特的值可以保存**Unicode**序数. 内建函数**unichr()**和**ord()**在字符和定义在**Unicode 3.0 **标准中非负的**Unicode**序数之间互换.与其它编码集转换是可能的, 可以通过**Unicode**方法**encode()**和内建方法**unicode().

元组Tuples The items of a tuple are arbitrary Python objects. Tuples of two or more items are formed by comma-separated lists of expressions. A tuple of one item (a 'singleton') can be formed by affixing a comma to an expression (an expression by itself does not create a tuple, since parentheses must be usable for grouping of expressions). An empty tuple can be formed by an empty pair of parentheses.

元组对象的项可以是做任意的Python**对象. 具有两个或多个项的元组是用逗号分隔开表达式列表. 只有一项的列表(独元**??)**其项可以后缀一个逗号使其成为满足要求元组要求的表达式**(**一个表达式本身不能创建一个元组,因为圆括号本身用于表达式的分组**).**一个空元组可以以一对空圆括号表示.**

可变对象Mutable sequences Mutable sequences can be changed after they are created. The subscription and slicing notations can be used as the target of assignment and `del` (delete) statements.

可变对象可以在其创建后改变, 其下标表示和片断表示可以作为赋值语句和del**语句的对象**

There is currently a single intrinsic mutable sequence type:

目前只有一种可变类型.

列表Lists The items of a list are arbitrary Python objects. Lists are formed by placing a comma-separated list of expressions in square brackets. (Note that there are no special cases needed to form lists of length 0 or 1.)

列表对象的项可以是任意类型的Python**对象. 列表对象是在方括号之间的用逗号分开的表达式表. (注意，形成长度为**0**或者**1**的链表不要求特别的写法**)

The extension module `array` provides an additional example of a mutable sequence type.

扩展模块array**提供了另一个可变有序类型的例子**

映射类型Mappings These represent finite sets of objects indexed by arbitrary index sets. The subscript notation `a[k]` selects the item indexed by `k` from the mapping `a`; this can be used in expressions and as the target of assignments or `del` statements. The built-in function `len()` returns the number of items in a mapping.

本类型描述了可以是由任意类型作索引的有限对象集合.下标表示法a[k]**从映射类型对象中选择一个由**k**索引的项, 它们可以用作赋值语句和**del**语句的目标. 内建函数**len()**返回映射对象的项的个数.**

There is currently a single intrinsic mapping type:

当前只有一个内置的映射类型:

字典Dictionaries These represent finite sets of objects indexed by nearly arbitrary values. The only types of values not acceptable as keys are values containing lists or dictionaries or other mutable types that are compared by value rather than by object identity, the reason being that the efficient implementation of dictionaries requires a key's hash value to remain constant. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (e.g., `1` and `1.0`) then they can be used interchangeably to index the same dictionary entry.

本类型表达了几乎是任意类型对象都可作索引的有限对象集合.不可接受的键值仅仅是那些包括有列表和字典的值,或那些是通过值比较而不是通过对象标识符比较的类型的值.其原因是字典的高效实现要求键的哈希值保持不变. 用于键值的数值型在比较时使用通常的规则:如果两值相等(如:1**和**1.0), **那么它们可以互换地索引相同的字典项**.

Dictionaries are mutable; they can be created by the `{...}` notation (see section 5.2.5, "Dictionary Displays").

字典是可变的, 它们由...**语法创建**,(**详见**5.2.5, "**字典的显示**").

The extension modules `dbm`, `gdbm`, `bsddb` provide additional examples of mapping types.

扩展模块dbm, gdbm**和**bsddb**提供了另外的映射类型的例子**.

可调用类型Callable types These are the types to which the function call operation (see section 5.3.4, "Calls") can be applied:

这是一个可用的函数调用操作类型, **详见**5.3.4, "**调用**")

用户自定义函数User-defined functions A user-defined function object is created by a function definition (see section 7.5, "Function definitions"). It should be called with an argument list containing the same number of items as the function's formal parameter list.

一个自定义函数对象由函数定义(见7.5, "**函数定义**"). **创建在函数调用时应该与定义时的形式参数相同数目参数**.

Special attributes: `func_doc` or `__doc__` is the function's documentation string, or `None` if unavailable; `func_name` or `__name__` is the function's name; `__module__` is the name of the module the function was defined in, or `None` if unavailable; `func_defaults` is a tuple containing default argument values for those arguments that have defaults, or `None` if no arguments have a default value; `func_code` is the code object representing the compiled function body; `func_globals` is (a reference to) the dictionary that holds the function's global variables — it defines the global namespace of the module in which the function was defined; `func_dict` or `__dict__` contains the namespace supporting arbitrary function attributes; `func_closure` is `None` or a tuple of cells that contain bindings for the function's free variables.

特殊属性:func_doc or __doc__ **是函数的文档串，如果无效，就是**None. func_defaults**是一个包括有参数中设有默认值的元组；**func_name **或**__name__**是函数名．**func_code**是一个编译后的代码对象；**func_globals**是一个引用，指向保存了函数的全局变量的字典**— **如果你在函数所在模块中定义了全局变量的话．**func_dict**或**__dict__**包括了支持任意函数属性的名字空间；**func_closure **要么是**None, **要么是包括有函数自由变量捆绑的单元的元组．＜最后两个属性这里译的可能不对＞**

Of these, `func_code`, `func_defaults`, `func_doc`/`__doc__`, and `func_dict`/`__dict__` may be writable; the others can never be changed. Additional information about a function's definition can be retrieved from its code object; see the description of internal types below.

其中，func_code, func_defaults,func_closure, func_doc**和**func_dict/__dict__**是可以写的;其它则不能改动. 关于函数的定义你可以由它的代码对象得到;见下面的关于内部对象的描述**.

用户自定义方法User-defined methods A user-defined method object combines a class, a class instance (or `None`) and any callable object (normally a user-defined function).

用户自定义方法合并了一个类, **一个类实例(或**None)**和任何可调用对象(一般是用户自定义函数**).

Special read-only attributes: `im_self` is the class instance object, `im_func` is the function object; `im_class` is the class of `im_self` for bound methods or the class that asked for the method for unbound methods; `__doc__` is the method's documentation (same as `im_func.__doc__`); `__name__` is the method name (same as `im_func.__name__`); `__module__` is the name of the module the method was defined in, or `None` if unavailable. Changed in version 2.2: `im_self` used to refer to the class that defined the method.

特殊的只读的属性:im_self**是类实例对象**, im_func**是函数对象**; im_class**是对于捆绑的方法**, **它是**im_self**的类**; **对于自由方法**, **它是其所要求的类**. __doc__**是其方法的文档(与**im_func.__doc__**相同**), __name__**是方法名(与**im_func.__name__**相同**) 2.2**版本中变化**: im_self**用于指出定义方法的类**.

Methods also support accessing (but not setting) the arbitrary function attributes on the underlying function object.

方法也支持访问(不是设置)的其实际所调用的函数的任意属性.

User-defined method objects may be created when getting an attribute of a class (perhaps via an instance of that class), if that attribute is a user-defined function object, an unbound user-defined method object, or a class method object. When the attribute is a user-defined method object, a new method object is only created if the class from which it is being retrieved is the same as, or a derived class of, the class stored in the original method object; otherwise, the original method object is used as it is.

When a user-defined method object is created by retrieving a user-defined function object from a class, its `im_self` attribute is `None` and the method object is said to be unbound. When one is created by retrieving a user-defined function object from a class via one of its instances, its `im_self` attribute is the instance, and the method object is said to be bound. In either case, the new method's `im_class` attribute is the class from which the retrieval takes place, and its `im_func` attribute is the original function object.

用户自定义方法可以用两种方法创建: **在获得是用户自定义函数对象的所属类的属性时**; **或者在获得一个用户自定义函数对象的类实例的属性时**. **在第一个情况(类属性)下**,im_self **属性的值是**None, **并且该方法称为是自由的**. **后一种情况(实例属性)是**, **属性是该实例**, **并且该方法称为是捆绑的**.**例如**, C**是一个具有方法**f() **的类，**C.f() **并不意味着一个的方法对象**f**．但是它指的是自由的方法对象**m, **其中**m.im_class**属性为**C, m.im_func**为**f(), **并且**m.im_self**为**None. **当**x**是**C**的一个实例时，**x.f**指出一个捆绑的方法对象**m, **其**m.im_class**为**C, m_im_func**是**f(), **并且**m.im_self**是**x.

When a user-defined method object is created by retrieving another method object from a class or instance, the behaviour is the same as for a function object, except that the `im_func` attribute of the new instance is not the original method object but its `im_func` attribute.

When a user-defined method object is created by retrieving a class method object from a class or instance, its `im_self` attribute is the class itself (the same as the `im_class` attribute), and its `im_func` attribute is the function object underlying the class method.

When an unbound user-defined method object is called, the underlying function (`im_func`) is called, with the restriction that the first argument must be an instance of the proper class (`im_class`) or of a derived class thereof.

当调用某个自由的用户自定义的方法(im_func)**对象时，有一个限制，其第一个参数必须是适当的类实例**(im_class)**或者是由它继承而来的的类的实例．**

When a bound user-defined method object is called, the underlying function (`im_func`) is called, inserting the class instance (`im_self`) in front of the argument list. For instance, when `C` is a class which contains a definition for a function `f()`, and `x` is an instance of `C`, calling `x.f(1)` is equivalent to calling `C.f(x, 1)`.

当一个捆绑的用户自定义的方法调用时，实际调用的是im_func, **并且在其参数表前部而插入类实例**(im_self)**．例如，当类**C**包括了一个函数定义**f(), **并且**x**是一个**C**的实例**, **调用**x.f(1), **相当于**C.f(x,1)

When a user-defined method object is derived from a class method object, the "class instance" stored in `im_self` will actually be the class itself, so that calling either `x.f(1)` or `C.f(1)` is equivalent to calling `f(C,1)` where `f` is the underlying function.

Note that the transformation from function object to (unbound or bound) method object happens each time the attribute is retrieved from the class or instance. In some cases, a fruitful optimization is to assign the attribute to a local variable and call that local variable. Also notice that this transformation only happens for user-defined functions; other callable objects (and all non-callable objects) are retrieved without transformation. It is also important to note that user-defined functions which are attributes of a class instance are not converted to bound methods; this *only* happens when the function is an attribute of the class.

Note that the transformation from function object to (unbound or bound) method object happens each time the attribute is retrieved from the class or instance. In some cases, a fruitful optimization is to assign the attribute to a local variable and call that local variable. Also notice that this transformation only happens for user-defined functions; other callable objects (and all non-callable objects) are retrieved without transformation. It is also important to note that user-defined functions which are attributes of a class instance are not converted to bound methods; this only happens when the function is an attribute of the class.

生成器函数Generator functions A function or method which uses the `yield` statement (see section 6.8, “The `yield` statement”) is called a *generator function*. Such a function, when called, always returns an iterator object which can be used to execute the body of the function: calling the iterator’s `next()` method will cause the function to execute until it provides a value using the `yield` statement. When the function executes a `return` statement or falls off the end, a `StopIteration` exception is raised and the iterator will have reached the end of the set of values to be returned.

一个使用yield**的语句**(**见**6.8yield **语句**)**的方法或函数**, **它叫做生成器函数**. **这样的函数**, **在返回后**, **通常返回一个迭代子对象**, **一个可以用于执行函数的对象**.**调用对象的**next()**方法会引起函数的执行直到其使用**yield**语句返回一个值**. **当函数在执行到**return**语句时**, **或在末尾结束时**, **会抛出异常**StopIteration, **此时迭代器也到达了返回值集合的结尾。**

内建函数Built-in functions A built-in function object is a wrapper around a C function. Examples of built-in functions are `len()` and `math.sin()` (`math` is a standard built-in module). The number and type of the arguments are determined by the C function. Special read-only attributes: `__doc__` is the function’s documentation string, or `None` if unavailable; `__name__` is the function’s name; `__self__` is set to `None` (but see the next item); `__module__` is the name of the module the function was defined in or `None` if unavailable.

一个内建函数就是一个C**函数的包装**, **例如内建函数**len()**和**math.sin() (math**是标准的内建模块**) **参数的类型和数目都由**C**函数检测**.**特殊的只读属性**: __doc__**是函数的文档串**(**如果有效的话**). __name__**是函数名**, __self__**设为**None(**见下述**).

内建方法Built-in methods This is really a different disguise of a built-in function, this time containing an object passed to the C function as an implicit extra argument. An example of a built-in method is *alist*`.append()`, assuming *alist* is a list object. In this case, the special read-only attribute `__self__` is set to the object denoted by *list*.

这实际上是内建函数的一个不同的包装而已. **此时传递给**C**函数一个对象作为隐含的参数**.**例如**,**内建方法**list.append()**假定**list**是一个列表对象**,**此时特殊只读属性**__self__**指定为对象**list.

类Class Types Class types, or “new-style classes,” are callable. These objects normally act as factories for new instances of themselves, but variations are possible for class types that override `__new__()`. The arguments of the call are passed to `__new__()` and, in the typical case, to `__init__()` to initialize the new instance.

Classic Classes Class objects are described below. When a class object is called, a new class instance (also described below) is created and returned. This implies a call to the class’s `__init__()` method if it has one. Any arguments are passed on to the `__init__()` method. If there is no `__init__()` method, the class must be called without arguments.

类对象下面会说到, **当一个类对象被调用时**,**一个类实例被创建并返回**.**这里隐含着调用**__init__ **方法**(**如果它有的话**), **任何参数都有会被传递到**__init__**中**.**如果类没有**__init__**方法**, **它在调用时**, **就不能指定参数**.

类实例Class instances Class instances are described below. Class instances are callable only when the class has a `__call__()` method; `x(arguments)` is a shorthand for `x.__call__(arguments)`.

类实例将在下面详述, **类实例仅仅在类具有**__call__**方法时**, **才是可调用的**.x(arguments) **只是**x.__call__(arguments)**的缩写形式**.

模块Modules Modules are imported by the `import` statement (see section 6.12, “The `import` statement”). A module object has a namespace implemented by a dictionary object (this is the dictionary referenced by the func_globals attribute of functions defined in the module). Attribute references are translated to lookups in this dictionary, e.g., `m.x` is equivalent to `m.__dict__["x"]`. A module object does not contain the code object used to initialize the module (since it isn’t needed once the initialization is done).

模块可以由import**语句**(**见**import**语句**). **一个模块有一个名字空间**, **用字典对象实现的**(**在模块中定义的函数可以通过**func_globals**属性来访问这个字典**).**属性的访问被翻译成查找这个字典**. **例如**, m.x **等价于**m.__dict__[”x”]. **模块对象并不包含用来初始化该模块的代码对象。**) (**因为一旦初始化完成它就不再需要了**.)

Attribute assignment updates the module’s namespace dictionary, e.g., ‘`m.x = 1`’ is equivalent to ‘`m.__dict__["x"] = 1`’.

属性的赋值更新模块的名字空间, **例如**m.x=1**等价于**m.__dict__[”x”]=1.

Special read-only attribute: `__dict__` is the module’s namespace as a dictionary object.

特殊只读属性:__dict__**是字典形式的模块名字空间**.

Predefined (writable) attributes: `__name__` is the module's name; `__doc__` is the module's documentation string, or `None` if unavailable; `__file__` is the pathname of the file from which the module was loaded, if it was loaded from a file. The `__file__` attribute is not present for C modules that are statically linked into the interpreter; for extension modules loaded dynamically from a shared library, it is the pathname of the shared library file.

预定义的可写属性:__name__**是模块名**;__doc__**是模块的文档串**, **如果无效就为**Nome. **如果模块装载自某文件**, __file__**是被装载模块的文件路径名**, **因为**C**模块不提供些属性**, **该模块已连接至解释器中**, **对于那些从共享库装载的模块**, **这个属性是那些共享库的路径**.

类Classes Class objects are created by class definitions (see section 7.6, "Class definitions"). A class has a namespace implemented by a dictionary object. Class attribute references are translated to lookups in this dictionary, e.g., '`C.x`' is translated to '`C.__dict__["x"]`'. When the attribute name is not found there, the attribute search continues in the base classes. The search is depth-first, left-to-right in the order of occurrence in the base class list.

类对象由类定义创建(**见**7.6 **类定义**).**一个类有一个用字典对象实现的名字空间**. **类属性的访问可以转换成对该字典的查找**.**例如**"C.x"**被解释成**"C.__dic__["x"]".**当属性名未找到时**,**继续在基类中查找**. **查找是深度优先的**,**由左至右的在基类列表中按出现的次序搜索**. **当引用一个类属性时**, **如果其实际是指出的一个用户自定义函数对象**, **它被转换成自由的用户自定义方法对象**(**见上**). **这个方法的**im_class**属性就是被该被初始化的属性引用的所在类**.

When a class attribute reference (for class `C`, say) would yield a user-defined function object or an unbound user-defined method object whose associated class is either `C` or one of its base classes, it is transformed into an unbound user-defined method object whose `im_class` attribute is `C`. When it would yield a class method object, it is transformed into a bound user-defined method object whose `im_class` and `im_self` attributes are both `C`. When it would yield a static method object, it is transformed into the object wrapped by the static method object. See section 3.3.2 for another way in which attributes retrieved from a class may differ from those actually contained in its `__dict__`.

Class attribute assignments update the class's dictionary, never the dictionary of a base class.

类属性的赋值会更新类的字典, **而不是基类的字典**.

A class object can be called (see above) to yield a class instance (see below).

一个类对象可以创建(**见上**), **这样会产生一个类实例**(**下述**).

Special attributes: `__name__` is the class name; `__module__` is the module name in which the class was defined; `__dict__` is the dictionary containing the class's namespace; `__bases__` is a tuple (possibly empty or a singleton) containing the base classes, in the order of their occurrence in the base class list; `__doc__` is the class's documentation string, or None if undefined.

特殊属性: __name__**是类名**, __module_**是类所定义的模块名**; __dict__**是包括类名字空间的字典**. __bases__**是一个元组**(**可能是**None**或独元**), **包括其基类**, **以基类列表中它们的排列次序出现**. __doc__ **是类的文档串**, **如果无效**, **它就是**None.

类实例Class instances A class instance is created by calling a class object (see above). A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes. If a class attribute is found that is a user-defined function object or an unbound user-defined method object whose associated class is the class (call it `C`) of the instance for which the attribute reference was initiated or one of its bases, it is transformed into a bound user-defined method object whose `im_class` attribute is `C` whose `im_self` attribute is the instance. Static method and class method objects are also transformed, as if they had been retrieved from class `C`; see above under "Classes". See section 3.3.2 for another way in which attributes of a class retrieved via its instances may differ from the objects actually stored in the class's `__dict__`. If no class attribute is found, and the object's class has a `__getattr__()` method, that is called to satisfy the lookup.

类实例可以通过调用一个类对象来创建, **类实例可以有一个用字典实现的名字空间**, **它只由搜索属性范围的第一个结果构成**. **如果属性没在那找到**, **并且实例的类有那个名字的属性**, **就继续在类属性中寻找**. **如果找到的是一个用户自定义函数对象**(**而不是其它情况**), **它被转换成自由的用户自定义方法对象**, **这个方法的**im_class**属性是该被初始化的属性引用的所在类**. **如果没有类属性找到**, **并且类有方法**__getattr__(), **那么就调用它来满足查找请求**.

Attribute assignments and deletions update the instance's dictionary, never a class's dictionary. If the class has a `__setattr__()` or `__delattr__()` method, this is called instead of updating the instance dictionary directly.

属性赋值和删除会更新实例目录, **而不是类的字典**. **如果类具有方法**__setattr__()**或**__delattr__(), **则它们会在更新类实例属性调用**, **而不是直接更新**.

Class instances can pretend to be numbers, sequences, or mappings if they have methods with certain special names. See section 3.3, "Special method names."

Class instances can pretend to be numbers, sequences, or mappings if they have methods with certain special names. See section 3.3, "Special method names."

Special attributes: `__dict__` is the attribute dictionary; `__class__` is the instance's class.

特殊属性: __dict__**是属性字典；**__class__**是实例的类．**

文件Files A file object represents an open file. File objects are created by the `open()` built-in function, and also by `os.popen()`, `os.fdopen()`, and the `makefile()` method of socket objects (and perhaps by other functions or methods provided by extension modules). The objects `sys.stdin`, `sys.stdout` and `sys.stderr` are initialized to file objects corresponding to the interpreter's standard input, output and error streams. See the *Python Library Reference* for complete documentation of file objects.

一个文件描述了一个打开的文件．文件由内建函数open()**创建，也可以由方法**os.popen(), os.fdopen() **套接口对象的**makefile()**方法**(**也可能由其它扩展模块的方法或函数）创建．对象**sys.stdin**和**sys.stdout**及**sys.stderr**被相应地初始化成解释器的标准输入流， 标准输出流， 标准错误输出流．关于文件对象的完整文档，详见**Python **库参考**

内部类型Internal types A few types used internally by the interpreter are exposed to the user. Their definitions may change with future versions of the interpreter, but they are mentioned here for completeness.

有少部分由解释器内部使用的类型，暴露给了用户．它们的定义可能会在未来的解释器版本中改变，但都会在这儿提及．

代码对象Code objects Code objects represent *byte-compiled* executable Python code, or *bytecode*. The difference between a code object and a function object is that the function object contains an explicit reference to the function's globals (the module in which it was defined), while a code object contains no context; also the default argument values are stored in the function object, not in the code object (because they represent values calculated at run-time). Unlike function objects, code objects are immutable and contain no references (directly or indirectly) to mutable objects.

代码对象表达了字节编译过的可执行Python**代码，或者叫字节码．代码对象和函数对象的不同在于函数对象包括了一个外在的全局变量引用（其所定义的模块），而代码对象不包括上下文．默认参数值存入函数对象中，而代码对象则不（因为它们的值由运行时计算）．不像函数对象，代码是不可改变的，并且不包括对可变对象的引用．**

Special read-only attributes: `co_name` gives the function name; `co_argcount` is the number of positional arguments (including arguments with default values); `co_nlocals` is the number of local variables used by the function (including arguments); `co_varnames` is a tuple containing the names of the local variables (starting with the argument names); `co_cellvars` is a tuple containing the names of local variables that are referenced by nested functions; `co_freevars` is a tuple containing the names of free variables; `co_code` is a string representing the sequence of bytecode instructions; `co_consts` is a tuple containing the literals used by the bytecode; `co_names` is a tuple containing the names used by the bytecode; `co_filename` is the filename from which the code was compiled; `co_firstlineno` is the first line number of the function; `co_lnotab` is a string encoding the mapping from byte code offsets to line numbers (for details see the source code of the interpreter); `co_stacksize` is the required stack size (including local variables); `co_flags` is an integer encoding a number of flags for the interpreter.

特殊属性:co_name**给出了函数名**; co_argument**是位置参数的数目**(**包括有默认值的参数**); co_nlocals**是函数中局部变量的数目**. co_varnames**是一个包括局部变量名的元组**(**从参数名开始**); co_callvars**是一个元组**, **包括由嵌套函数所引用局部变量名**; co_freevals**包括了既不是局部变量也不是全局变量的**;co_code **包括了编译过的字节码指令序列的字符串**; co_consts**包括字节码中使用的字面值的元组**; co_names**包括字节码中使用的名字的元组**;co_filename**包括着编译过的字节码文件名**; co_firstlineno**是函数首行行号**; co_lnotab**是一个字符串**, **是字节码偏移到行号的映射**(**详见解释器代码**);co_stacksize**是所需要的堆栈尺寸**(**包括局部变量**);co_flags**是一个整数**, **其解释成为许多解释器的标志**.

The following flag bits are defined for `co_flags`: bit `0x04` is set if the function uses the '`*arguments`' syntax to accept an arbitrary number of positional arguments; bit `0x08` is set if the function uses the '`**keywords`' syntax to accept arbitrary keyword arguments; bit `0x20` is set if the function is a generator.

以下标志位由co_flags**定义:如果函数使用**"*参数"**语法来接受任意数目的位置参数就置位**0x04;**如果函数使用**"**keywords"**语法来接受任意数量的关键字参数, 就置位**0x08.**其它位为内部使用, 或留作后用. 如果函数是允许作用域嵌套条件下编译的,则置位**0x10; **如果代码表达的是一个函数,则**co_const**的第一项是该函数的文档串,如果未定义, 它就是**None.

Future feature declarations ('`from __future__ import division`') also use bits in `co_flags` to indicate whether a code object was compiled with a particular feature enabled: bit `0x2000` is set if the function was compiled with future division enabled; bits `0x10` and `0x1000` were used in earlier versions of Python.

Other bits in `co_flags` are reserved for internal use.

If a code object represents a function, the first item in `co_consts` is the documentation string of the function, or `None` if undefined.

堆栈结构对象Frame objects Frame objects represent execution frames. They may occur in traceback objects (see below).

堆栈结构对象描述了可执行结构.它们会在跟踪回溯对象中出现(下述).

Special read-only attributes: `f_back` is to the previous stack frame (towards the caller), or `None` if this is the bottom stack frame; `f_code` is the code object being executed in this frame; `f_locals` is the dictionary used to look up local variables; `f_globals` is used for global variables; `f_builtins` is used for built-in (intrinsic) names; `f_restricted` is a flag indicating whether the function is executing in restricted execution mode; `f_lasti` gives the precise instruction (this is an index into the bytecode string of the code object).

特殊只读属性:f_back**指出前一个堆栈结构(向着调用者的方向), 如果位于堆栈的底部它就是**None; f_code**指出本结构中能执行的代码对象**.f_locals**是用于查找局部变量的字典**;f_globals**用于全局变量**; f_builtin**用于内建名字**;f_restricted**是一个标志, 用于指出代码是否以受限方式执行**. f_lineno**给出行号**,f_lasti**给出精确的指令(是一个代码对象的字符串的索引)**.

Special writable attributes: `f_trace`, if not `None`, is a function called at the start of each source code line (this is used by the debugger); `f_exc_type`, `f_exc_value`, `f_exc_traceback` represent the most recent exception caught in this frame; `f_lineno` is the current line number of the frame — writing to this from within a trace function jumps to the given line (only for the bottom-most frame). A debugger can implement a Jump command (aka Set Next Statement) by writing to f_lineno.

特殊可写属性:f_trace, **如果非空, 就是从每个源代码行的开始处调用的函数(用于调试器)**. f_exc_type, f_exc_value, f_exc_traceback **表述了在这个结构中所捕捉到的的最后的异常**.

跟踪回溯对象Traceback objects Traceback objects represent a stack trace of an exception. A traceback object is created when an exception occurs. When the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered, the stack trace is made available to the program. (See section 7.4, "The `try` statement.") It is accessible as `sys.exc_traceback`, and also as the third item of the tuple returned by `sys.exc_info()`. The latter is the preferred interface, since it works correctly when the program is using multiple threads. When the program contains no suitable handler, the stack trace is written (nicely formatted) to the standard error stream; if the interpreter is interactive, it is also made available to the user as `sys.last_traceback`.

跟踪回溯对象描述一个异常的栈回溯, 跟踪回溯对象在异常发生时创建, 在展开可执行堆栈搜索异常处理器时, 每个展开层的跟踪回溯对象插进当前跟踪回溯对象的前面.在遇到异常处理器时, 跟踪回溯对象也对程序有效了.(见7.4 try **语句) 它可以由**sys.exc_traceback**访问, 也可以由**sys.exc_info()**返回的元组的第三项访问到.后一种是推荐的接口, 因为它也可以使用多线程的程序中工作良好. 当程序中未包括适当的异常处理器, 跟踪回溯对象就被打印到标准错误输出流上.如果工作在交互方式上, 它也可以通过**sys.last_traceback.

Special read-only attributes: `tb_next` is the next level in the stack trace (towards the frame where the exception occurred), or `None` if there is no next level; `tb_frame` points to the execution frame of the current level; `tb_lineno` gives the line number where the exception occurred; `tb_lasti` indicates the precise instruction. The line number and last instruction in the traceback may differ from the line number of its frame object if the exception occurred in a `try` statement with no matching except clause or with a finally clause.

特殊只读属性:tb_text**是堆栈的下一层(向着异常发生的那一层结构), 或者如果没有下一层, 就为**None. tb_frame**指出当前层次的可执行结构**; tb_lineno**给出异常发生的行号**; tb_lasti**指出精确的指令**; **如果异常发生在没有**except**或**finally**子句匹配的**try**语句中的话, 这里的行号和指令可能与结构中的行号和指令不同**.

片断对象Slice objects Slice objects are used to represent slices when *extended slice syntax* is used. This is a slice using two colons, or multiple slices or ellipses separated by commas, e.g., `a[i:j:step]`, `a[i:j, k:l]`, or `a[..., i:j]`. They are also created by the built-in `slice()` function.

片断对象在使用扩展片断句法时描述片断.这是一个使用两个冒号的片断, 或者是用冒号分开的多个片断. 例如, [i:j:step], a[i:j, k:l], **或**a[..., i:j]), **它们也可以由内建函数**slice()**创建**.

Special read-only attributes: `start` is the lower bound; `stop` is the upper bound; `step` is the step value; each is `None` if omitted. These attributes can have any type.

特殊只读属性:start**是下界**, stop**是上界**, step**是步进值**, **如果在片断中忽略了它**, **就是**None.**这些属性可以任意类型**.

Slice objects support one method:

`indices`(*self, length*)

This method takes a single integer argument *length* and computes information about the extended slice that the slice object would describe if applied to a sequence of *length* items. It returns a tuple of three integers; respectively these are the *start* and *stop* indices and the *step* or stride length of the slice. Missing or out-of-bounds indices are handled in a manner consistent with regular slices. New in version 2.3.

Static method objects Static method objects provide a way of defeating the transformation of function objects to method objects described above. A static method object is a wrapper around any other object, usually a user-defined method object. When a static method object is retrieved from a class or a class instance, the object actually returned is the wrapped object, which is not subject to any further transformation. Static method objects are not themselves callable, although the objects they wrap usually are. Static method objects are created by the built-in `staticmethod()` constructor.

Class method objects A class method object, like a static method object, is a wrapper around another object that alters the way in which that object is retrieved from classes and class instances. The behaviour of class method objects upon such retrieval is described above, under "User-defined methods". Class method objects are created by the built-in `classmethod()` constructor.

3.3 特殊方法名Special method names

A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python's approach to *operator overloading*, allowing classes to define their own behavior with respect to language operators. For instance, if a class defines a method named `__getitem__()`, and `x` is an instance of this class, then `x[i]` is equivalent to `x.__getitem__(i)`. Except where mentioned, attempts to execute an operation raise an exception when no appropriate method is defined.

一个类可以实现以特殊句法才调用的某些操作(例如算术运算,下标操作及片断操作), 这是通过以特殊名字定义方法来实现的. 例如, 如果一个类定义了的方法名为__getitem__(), **并且**x**是这个类的实例, 那么**x[i]**就等价于**x.__getitem__(i), (**反过来是不正确的**— **如果**x**是一个列表对象**, x.__getitem__(i)**不等价于**x[i].)**除了所提及的地方**,**试图执行没有适当的方法定义的操作会引起一个异常的抛出**.

When implementing a class that emulates any built-in type, it is important that the emulation only be implemented to the degree that it makes sense for the object being modelled. For example, some sequences may work well with retrieval of individual elements, but extracting a slice may not make sense. (One example of this is the `NodeList` interface in the W3C's Document Object Model.)

当实现一个模拟任何内建类型的类时, 重要的地方在于模拟的程度只要使对象模拟时有效就行了. 例如, 某些有序类型的对象可能在单独提取某引起值时有效, 但在使用片断时是没有意义的.(**这样的一个例子是在**W3C**的文档对象模型中的**NodeList**接口**.)

3.3.1 基本定制Basic customization

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

Called when the instance is created. The arguments are those passed to the class constructor expression. If a base class has an `__init__()` method, the derived class's `__init__()` method, if any, must explicitly call it to ensure proper initialization of the base class part of the instance; for example: '`BaseClass.__init__(`*self*`, [`*args*`...])`'. As a special constraint on constructors, no value may be returned; doing so will cause a `TypeError` to be raised at runtime.

在实例创建时调用, **参数上传递给类型构造器的表达式**. **如果其基类也具有**__init__(), **必须显式地在**__init__()**调用它**, **以保证能够适当地初始化它的基类部分**; **例如**: "BaseClass.__init__(self, [args...])"**作为构造器的特殊情况**, **它没有值被返回**, **如果返回某个值**, **会在运行时抛出异常**TypeError.

`__del__`(*self*)

Called when the instance is about to be destroyed. This is also called a destructor. If a base class has a `__del__()` method, the derived class's `__del__()` method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance. Note that it is possible (though not recommended!) for the `__del__()` method to postpone destruction of the instance by creating a new reference to it. It may then be called at a later time when this new reference is deleted. It is not guaranteed that `__del__()` methods are called for objects that still exist when the interpreter exits.

在实例被删掉时被调用, **也叫作析构器**. **如果其基类也具有**__del__()**方法**, **继承类应该在其**__del__()**显式地调用它**, **以保证适当地删掉它的父类部分**.**注意**, **在**__del__()**通过创建新的引用来推迟删除操作是允许的**, **但这不是推荐的做法**. **它然后在最后这个引用删除时被调用**, **不能保证在解释器退出时**, **仍存在的对象一定会调用**__del__()**方法**.

Note: '`del x`' doesn't directly call `x.__del__()` — the former decrements the reference count for `x` by one, and the latter is only called when `x`'s reference count reaches zero. Some common situations that may prevent the reference count of an object from going to zero include: circular references between objects (e.g., a doubly-linked list or a tree data structure with parent and child pointers); a reference to the object on the stack frame of a function that caught an exception (the traceback stored in `sys.exc_traceback` keeps the stack frame alive); or a reference to the object on the stack frame that raised an unhandled exception in interactive mode (the traceback stored in `sys.last_traceback` keeps the stack frame alive). The first situation can only be remedied by explicitly breaking the cycles; the latter two situations can be resolved by storing `None` in `sys.exc_traceback` or `sys.last_traceback`. Circular references which are garbage are detected when the option cycle detector is enabled (it's on by default), but can only be cleaned up if there are no Python-level `__del__()` methods involved. Refer to the documentation for the `gc` module for more information about how `__del__()` methods are handled by the cycle detector, particularly the description of the `garbage` value.

注意: "del x"**不直接调用**x__del__() —- **前者将引用计数减一**, **而后者仅仅在引用计数减到零时才被调用**.**有一些经常使用的方法**, **可以防止引用计数减为零**: **对象之间的循环引用**(**例如**, **一个双链表或一个具有父结点和子结点指针的树形数据结构**); **对某函数堆栈结构上的引发异常的对象进行引用**(**跟踪回溯对象保存在**sys.exc_traceback**是以保持其有效**);**或者在交互模式下对某函数堆栈上的引发了没有处理器的异常的对象做引用**(**跟踪回溯对象保存在**sys.last_traceback**中以保持其有效**). **第一种方法仅能通过显式地破坏循环才能恢复**,**后两种情况**, **可以通过将**sys.exc_traceback**和**sys.last_traceback**赋给**None**来恢复**. **仅当循环引用检测器选项被允许时**(**这是默认的**)**循环引用才能为垃圾回收机制所发觉**, **但这只在没有相关的**Python**级的**__del__()**方法时才会被清除**.**关于**__del__()**方法怎样处理循环引用的进一步信息参见**gc module, **该处具体地描述了垃圾回收**.

Warning: Due to the precarious circumstances under which `__del__()` methods are invoked, exceptions that occur during their execution are ignored, and a warning is printed to `sys.stderr` instead. Also, when `__del__()` is invoked in response to a module being deleted (e.g., when execution of the program is done), other globals referenced by the `__del__()` method may already have been deleted. For this reason, `__del__()` methods should do the absolute minimum needed to maintain external invariants. Starting with version 1.5, Python guarantees that globals whose name begins with a single underscore are deleted from their module before other globals are deleted; if no other references to such globals exist, this may help in assuring that imported modules are still available at the time when the `__del__()` method is called.

因为调用__del__()**方法的不确定性**, **在它执行时的异常会被忽略**, **而只是在**sys.stderr**打印警告信息**.**另外**, **当某模块被删除**,**相应的**__del__()**方法调用时**(**例如**, **程序退出时**), **有些为**__del_()**方法所引**

用的全局名字可能已经删除了.**由于这个原因**, __del__()**方法应该对其外部要求保持到最小**.Python 1.5**可以保证以单下划线开始的全局名字在其它全局名字被删除之前从该模块中被删除**; **如果没有其它对存在的全局名字的引用**, **这会对确定那些已导入的模块在调用**__del__()**之后有那些还是有效的时是有所帮助的**.

`__repr__`(*self*)

Called by the `repr()` built-in function and by string conversions (reverse quotes) to compute the "official" string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value (given an appropriate environment). If this is not possible, a string of the form '<...*some useful description*...>' should be returned. The return value must be a string object. If a class defines `__repr__()` but not `__str__()`, then `__repr__()` is also used when an "informal" string representation of instances of that class is required.

由repr()**内建函数调用或者在串转换**(**保留引号**)**时用来计算对象的**"**正式说明**"**字符串**, **尽可能的**, **这应该是一个能以相同的值重建一个对象的有效的**Python**表达式**(**在给定的适当有环境下**), **如果这不可能的**,**也应该是返回一个**"¡... **某些有用的信息**...¿"**形式的字符串**. **返回值必须是一个字符串**.

This is typically used for debugging, so it is important that the representation is information-rich and unambiguous.

本函数典型地用法是用于调试, **所以这个串表述成详尽并无歧义是十分重要的**.

`__str__`(*self*)

Called by the `str()` built-in function and by the `print` statement to compute the "informal" string representation of an object. This differs from `__repr__()` in that it does not have to be a valid Python expression: a more convenient or concise representation may be used instead. The return value must be a string object.

由str()**内建函数调用**, **或由**print**语句来计算该对象的**"**非正式**"**串描述**. **这与**__repr__()**是不同的**, **因为它不要求必须为一个有效的**Python**表达式**: **可以采用一个更通俗或更简洁的表述方式**.**返回值必须是一个字符串对象**.

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

New in version 2.1. These are the so-called "rich comparison" methods, and are called for comparison operators in preference to `__cmp__()` below. The correspondence between operator symbols and method names is as follows: *x*<*y* calls *x*`.__lt__(`*y*`)`, *x*<=*y* calls *x*`.__le__(`*y*`)`, *x*==*y* calls *x*`.__eq__(`*y*`)`, *x*`!=`*y* and *x*<>*y* call *x*`.__ne__(`*y*`)`, *x*>*y* calls *x*`.__gt__(`*y*`)`, and *x*>=*y* calls *x*`.__ge__(`*y*`)`. These methods can return any value, but if the comparison operator is used in a Boolean context, the return value should be interpretable as a Boolean value, else a `TypeError` will be raised. By convention, `False` is used for false and `True` for true.

这是在2.1**版本中新增的**, **它们称为**"**厚比较**"**方法集**,**并且比下述的**__cmp__()**方法具有更高的优先级**.**具体的方法名和相应的运算符的对应关系如下**:x¡y**调用**x.__lt__(y); x¡=y**调用**x.__le__(y);x==y**调用**x.__eq__(y);x!=y**和**x¡¿y**调用**x.__ne__(y); x¿y**调用**x.__gt__(y); x¿=y**调用**x.__ge__(y).**这些方法可以返回任何值**, **但是如果比较运算符用于布尔上下文**, **返回值应该被解释成布尔值**, **否则将抛出**TypeError**异常**, **通常**0**表示假**, 1**用于表示真**.

There are no implied relationships among the comparison operators. The truth of *x*==*y* does not imply that *x*`!=`*y* is false. Accordingly, when defining `__eq__`, one should also define `__ne__` so that the operators will behave as expected.

对于这些方法是没有反函数版本(**互换参数**)**的**(**在左边参数不支持该操作**, **但右面的参数支持时使用**).**虽然**, __lt__()**和**__gt__(),__le__()**和**__ge__(), __eq__()**和**__ne__()**看起来是反函数**.

There are no reflected (swapped-argument) versions of these methods (to be used when the left argument does not support the operation but the right argument does); rather, `__lt__()` and `__gt__()` are each other's reflection, `__le__()` and `__ge__()` are each other's reflection, and `__eq__()` and `__ne__()` are their own reflection.

Arguments to rich comparison methods are never coerced. A rich comparison method may return `NotImplemented` if it does not implement the operation for a given pair of arguments.

厚比较方法的参数并不是非要有.如果它对给定的参数对没有实现操作, 一个厚比较方法可以返回NotImplemented.

`__cmp__`(*self, other*)
Called by comparison operations if rich comparison (see above) is not defined. Should return a negative integer if `self < other`, zero if `self == other`, a positive integer if `self > other`. If no `__cmp__()`, `__eq__()` or `__ne__()` operation is defined, class instances are compared by object identity ("address"). See also the description of `__hash__()` for some important notes on creating objects which support custom comparison operations and are usable as dictionary keys. (Note: the restriction that exceptions are not propagated by `__cmp__()` has been removed since Python 1.5.)

如果厚比较操作(见上)未定义, 就调用本函数. 如果self¡other小于应该返回负数, 如果self¿other应该返回正数, 如果self == other返回, 应该返回零. 如果没有定义__cmp__(), 但定义了__eq__()或__ne__(), 则类的实例可以通过对象的标识(”地址”)比较. 对于创建支持定制比较操作的对象及可用的字典键的要点详见__hash__的描述(注意: 不能通过__cmp__()传播(propagated)的限制, 已经在Python1.5去掉了).

`__rcmp__`(*self, other*)
Changed in version 2.1: No longer supported.

版本2.1中的变化:不再支持.

`__hash__`(*self*)
Called for the key object for dictionary operations, and by the built-in function `hash()`. Should return a 32-bit integer usable as a hash value for dictionary operations. The only required property is that objects which compare equal have the same hash value; it is advised to somehow mix together (e.g., using exclusive or) the hash values for the components of the object that also play a part in comparison of objects. If a class does not define a `__cmp__()` method it should not define a `__hash__()` operation either; if it defines `__cmp__()` or `__eq__()` but not `__hash__()`, its instances will not be usable as dictionary keys. If a class defines mutable objects and implements a `__cmp__()` or `__eq__()` method, it should not implement `__hash__()`, since the dictionary implementation requires that a key's hash value is immutable (if the object's hash value changes, it will be in the wrong hash bucket).

为键对象的字典操作而调用, 并且通过内建函数hash()实现.应该返回一个可以在字典操作中可用的32位整数值.仅有一个要求, 具有相同的值的对象应该有相同的散列值. 应该考虑以某种方式将散列值与在对象中比较中起作用的部分联系起来(例如, 用排斥或). 如果类没有定义__cmp__()方法, 它也不应该定义__hash__()操作;如果类定义了__cmp__()或__eq__()但没有__hash__(), 它的实例就不能作为散列表的键使用.如果类定义的是可变对象并且实现了__cmp__()或__eq__()方法, 它就不应该实现__hash__(), 因为字典实现一个散列表的键值是不可变的(如果对象的散列值改变了, 它会放在错误的散列表位置中).

`__nonzero__`(*self*)
Called to implement truth value testing, and the built-in operation `bool()`; should return `False` or `True`, or their integer equivalents `0` or `1`. When this method is not defined, `__len__()` is called, if it is defined (see below). If a class defines neither `__len__()` nor `__nonzero__()`, all its instances are considered true.

在真值测试时调用; 应该返回0或者1.当此方法未定义时, __len__()就会被调用.如果类没定义__len__,没有定义__nonzero__(), 那么它就被看作为真.

`__unicode__`(*self*)
Called to implement `unicode()` builtin; should return a Unicode object. When this method is not defined, string conversion is attempted, and the result of string conversion is converted to Unicode using the system default encoding.

3.3.2 定制属性访问Customizing attribute access

The following methods can be defined to customize the meaning of attribute access (use of, assignment to, or deletion of `x.name`) for class instances.

可以定义以下方法用于定制类实例属性的访问的含义(用于赋值, 或删除x.name)

`__getattr__`(*self, name*)
Called when an attribute lookup has not found the attribute in the usual places (i.e. it is not an instance

attribute nor is it found in the class tree for `self`). `name` is the attribute name. This method should return the (computed) attribute value or raise an `AttributeError` exception.

当以正常的方式的访问属性(就是说,要访问的属性既不是实例的属性,也在其所在的类树结构中找不到).name**是属性名**. **方法应该返回一个属性值,或抛出一个**AttributeError**异常**.

Note that if the attribute is found through the normal mechanism, `__getattr__()` is not called. (This is an intentional asymmetry between `__getattr__()` and `__setattr__()`.) This is done both for efficiency reasons and because otherwise `__setattr__()` would have no way to access other attributes of the instance. Note that at least for instance variables, you can fake total control by not inserting any values in the instance attribute dictionary (but instead inserting them in another object). See the `__getattribute__()` method below for a way to actually get total control in new-style classes.

注意如果属性可以通过正常的机制访问, __getattr__()**不会被调用(是故意将**__getattr__()**和**_setattr__()**设置成不对称的)这样的原因是由于效率并且**__setattr__()**不能访问实例的其它属性.注意,至少对于实例变量来说, 你可能通过不往实例字典里插入任何值来伪装所有控制(但将它们插入到其它对象中)**.

`__setattr__`(*self, name, value*)

Called when an attribute assignment is attempted. This is called instead of the normal mechanism (i.e. store the value in the instance dictionary). *name* is the attribute name, *value* is the value to be assigned to it.

在属性将被赋值时调用. **这是作为正常机制的代替使用的(就是地实例字典中存储值)**.name**是实例值**,vaule**是要赋的值**.

If `__setattr__()` wants to assign to an instance attribute, it should not simply execute '`self.`*name* `= value`' — this would cause a recursive call to itself. Instead, it should insert the value in the dictionary of instance attributes, e.g., '`self.__dict__[`*name*`] = value`'. For new-style classes, rather than accessing the instance dictionary, it should call the base class method with the same name, for example, '`object.__setattr__(self, name, value)`'.

如果在__setattr__()**内部试图为一个实例属性赋值**, **不能简单地执行**"self.name = vaule",**这会引起对自身的递归调用**. **而应该这样**, **直接在实例属性字典中插入值**,**如**:"self.__dict__[name] = vaule".

`__delattr__`(*self, name*)

Like `__setattr__()` but for attribute deletion instead of assignment. This should only be implemented if '`del obj.`*name*' is meaningful for the object.

就像__setattr__()**一样**, **不过其作用是删除而不是赋值**.**仅仅对于对象用**"del obj.name"**实现才有意义**.

More attribute access for new-style classes

The following methods only apply to new-style classes.

`__getattribute__`(*self, name*)

Called unconditionally to implement attribute accesses for instances of the class. If the class also defines `__getattr__`, it will never be called (unless called explicitly). This method should return the (computed) attribute value or raise an `AttributeError` exception. In order to avoid infinite recursion in this method, its implementation should always call the base class method with the same name to access any attributes it needs, for example, '`object.__getattribute__(self, name)`'.

Implementing Descriptors

The following methods only apply when an instance of the class containing the method (a so-called *descriptor* class) appears in the class dictionary of another new-style class, known as the *owner* class. In the examples below, "the attribute" refers to the attribute whose name is the key of the property in the owner class' `__dict__`.

`__get__`(*self, instance, owner*)

Called to get the attribute of the owner class (class attribute access) or of an instance of that class (instance attribute acces). *owner* is always the owner class, while *instance* is the instance that the attribute was accessed through, or `None` when the attribute is accessed through the *owner*. This method should return the (computed) attribute value or raise an `AttributeError` exception.

`__set__`(*self, instance, value*)
Called to set the attribute on an instance *instance* of the owner class to a new value, *value*.

`__delete__`(*self, instance*)
Called to delete the attribute on an instance *instance* of the owner class.

Invoking Descriptors

In general, a descriptor is an object attribute with "binding behavior", one whose attribute access has been overridden by methods in the descriptor protocol: `__get__()`, `__set__()`, and `__delete__()`. If any of those methods are defined for an object, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, `a.x` has a lookup chain starting with `a.__dict__['x']`, then `type(a).__dict__['x']`, and continuing through the base classes of `type(a)` excluding metaclasses.

However, if the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined and how they were called. Note that descriptors are only invoked for new style objects or classes (ones that subclass `object()` or `type()`).

The starting point for descriptor invocation is a binding, `a.x`. How the arguments are assembled depends on `a`:

Direct Call The simplest and least common call is when user code directly invokes a descriptor method: `x.__get__(a)`.

Instance Binding If binding to a new-style object instance, `a.x` is transformed into the call: `type(a).__dict__['x'].__get__(a, type(a))`.

Class Binding If binding to a new-style class, `A.x` is transformed into the call: `A.__dict__['x'].__get__(None, A)`.

Super Binding If `a` is an instance of `super`, then the binding `super(B, obj).m()` searches `obj.__class__.__mro__` for the base class `A` immediately preceding `B` and then invokes the descriptor with the call: `A.__dict__['m'].__get__(obj, A)`.

For instance bindings, the precedence of descriptor invocation depends on the which descriptor methods are defined. Data descriptors define both `__get__()` and `__set__()`. Non-data descriptors have just the `__get__()` method. Data descriptors always override a redefinition in an instance dictionary. In contrast, non-data descriptors can be overridden by instances.

Python methods (including `staticmethod()` and `classmethod()`) are implemented as non-data descriptors. Accordingly, instances can redefine and override methods. This allows individual instances to acquire behaviors that differ from other instances of the same class.

The `property()` function is implemented as a data descriptor. Accordingly, instances cannot override the behavior of a property.

__slots__

By default, instances of both old and new-style classes have a dictionary for attribute storage. This wastes space for objects having very few instance variables. The space consumption can become acute when creating large numbers of instances.

The default can be overridden by defining *__slots__* in a new-style class definition. The *__slots__* declaration takes a sequence of instance variables and reserves just enough space in each instance to hold a value for each variable. Space is saved because *__dict__* is not created for each instance.

`__slots__`
This class variable can be assigned a string, iterable, or sequence of strings with variable names used by instances. If defined in a new-style class, *__slots__* reserves space for the declared variables and prevents the automatic creation of *__dict__* and *__weakref__* for each instance. New in version 2.2.

Notes on using *__slots__*

- Without a *__dict__* variable, instances cannot be assigned new variables not listed in the *__slots__* definition. Attempts to assign to an unlisted variable name raises `AttributeError`. If dynamic assignment of new variables is desired, then add `'__dict__'` to the sequence of strings in the *__slots__* declaration. Changed in version 2.3: Previously, adding `'__dict__'` to the *__slots__* declaration would not enable the assignment of new attributes not specifically listed in the sequence of instance variable names.
- Without a *__weakref__* variable for each instance, classes defining *__slots__* do not support weak references to its instances. If weak reference support is needed, then add `'__weakref__'` to the sequence of strings in the *__slots__* declaration. Changed in version 2.3: Previously, adding `'__weakref__'` to the *__slots__* declaration would not enable support for weak references.
- *__slots__* are implemented at the class level by creating descriptors (3.3.2) for each variable name. As a result, class attributes cannot be used to set default values for instance variables defined by *__slots__*; otherwise, the class attribute would overwrite the descriptor assignment.
- If a class defines a slot also defined in a base class, the instance variable defined by the base class slot is inaccessible (except by retrieving its descriptor directly from the base class). This renders the meaning of the program undefined. In the future, a check may be added to prevent this.
- The action of a *__slots__* declaration is limited to the class where it is defined. As a result, subclasses will have a *__dict__* unless they also define *__slots__*.
- *__slots__* do not work for classes derived from "variable-length" built-in types such as `long`, `str` and `tuple`.
- Any non-string iterable may be assigned to *__slots__*. Mappings may also be used; however, in the future, special meaning may be assigned to the values corresponding to each key.

3.3.3 Customizing class creation

By default, new-style classes are constructed using `type()`. A class definition is read into a separate namespace and the value of class name is bound to the result of `type(name, bases, dict)`.

When the class definition is read, if *__metaclass__* is defined then the callable assigned to it will be called instead of `type()`. The allows classes or functions to be written which monitor or alter the class creation process:

- Modifying the class dictionary prior to the class being created.
- Returning an instance of another class – essentially performing the role of a factory function.

`__metaclass__`
    This variable can be any callable accepting arguments for `name`, `bases`, and `dict`. Upon class creation, the callable is used instead of the built-in `type()`. New in version 2.2.

The appropriate metaclass is determined by the following precedence rules:

- If `dict['__metaclass__']` exists, it is used.
- Otherwise, if there is at least one base class, its metaclass is used (this looks for a *__class__* attribute first and if not found, uses its type).
- Otherwise, if a global variable named __metaclass__ exists, it is used.
- Otherwise, the old-style, classic metaclass (types.ClassType) is used.

The potential uses for metaclasses are boundless. Some ideas that have been explored including logging, interface checking, automatic delegation, automatic property creation, proxies, frameworks, and automatic resource locking/synchronization.

3.3.4 模拟可调用对象Emulating callable objects

`__call__`(*self*[, *args...*])
Called when the instance is "called" as a function; if this method is defined, *x*`(arg1, arg2, ...)` is a shorthand for *x*`.__call__(arg1, arg2, ...)`.

当实例像一个函数使用时调用本方法.**如果定义了这个方法**, **那么**x(arg1, arg2, ...)**是**x.__call__(arg1, arg2, ...)**的缩写形式**.

3.3.5 模拟包容器类型Emulating container types

The following methods can be defined to implement container objects. Containers usually are sequences (such as lists or tuples) or mappings (like dictionaries), but can represent other containers as well. The first set of methods is used either to emulate a sequence or to emulate a mapping; the difference is that for a sequence, the allowable keys should be the integers *k* for which `0 <= ` *k* ` < ` *N* where *N* is the length of the sequence, or slice objects, which define a range of items. (For backwards compatibility, the method `__getslice__()` (see below) can also be defined to handle simple, but not extended slices.) It is also recommended that mappings provide the methods `keys()`, `values()`, `items()`, `has_key()`, `get()`, `clear()`, `setdefault()`, `iterkeys()`, `itervalues()`, `iteritems()`, `pop()`, `popitem()`, `copy()`, and `update()` behaving similar to those for Python's standard dictionary objects. The `UserDict` module provides a `DictMixin` class to help create those methods from a base set of `__getitem__()`, `__setitem__()`, `__delitem__()`, and `keys()`. Mutable sequences should provide methods `append()`, `count()`, `index()`, `extend()`, `insert()`, `pop()`, `remove()`, `reverse()` and `sort()`, like Python standard list objects. Finally, sequence types should implement addition (meaning concatenation) and multiplication (meaning repetition) by defining the methods `__add__()`, `__radd__()`, `__iadd__()`, `__mul__()`, `__rmul__()` and `__imul__()` described below; they should not define `__coerce__()` or other numerical operators. It is recommended that both mappings and sequences implement the `__contains__()` method to allow efficient use of the `in` operator; for mappings, `in` should be equivalent of `has_key()`; for sequences, it should search through the values. It is further recommended that both mappings and sequences implement the `__iter__()` method to allow efficient iteration through the container; for mappings, `__iter__()` should be the same as `iterkeys()`; for sequences, it should iterate through the values.

定义以下方法可以实现包容器对象.**包容器通常指有序类型**(**像列表或元组**)**或映射**(**像字典**), **但也可以表示其它包容器**. **第一个方法集用于模拟有序类型或映射**; **有序类型的区别就在于**, **允许键可以是整数**k, **其中**0 ¡= K ¡ N, N**是有序类型的长度**, **或者是描述了一定范围的片断**(**出于向后兼容性的考虑**, __getslice__()**方法可能用于控制简单的片断**, **但不能用于扩展的片断句法**). **在实现映射时**, **推荐提供**keys(), values(), items(), has_key(), get(), clear(), copy(), **和**update(), **使其行为类似于**Python**标准的字典对象**; **可变的有序类型应该提供方法**append(), count(), index(), insert(), pop(), remove(), reverse() **和**sort(),**就像**Python**标准的列表类型**. **最后**, **有序类型应该通过定义下述的方法**__add__(), __radd__(), __iadd__(), __mul__(), __rmul__() **和**__imul__()**实现加法运算**(**就是指连接**)**和乘法运算(指重复).它们不应该定义**__coerce__() **或其它数值运算操作.对于有序类型和字典都推荐实现**__contains__()**, 以便于高效的使用**in**运算符.对于它应该等价于**has_key()**方法,对于有序类型,通过值进行搜索.**

`__len__`(*self*)
Called to implement the built-in function `len()`. Should return the length of the object, an integer >= 0. Also, an object that doesn't define a `__nonzero__()` method and whose `__len__()` method returns zero is considered to be false in a Boolean context.

实现内建函数len()**相仿的功能, 应该返回对象的长度,并且返回一个大于等于**0**的整数, 另外,一个没有定义**__nonzero__()**的方法返回**0**被认为是返回一个逻辑假值.**

`__getitem__`(*self, key*)
Called to implement evaluation of *self*`[`*key*`]`. For sequence types, the accepted keys should be integers and slice objects. Note that the special interpretation of negative indexes (if the class wishes to emulate a sequence type) is up to the `__getitem__()` method. If *key* is of an inappropriate type, `TypeError` may be raised; if of a value outside the set of indexes for the sequence (after any special interpretation of negative values), `IndexError` should be raised. **Note:** `for` loops expect that an `IndexError` will be raised for illegal indexes to allow proper detection of the end of the sequence.

实现self[key]**相仿的功能.对于有序类型,可接受的键包括整数和片断对象.注意对负数索引(如果类希望模拟有序类型)的特殊解释也依赖于**__getitem__()**方法.如果键是不合适的类型,一**

个TypeError异常就会被抛出，如果某个值在有序类型的索引值集合之外（在任何负值索引的特定解释也不能行的通的情况下），会抛出一个IndexError的异常．注意：for循环可以通过对由于对无效索引值而抛出的IndexError异常进行捕获来对访问有序类型的结尾做适当地检测．

`__setitem__`(*self, key, value*)
: Called to implement assignment to *self*`[`*key*`]`. Same note as for `__getitem__()`. This should only be implemented for mappings if the objects support changes to the values for keys, or if new keys can be added, or for sequences if elements can be replaced. The same exceptions should be raised for improper *key* values as for the `__getitem__()` method.

  在对self[key]进行赋值时调用．与__getitem__()有着相同的注意事项．通常只对映射实现本方法，并且要求对象支持改变键所对应的值，或支持增加新键；也可以在有序类型中实现，此时支持单元可以替换．在使用无效的键值时，会抛出与__getitem__()相同的异常．

`__delitem__`(*self, key*)
: Called to implement deletion of *self*`[`*key*`]`. Same note as for `__getitem__()`. This should only be implemented for mappings if the objects support removal of keys, or for sequences if elements can be removed from the sequence. The same exceptions should be raised for improper *key* values as for the `__getitem__()` method.

  在删除self[key]时调用，与__getitem__()有着相同的对象．本方法通常仅仅在映射中实现，并且对象支持键的删除；也可以在有序类型中实现，此时单元可以从有序类型删除．在使用无效的键值时，会抛出与__getitem__()相同的异常．

`__iter__`(*self*)
: This method is called when an iterator is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container, and should also be made available as the method `iterkeys()`.

  要求使用包容器的迭代子时，这个方法被调用．本方法应该返回一个可以迭代包容器所有对象的迭代子对象．对于映射，应该在键的基础上进行迭代，并且也应该像方法iterkeys()一样有效．

  Iterator objects also need to implement this method; they are required to return themselves. For more information on iterator objects, see "Iterator Types" in the *Python Library Reference*.

  迭代子对象也需要实现这个方法，它们应该返回它自己．对于更多的关于迭代子对象的信息，参见Python 库参考　中的　Iterator Types．

The membership test operators (`in` and `not in`) are normally implemented as an iteration through a sequence. However, container objects can supply the following special method with a more efficient implementation, which also does not require the object be a sequence.

成员测试运算符（in和not in）一般通过对有序类型的迭代来实现．但是包容器也可以提供以下方法得到更有效的实现，不要对象是有序类型．

`__contains__`(*self, item*)
: Called to implement membership test operators. Should return true if *item* is in *self*, false otherwise. For mapping objects, this should consider the keys of the mapping rather than the values or the key-item pairs.

  使用成员测试运算符时调用．如果item在self中，　返回true; 否则返回false. 对于映射对象，比较应该在键上进行，不应该是键值对．

3.3.6 模拟有序类型的其它方法Additional methods for emulation of sequence types

The following optional methods can be defined to further emulate sequence objects. Immutable sequences methods should at most only define `__getslice__()`; mutable sequences might define all three methods.

定义以下方法可以进一步模拟有序类型对象; 不可变的有序类型应该只定义方法__getslice__(); 可变类型应该以下所有的方法.

`__getslice__`(*self, i, j*)
: **Deprecated since release 2.0.** Support slice objects as parameters to the `__getitem__()` method.

  从2.0版本开始, 这个函数不再推荐使用. 现在可以使用方法__getitem__()来实现它的功能.

  Called to implement evaluation of *self*`[`*i*`:`*j*`]`. The returned object should be of the same type as *self*. Note that missing *i* or *j* in the slice expression are replaced by zero or `sys.maxint`, respectively. If negative indexes are used in the slice, the length of the sequence is added to that index. If the instance does not implement the `__len__()` method, an `AttributeError` is raised. No guarantee is made that indexes adjusted this way are not still negative. Indexes which are greater than the length of the sequence are not modified. If no `__getslice__()` is found, a slice object is created instead, and passed to `__getitem__()` instead.

在对self[i:j]**求值时调用本方法**. **返回的对象应该与**self**的类型相同**. **注意如果片断中缺少**i**或**j, **它们就是分别替换成**0**或**sys.maxint. **如果在片断中使用了负数**, **运算时就是它加上有序类型对象的长度**. **如果实例未实现**__len__(), **则抛出**AttributeError**异常**. **不能保证这样调整后的索引就是正数**. **对于大于有序类型对象长度的索引是不做修改的**. **如果没有找到**__getslice__(), **就创建一个片断对象**, **然后传递给**__getitem__().

`__setslice__`(*self, i, j, sequence*)

Called to implement assignment to *self*[*i*:*j*]. Same notes for *i* and *j* as for `__getslice__()`.

在对self[i:j]**赋值时调用**, **在**__getslice__()**中对于**i**和**j**的注意也适用于本方法**.

This method is deprecated. If no `__setslice__()` is found, or for extended slicing of the form *self*[*i*:*j*:*k*], a slice object is created, and passed to `__setitem__()`, instead of `__setslice__()` being called.

本方法已经过时了. **如果没有找到**__setslice__(), **就创建一个片断对象**, **然后传递给**__setitem__().

`__delslice__`(*self, i, j*)

Called to implement deletion of *self*[*i*:*j*]. Same notes for *i* and *j* as for `__getslice__()`. This method is deprecated. If no `__delslice__()` is found, or for extended slicing of the form *self*[*i*:*j*:*k*], a slice object is created, and passed to `__delitem__()`, instead of `__delslice__()` being called.

在删除self[i:j]**时调用本方法**.**在**__getslice__()**中对**i**和**j**的注意也适用于本方法**.**本方法已经过时了**. **如果没有找到**__delslice__(), **就创建一个片断对象**, **然后传递给**__delitem__().

Notice that these methods are only invoked when a single slice with a single colon is used, and the slice method is available. For slice operations involving extended slice notation, or in absence of the slice methods, `__getitem__()`, `__setitem__()` or `__delitem__()` is called with a slice object as argument.

注意这些方法只在使用单冒号的片断语法时调用.**对于扩展的片断记法或者缺少这些片断方法的情况下**, **就调用**__getitem__(), __setitem__() **或**__delitem__() , **以一个片断对象为参数**,

The following example demonstrate how to make your program or module compatible with earlier versions of Python (assuming that methods `__getitem__()`, `__setitem__()` and `__delitem__()` support slice objects as arguments):

以下程序对如何你的程序兼容以前版本的Python**做了一个示范**(**假定**__getitem__(), __setitem__() **和**__delitem__()**支持以片断对参数**.)

```
class MyClass:
    ...
    def __getitem__(self, index):
        ...
    def __setitem__(self, index, value):
        ...
    def __delitem__(self, index):
        ...

    if sys.version_info < (2, 0):
        # They won't be defined if version is at least 2.0 final

        def __getslice__(self, i, j):
            return self[max(0, i):max(0, j):]
        def __setslice__(self, i, j, seq):
            self[max(0, i):max(0, j):] = seq
        def __delslice__(self, i, j):
            del self[max(0, i):max(0, j):]
    ...
```

Note the calls to `max()`; these are necessary because of the handling of negative indices before the `__*slice__()` methods are called. When negative indexes are used, the `__*item__()` methods receive them as provided, but the `__*slice__()` methods get a "cooked" form of the index values. For each negative index value, the length of the sequence is added to the index before calling the method (which may still result in a negative index); this is the customary handling of negative indexes by the built-in sequence types, and the `__*item__()` methods are expected to do this as well. However, since they should already be doing that, negative indexes cannot be passed in; they must be constrained to the bounds of the sequence before being passed to the `__*item__()` methods. Calling `max(0, i)` conveniently returns the proper value.

注意调用max()**的代码**; **它是必要的**, **因为在调用**__*slice__()**之前要对负数片断进行控制**.**在使用负数索引后**,**方法**__*item__()**会获得和在提供该参数时的形式相同的数据**. **但**__*slice__()**方法会得到一个包装过的索引值参数**.**对于每个负数索引值**, **在调用这个方法前会加上该有序类型的长度**(**结果仍然可能是负数**). **这个特征可以用来对内建有序类型的负数索引进行定制处理**, **并且**__*item__()**系列方法是可以用于完成这个工作**.**但是由于负数索引应该已经被处理过了**, **因此负数索引是不可能传进来的**; **它们应该在传递进**__*item__()**方法之前就被限制在有该有序类型的长度范围之内**, **通常用调用**max(0,i)**返回适当的值**.

3.3.7 模拟数值类型Emulating numeric types

The following methods can be defined to emulate numeric objects. Methods corresponding to operations that are not supported by the particular kind of number implemented (e.g., bitwise operations for non-integral numbers) should be left undefined.

以下方法用于模拟数值类型.**其中**,**对于有些种类数值所不支持的操作对应的方法未定义**(**如**, **对非整数值的位运算**).

`__`**`add`**`__`(*self, other*)
`__`**`sub`**`__`(*self, other*)
`__`**`mul`**`__`(*self, other*)
`__`**`floordiv`**`__`(*self, other*)
`__`**`mod`**`__`(*self, other*)
`__`**`divmod`**`__`(*self, other*)
`__`**`pow`**`__`(*self, other*[*, modulo*])
`__`**`lshift`**`__`(*self, other*)
`__`**`rshift`**`__`(*self, other*)
`__`**`and`**`__`(*self, other*)
`__`**`xor`**`__`(*self, other*)
`__`**`or`**`__`(*self, other*)

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`). For instance, to evaluate the expression *x*+*y*, where *x* is an instance of a class that has an `__add__()` method, `x.__add__(y)` is called. The `__divmod__()` method should be the equivalent to using `__floordiv__()` and `__mod__()`; it should not be related to `__truediv__()` (described below). Note that `__pow__()` should be defined to accept an optional third argument if the ternary version of the built-in `pow()` function is to be supported.

这些方法用于实现二元算术运算(+,-,*,/,

`__`**`div`**`__`(*self, other*)
`__`**`truediv`**`__`(*self, other*)

The division operator (`/`) is implemented by these methods. The `__truediv__()` method is used when `__future__.division` is in effect, otherwise `__div__()` is used. If only one of these two methods is defined, the object will not support division in the alternate context; `TypeError` will be raised instead.

这些方法用于支持除法运算符. __truediv__()**方法在**__future__.division**有效时使用**,**否则就使用**__div__(), **如果只定义了其中一个方法**, **那么这个对象在哪种情况下都是不支持除法运算的**,**此时会有一个**TypeError**异常抛出**.

`__`**`radd`**`__`(*self, other*)
`__`**`rsub`**`__`(*self, other*)
`__`**`rmul`**`__`(*self, other*)
`__`**`rdiv`**`__`(*self, other*)
`__`**`rtruediv`**`__`(*self, other*)

`__rfloordiv__`(*self, other*)
`__rmod__`(*self, other*)
`__rdivmod__`(*self, other*)
`__rpow__`(*self, other*)
`__rlshift__`(*self, other*)
`__rrshift__`(*self, other*)
`__rand__`(*self, other*)
`__rxor__`(*self, other*)
`__ror__`(*self, other*)

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `/`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation. For instance, to evaluate the expression *x*`-`*y*, where *y* is an instance of a class that has an `__rsub__()` method, *y*`.__rsub__(`*x*`)` is called. Note that ternary `pow()` will not try calling `__rpow__()` (the coercion rules would become too complicated).

这些方法用于实现二元算术运算(+,-,*,/,

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

These methods are called to implement the augmented arithmetic operations (`+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=`, `|=`). These methods should attempt to do the operation in-place (modifying *self*) and return the result (which could be, but does not have to be, *self*). If a specific method is not defined, the augmented operation falls back to the normal methods. For instance, to evaluate the expression *x*`+=`*y*, where *x* is an instance of a class that has an `__iadd__()` method, *x*`.__iadd__(`*y*`)` is called. If *x* is an instance of a class that does not define a `__iadd()` method, *x*`.__add__(`*y*`)` and *y*`.__radd__(`*x*`)` are considered, as with the evaluation of *x*`+`*y*.

这些方法用于实现赋值运算符(+=, -=,*=,/=,

`__neg__`(*self*)
`__pos__`(*self*)
`__abs__`(*self*)
`__invert__`(*self*)

Called to implement the unary arithmetic operations (`-`, `+`, `abs()` and `~`).

实现一元运算符(-,+,abs(),)

`__complex__`(*self*)
`__int__`(*self*)
`__long__`(*self*)
`__float__`(*self*)

Called to implement the built-in functions `complex()`, `int()`, `long()`, and `float()`. Should return a value of the appropriate type.

在自定义类上实现内建函数complex(), int(), long()**和**float()**的功能, 应该返回适当的类型**.

`__oct__`(*self*)
`__hex__`(*self*)

Called to implement the built-in functions `oct()` and `hex()`. Should return a string value.

在自定义类上实现内建函数oct()**和**hex()**的功能, 应该返回适当的串值**.

`__coerce__`(*self, other*)

Called to implement "mixed-mode" numeric arithmetic. Should either return a 2-tuple containing *self* and *other* converted to a common numeric type, or `None` if conversion is impossible. When the common type would be the type of `other`, it is sufficient to return `None`, since the interpreter will also ask the other object to attempt a coercion (but sometimes, if the implementation of the other type cannot be changed, it is useful to do the conversion to the other type here). A return value of `NotImplemented` is equivalent to returning `None`.

实现混合式数值运算, **返回值要么是一个包括已经转换到一般数值类型的数据的**self**和**other**的二元组**,**或者当转换无效时返回**None. **当转换后的类型可能是其它类型时**, **可以返回**None, **因为解释器会试着强制地转换为其它类型**(**但有的情况下**,**如果其它的类型的实现不能改变**, **那么就应该在这将其转换为该类型**).

3.3.8 强制规则: Coercion rules

This section used to document the rules for coercion. As the language has evolved, the coercion rules have become hard to document precisely; documenting what one version of one particular implementation does is undesirable. Instead, here are some informal guidelines regarding coercion. In Python 3.0, coercion will not be supported.

- If the left operand of a % operator is a string or Unicode object, no coercion takes place and the string formatting operation is invoked instead.
- It is no longer recommended to define a coercion operation. Mixed-mode operations on types that don't define coercion pass the original arguments to the operation.
- New-style classes (those derived from `object`) never invoke the `__coerce__()` method in response to a binary operator; the only time `__coerce__()` is invoked is when the built-in function `coerce()` is called.
- For most intents and purposes, an operator that returns `NotImplemented` is treated the same as one that is not implemented at all.
- Below, `__op__()` and `__rop__()` are used to signify the generic method names corresponding to an operator; `__iop__` is used for the corresponding in-place operator. For example, for the operator '`+`', `__add__()` and `__radd__()` are used for the left and right variant of the binary operator, and `__iadd__` for the in-place variant.
- For objects *x* and *y*, first *x*`.__op__(`*y*`)` is tried. If this is not implemented or returns `NotImplemented`, *y*`.__rop__(`*x*`)` is tried. If this is also not implemented or returns `NotImplemented`, a `TypeError` exception is raised. But see the following exception:
- Exception to the previous item: if the left operand is an instance of a built-in type or a new-style class, and the right operand is an instance of a proper subclass of that type or class, the right operand's `__rop__()` method is tried *before* the left operand's `__op__()` method. This is done so that a subclass can completely override binary operators. Otherwise, the left operand's __op__ method would always accept the right operand: when an instance of a given class is expected, an instance of a subclass of that class is always acceptable.
- When either operand type defines a coercion, this coercion is called before that type's `__op__()` or `__rop__()` method is called, but no sooner. If the coercion returns an object of a different type for the operand whose coercion is invoked, part of the process is redone using the new object.
- When an in-place operator (like '`+=`') is used, if the left operand implements `__iop__()`, it is invoked without any coercion. When the operation falls back to `__op__()` and/or `__rop__()`, the normal coercion rules apply.
- In *x*+*y*, if *x* is a sequence that implements sequence concatenation, sequence concatenation is invoked.
- In *x*`*`*y*, if one operator is a sequence that implements sequence repetition, and the other is an integer (`int` or `long`), sequence repetition is invoked.
- Rich comparisons (implemented by methods `__eq__()` and so on) never use coercion. Three-way comparison (implemented by `__cmp__()`) does use coercion under the same conditions as other binary operations use it.

- In the current implementation, the built-in numeric types `int`, `long` and `float` do not use coercion; the type `complex` however does use it. The difference can become apparent when subclassing these types. Over time, the type `complex` may be fixed to avoid coercion. All these types implement a `__coerce__()` method, for use by the built-in `coerce()` function.

0. 如果x是串对象,并且op是取模运算符(

1. 如果x是一个类实例:

1a. 如果x有__coerce__()方法:就用x.__coerce__(y)返回的二元组的值替换x和y. 如果它返回None则跳过步骤2.

1b. 如果x或y在强制转换后都不是类实例了, 转到步骤3.

1c. 如果x有方法__op__(), 返回x.__op__(y);否则恢复步骤1a之前的x和y的值.

2.

2a. 如果y有__coerce__()方法:就用y.__coerce__(x)返回的二元组的值替换y和x. 如果它返回None则跳过步骤2.

2b. 如果x或y在强制转换后都不是类实例了, 转到步骤3.

2b. 如果y有方法__rop__(), 返回y.__rop__(x);否则恢复步骤2a之前的x和y的值.

3. 仅仅在x和y都不是类实例时, 才会执行到这一步.

3a. 如果op为+并且x是一个有序类型, 那么就执行有序类型的连接操作.

3b. 如果op为*并且一个操作数为有序类型, 另一个是整数, 就执行有序类型重复操作.

3c. 否则, 两个操作数必须是数值型的; 它们尽可能地强制转换成通用类型, 并且为该类型调用数值运算符.

第四章

运行模型Execution model

4.1 !!! 代码块, 运行结构框架和命名空间Naming and binding

Names refer to objects. Names are introduced by name binding operations. Each occurrence of a name in the program text refers to the *binding* of that name established in the innermost function block containing the use.

A *block* is a piece of Python program text that is executed as a unit. The following are blocks: a module, a function body, and a class definition. Each command typed interactively is a block. A script file (a file given as standard input to the interpreter or specified on the interpreter command line the first argument) is a code block. A script command (a command specified on the interpreter command line with the '**-c**' option) is a code block. The file read by the built-in function `execfile()` is a code block. The string argument passed to the built-in function `eval()` and to the `exec` statement is a code block. The expression read and evaluated by the built-in function `input()` is a code block.

一个代码块是一个可以作为一个单元执行的Python程序文本, 像模块, 类定义或函数体. 有些代码块(如模块)通常只执行一次, 其它(例如函数体)可能会执行多次.代码块可以直接包含其它代码块, 也可以调用其它代码块(可能包括也可能不包括它们), 例如调用函数.

A code block is executed in an *execution frame*. A frame contains some administrative information (used for debugging) and determines where and how execution continues after the code block's execution has completed.

每个代码块在一个运行结构框架中执行.运行结构框架包括一个管理信息(用于调试),决定执行完代码块后在哪继续执行和怎么执行, 并且(可能也是最重要的)定义两个名字空间, 对于所执行代码块有效的局部和全局名字空间.

A *scope* defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. If the definition occurs in a function block, the scope extends to any blocks contained within the defining one, unless a contained block introduces a different binding for the name. The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods.

When a name is used in a code block, it is resolved using the nearest enclosing scope. The set of all such scopes visible to a code block is called the block's *environment*.

If a name is bound in a block, it is a local variable of that block. If a name is bound at the module level, it is a global variable. (The variables of the module code block are local and global.) If a variable is used in a code block but not defined there, it is a *free variable*.

When a name is not found at all, a `NameError` exception is raised. If the name refers to a local variable that has not been bound, a `UnboundLocalError` exception is raised. `UnboundLocalError` is a subclass of `NameError`.

The following constructs bind names: formal parameters to functions, `import` statements, class and function definitions (these bind the class or function name in the defining block), and targets that are identifiers if occurring in an assignment, `for` loop header, or in the second position of an `except` clause header. The `import` statement of the form "'`from ...import *`"' binds all names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

A target occurring in a `del` statement is also considered bound for this purpose (though the actual semantics are to unbind the name). It is illegal to unbind a name that is referenced by an enclosing scope; the compiler will report a `SyntaxError`.

Each assignment or import statement occurs within a block defined by a class or function definition or at the module level (the top-level code block).

If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block. This can lead to errors when a name is used within a block before it is bound. This rule is subtle. Python lacks declarations and allows name binding operations to occur anywhere within a code block. The local variables of a code block can be determined by scanning the entire text of the block for name binding operations.

If the global statement occurs within a block, all uses of the name specified in the statement refer to the binding of that name in the top-level namespace. Names are resolved in the top-level namespace by searching the global namespace, i.e. the namespace of the module containing the code block, and the builtin namespace, the namespace of the module `__builtin__`. The global namespace is searched first. If the name is not found there, the builtin namespace is searched. The global statement must precede all uses of the name.

The built-in namespace associated with the execution of a code block is actually found by looking up the name `__builtins__` in its global namespace; this should be a dictionary or a module (in the latter case the module's dictionary is used). Normally, the `__builtins__` namespace is the dictionary of the built-in module `__builtin__` (note: no 's'). If it isn't, restricted execution mode is in effect.

The namespace for a module is automatically created the first time a module is imported. The main module for a script is always called `__main__`.

The global statement has the same scope as a name binding operation in the same block. If the nearest enclosing scope for a free variable contains a global statement, the free variable is treated as a global.

A class definition is an executable statement that may use and define names. These references follow the normal rules for name resolution. The namespace of the class definition becomes the attribute dictionary of the class. Names defined at the class scope are not visible in methods.

4.1.1 Interaction with dynamic features

There are several cases where Python statements are illegal when used in conjunction with nested scopes that contain free variables.

If a variable is referenced in an enclosing scope, it is illegal to delete the name. An error will be reported at compile time.

If the wild card form of import — '`import *`' — is used in a function and the function contains or is a nested block with free variables, the compiler will raise a SyntaxError.

If `exec` is used in a function and the function contains or is a nested block with free variables, the compiler will raise a `SyntaxError` unless the exec explicitly specifies the local namespace for the `exec`. (In other words, '`exec obj`' would be illegal, but '`exec obj in ns`' would be legal.)

The `eval()`, `execfile()`, and `input()` functions and the `exec` statement do not have access to the full environment for resolving names. Names may be resolved in the local and global namespaces of the caller. Free variables are not resolved in the nearest enclosing namespace, but in the global namespace.[1] The `exec` statement and the `eval()` and `execfile()` functions have optional arguments to override the global and local namespace. If only one namespace is specified, it is used for both.

4.2 异常Exceptions

Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions. An exception is *raised* at the point where the error is detected; it may be *handled* by

[1]This limitation occurs because the code that is executed by these operations is not available at the time the module is compiled.

the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred.

异常就是为了处理出错或者处理其它意外情况而中断代码块的正常控制流。异常在错误被检测到的位置被抛出.**它可以被其周围相关代码的处理**, **或者错误发生处直接或间接调用的代码块处理**.

The Python interpreter raises an exception when it detects a run-time error (such as division by zero). A Python program can also explicitly raise an exception with the `raise` statement. Exception handlers are specified with the `try` ... `except` statement. The `try` ... `finally` statement specifies cleanup code which does not handle the exception, but is executed whether an exception occurred or not in the preceding code.

Python**解释器在它检测到一个运行时错误时抛出一个异常**(**比如除法零**).**某个**Python **程序也可以通过**raise**语句显式地抛出异常**.**异常处理器可以用**try ... except**语句指定**. try ... finally**语句指定清理代码块，但是它不处理异常，只是无论先前代码中是否产生异常都会得到执行。**

Python uses the "termination" model of error handling: an exception handler can find out what happened and continue execution at an outer level, but it cannot repair the cause of the error and retry the failing operation (except by re-entering the offending piece of code from the top).

Python**使用所谓的**"**中断**"**错误处理模型**:**一个异常处理器能在外层找出错误发生和继续执行的位置**. **但是它不能修复错误和重试错误的操作**(**除非重新从头进入该段出错的代码**).

When an exception is not handled at all, the interpreter terminates execution of the program, or returns to its interactive main loop. In either case, it prints a stack backtrace, except when the exception is `SystemExit`.

当一个异常没有得到控制, **解释器就中断程序的执行**, **或返回到它的主循环的迭代中**. **其它情况下**, **除了不是**SystemExit**异常**, **它还打印一个堆栈跟踪回溯对象**.

Exceptions are identified by class instances. Selection of a matching except clause is based on object identity. The `except` clause must reference the same class or a base class of it.

异常由一个字符串对象或一个类实例标识.**所匹配的**except**子句的选择是基于对象标识的**. (**也就是说**, **两个具有相同值的字符串对象描述的是不同的对象**)**对于字符串对象**, except**子句必须引用相同的串对象**, **对于类异常**, except**子句必须引用相同的类或其基类**.

When an exception is raised, an object (maybe `None`) is passed as the exception's *value*; this object does not affect the selection of an exception handler, but is passed to the selected exception handler as additional information. For class exceptions, this object must be an instance of the exception class being raised.

当一个异常被抛出时, **某个对象**(**可能是**None)**会作为异常的参数或值被传给异常处理器**; **这个对象不影响异常处理器的选择**, **但会传递给异常处理器以提供扩展信息**.**对于类异常**, **这个对象必须是被抛出的异常类的实例**.

Warning: Messages to exceptions are not part of the Python API. Their contents may change from one version of Python to the next without warning and should not be relied on by code which will run under multiple versions of the interpreter.

See also the description of the `try` statement in section 7.4 and `raise` statement in section 6.9.

关于try**语句详见**7.4; **关于**raise**语句详见**6.9**。**

第五章

表达式Expressions

This chapter explains the meaning of the elements of expressions in Python.

本章描述了Python**中表达式的组成元素的含义。**

Syntax Notes: In this and the following chapters, extended BNF notation will be used to describe syntax, not lexical analysis. When (one alternative of) a syntax rule has the form

句法注意：在本章和之后的章节中，描述句法时使用与词法分析时不同的扩展BNF**记法。当某个句法规则（可能是可选的）具有如下形式**

```
name  ::=  othername
```

and no semantics are given, the semantics of this form of `name` are the same as for `othername`.

并且未给出特定语义时，name**的这种形式规则的意义就是其与**othername**含义相同。**

5.1 数值型间的转换Arithmetic conversions

When a description of an arithmetic operator below uses the phrase "the numeric arguments are converted to a common type," the arguments are coerced using the coercion rules listed at the end of chapter 3. If both arguments are standard numeric types, the following coercions are applied:

当用以下短语"数值型参数转换为通用类型"描述数值型操作数时, **参数使用第三章中结尾处的强制规则进行强制转换**.**如果两个参数都属于标准数值型的**, **就使用以下的强制规则**:

- If either argument is a complex number, the other is converted to complex; **如果其中一个参数是复数**, **另一个也要转换成复数**;
- otherwise, if either argument is a floating point number, the other is converted to floating point; **否则**, **如果其中一个参数是浮点数**, **另一个也要转换成浮点数**;
- otherwise, if either argument is a long integer, the other is converted to long integer; **否则**, **如果其中一个参数是长整数**, **另一个也要转换成长整数**;
- otherwise, both must be plain integers and no conversion is necessary. **否则**, **两个都是普通整数**, **不需要转换**.

Some additional rules apply for certain operators (e.g., a string left argument to the '%' operator). Extensions can define their own coercions.

对于某些运算符有特殊的规则(例如, "

5.2 原子Atoms

Atoms are the most basic elements of expressions. The simplest atoms are identifiers or literals. Forms enclosed in reverse quotes or in parentheses, brackets or braces are also categorized syntactically as atoms. The syntax for atoms is:

原子是表达式最基本的组成单位. 最简单的原子是标识符或者字面值. 以反引号,圆括号, 方括号或大括号括住的符号在句法上也看成是原子.原子的句法如下:

```
atom       ::=  identifier | literal | enclosure
enclosure  ::=  parenth_form | list_display
                | dict_display | string_conversion
```

5.2.1 标识符(名字) Identifiers (Names)

An identifier occurring as an atom is a name. See section 4.1 for documentation of naming and binding.

作为一个原子出现的标识符是对一个局部名字, 或全局名字或内建名字捆绑的引用. 如果该名字出现在某代码块的任意的一个地方(即使是在不可达的代码中), 而且它未在global**语句中的话, 那么它就是该代码块的局部名字. 当它没在代码块被赋值,或者虽然被赋值但是它是在**globals**语句中显式地列出的话, 它就是引用的一个全局名字(如果它存在), 或者一个内建名字(这个捆绑规则可以动态改变)**.5.1

When the name is bound to an object, evaluation of the atom yields that object. When a name is not bound, an attempt to evaluate it raises a `NameError` exception.

当某名字捆绑的是一个对象时, 使用该原子就是使用那个对象. 当某名字没有捆绑就直接使用它, 则会抛出一个 NameError**异常.**

私有名字变换: Private name mangling: When an identifier that textually occurs in a class definition begins with two or more underscore characters and does not end in two or more underscores, it is considered a *private name* of that class. Private names are transformed to a longer form before code is generated for them. The transformation inserts the class name in front of the name, with leading underscores removed, and a single underscore inserted in front of the class name. For example, the identifier `__spam` occurring in a class named `Ham` will be transformed to `_Ham__spam`. This transformation is independent of the syntactical context in which the identifier is used. If the transformed name is extremely long (longer than 255 characters), implementation defined truncation may happen. If the class name consists only of underscores, no transformation is done.

在类定义中, 以两个或多个下划线开始, 并且尾部不是以两个或多个下划线结束的标识符, 它被看作是类的私有名字. 在产生它的代码之前, 私有名字被变换成更长的形式. 这种变换是在将去掉前导下划线的类名插入到名字前，再在类名前插入一个下划线。.例如, 在类 Ham**中定义的标识符**__spam, **会被变换成**_Ham__spam. **本变换是不依赖于使用该标识符处代码的句法上的上下文的. 如果变换后的结果过长(超过**255**个字符), 就会执行该**Python**实现定义的截短名字的操作. 如果某类的名字仅仅由下划线组成, 这种变换是不会发生的.**

5.2.2 字面值Literals

Python supports string literals and various numeric literals:

Python**支持字符串字面值和数值型字面值：**

```
literal  ::=  stringliteral | integer | longinteger
              | floatnumber | imagnumber
```

Evaluation of a literal yields an object of the given type (string, integer, long integer, floating point number, complex number) with the given value. The value may be approximated in the case of floating point and imaginary (complex) literals. See section 2.4 for details.

使用一个字面值会得到一个具有给定值的相应类型的对象（字符串、整数、长整数、浮点数、复数），如果是浮点数和复数那么这个值可能是个近似值，详见2.4**节。**

All literals correspond to immutable data types, and hence the object's identity is less important than its value. Multiple evaluations of literals with the same value (either the same occurrence in the program text or a different occurrence) may obtain the same object or a different object with the same value.

所有字面值都属于不可变的数据类型，因此对象的标识比起它们的值来说显得次要一些。多次使用相同值的字面值（在程序代码中以相同形式出现或者以不同的形式出现）可能获得的是相同的对象或具有相同值的不同对象。

5.2.3 括号表达式Parenthesized forms

A parenthesized form is an optional expression list enclosed in parentheses:

一个括号表达式是位于一对小括号内可选的表达式表.

```
parenth_form  ::=  "(" [expression_list] ")"
```

A parenthesized expression list yields whatever that expression list yields: if the list contains at least one comma, it yields a tuple; otherwise, it yields the single expression that makes up the expression list.

表达式表生成什么类型的值括号表达式也就生成什么类型的值:**如果表达式表中包括了至少一个逗号**, **它就生成一个元组**; **否则**, **就生成那个组成表达式表的唯一的表达式**.

An empty pair of parentheses yields an empty tuple object. Since tuples are immutable, the rules for literals apply (i.e., two occurrences of the empty tuple may or may not yield the same object).

一个空的表达式表会生成一个空的元组对象.**因为元组是不可变的**, **因此这里适用字符串所用的规则**(**即**, **两个具有空的元组可能是同一个对象也可能是不同的对象**).

Note that tuples are not formed by the parentheses, but rather by use of the comma operator. The exception is the empty tuple, for which parentheses *are* required — allowing unparenthesized "nothing" in expressions would cause ambiguities and allow common typos to pass uncaught.

请注意元组不是依靠小括号成定义的, **而使用逗号**. **其中空元组是个例外**, **此时要求有小括号**—– **在表达式中允许没有小括号的**"**空**"**可能会引起歧义**,**并容易造成难以查觉的笔误**.

5.2.4 列表的表示List displays

A list display is a possibly empty series of expressions enclosed in square brackets:

一个列表用一对方括号括住的表达式序列(**可能为空**)**表示**:

```
test           ::=  and_test ( "or" and_test )* | lambda_form
testlist       ::=  test ( "," test )* [ "," ]
list_display   ::=  "[" [listmaker] "]"
listmaker      ::=  expression ( list_for | ( "," expression )* [","] )
list_iter      ::=  list_for | list_if
list_for       ::=  "for" expression_list "in" testlist [list_iter]
list_if        ::=  "if" test [list_iter]
```

A list display yields a new list object. Its contents are specified by providing either a list of expressions or a list comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and placed into the list object in that order. When a list comprehension is supplied, it consists of a single expression followed by at least one `for` clause and zero or more `for` or `if` clauses. In this case, the elements of the new list are those that would be produced by considering each of the `for` or `if` clauses a block, nesting from left to right, and evaluating the expression to produce a list element each time the innermost block is reached.

使用一个列表会生成一个新的列表对象.**它的值由表达式表或由深列表给出**.**当给出一个逗号分隔的表达式表时**, **从左到右地对每个元素求值然后按顺序放进列表对象中**. **如果给出的是深列表**, **它由至少是一个**for**子句以及后跟零个或多个**for/if**子句构成的一个表达式组成**. **此时**, **新列表的元素由每个**for**或**if**子句块决定**, **嵌套是从左至右方向的**, **而且每执行到最内部的语句块就产生一个列表元素**.

5.2.5 字典的表示Dictionary displays

A dictionary display is a possibly empty series of key/datum pairs enclosed in curly braces:

一个字典用一对大括号括住的键/数据对的序列(可能为空)表示:

```
dict_display    ::=  "{" [key_datum_list] "}"
key_datum_list  ::=  key_datum ("," key_datum)* [","]
key_datum       ::=  expression ":" expression
```

A dictionary display yields a new dictionary object.

使用一个字典会生成一个新的字典对象.

The key/datum pairs are evaluated from left to right to define the entries of the dictionary: each key object is used as a key into the dictionary to store the corresponding datum.

键/数据对按在字典中定义的从左到右的顺序求值:每个键对象作为键嵌入到字典中存储相应的数据.

Restrictions on the types of the key values are listed earlier in section 3.2. (To summarize,the key type should be hashable, which excludes all mutable objects.) Clashes between duplicate keys are not detected; the last datum (textually rightmost in the display) stored for a given key value prevails.

关于键值类型的限制已在前述3.2**提及(总而言之，键的类型应该是可散列的，这就排除了所有的可变对象)。重复键之间的冲突不会被检测到；对给定的(有重复的)键来说，最后出现的数据(就是文字显示中出现在最右边的)成为(最终的)胜利者。**

5.2.6 **串的转换**String conversions

A string conversion is an expression list enclosed in reverse (a.k.a. backward) quotes:

一个串转换是由一对反引号(‘)引用的表达式表.

```
string_conversion  ::=  "‘" expression_list "‘"
```

A string conversion evaluates the contained expression list and converts the resulting object into a string according to rules specific to its type.

串的转换先计算所包括的表达式表的值,然后按照其结果类型所特定的规则转换其结果对象为字符串.

If the object is a string, a number, `None`, or a tuple, list or dictionary containing only objects whose type is one of these, the resulting string is a valid Python expression which can be passed to the built-in function `eval()` to yield an expression with the same value (or an approximation, if floating point numbers are involved).

如果结果对象属于字符串, 数值, None, **或元组,字典其中的一个类型, 那么结果串就是可以通过传递给内建函数**eval()**来生成一个具有与原值相同的正确的**Python**表达式(或, 对于浮点数来说生成的是近似数)**.

(In particular, converting a string adds quotes around it and converts “funny” characters to escape sequences that are safe to print.)

(**特别地, 经增加反引号转换的字符串和由特殊字符串所转换得到的转义字符串是能够显示输出的**.)

Recursive objects (for example, lists or dictionaries that contain a reference to themselves, directly or indirectly) use ‘`...`’ to indicate a recursive reference, and the result cannot be passed to `eval()` to get an equal value (`SyntaxError` will be raised instead).

试图转换一个递归对象是无效的(例如, 一个直接或间接地包含有对自身引用的列表或字典.)

The built-in function `repr()` performs exactly the same conversion in its argument as enclosing it in parentheses and reverse quotes does. The built-in function `str()` performs a similar but more user-friendly conversion.

内建函数repr()**作相同的转换: 将其括号中的参数转换成用反引号引用它后的结果. 内建函数**str()**与之类似,但结果更具可读性**.

5.3 **基元**Primaries

Primaries represent the most tightly bound operations of the language. Their syntax is:

基元指和语言本身中接合最紧密的若干操作.它们的语法如下:

```
primary  ::=  atom | attributeref | subscription | slicing | call
```

5.3.1 属性引用Attribute references

An attribute reference is a primary followed by a period and a name:

一个属性引用是由一个主元(primary)**后跟一个句号和一个名字构成**:

```
attributeref  ::=  primary "." identifier
```

The primary must evaluate to an object of a type that supports attribute references, e.g., a module, list, or an instance. This object is then asked to produce the attribute whose name is the identifier. If this attribute is not available, the exception `AttributeError` is raised. Otherwise, the type and value of the object produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

主元必须是一个支持属性引用的类型的对象.**例如**, **模块**, **列表**, **或一个实例**. **引用对象属性时**, **即要求该被对象生成指定名字的属性**. **如果该属性无效**, **将会抛出异常**AttribError.**否则**,**产生的类型和对象由决定**.**对同一属性的多次求值是有可能生成不同对象的**.

5.3.2 下标Subscriptions

A subscription selects an item of a sequence (string, tuple or list) or mapping (dictionary) object:

一个下标选择一个有序类型对象或映射(**字典**)**对象的一项**:

```
subscription  ::=  primary "[" expression_list "]"
```

The primary must evaluate to an object of a sequence or mapping type.

主元(primary)**必须是一个有序类型或映射的对象**.

If the primary is a mapping, the expression list must evaluate to an object whose value is one of the keys of the mapping, and the subscription selects the value in the mapping that corresponds to that key. (The expression list is a tuple except if it has exactly one item.)

如果主元是一个映射, **则对表达式表求值的结果必须是映射中的一个键**, **然后此下标操作在主元映射中选择与该键所对应的值**.(**如果表达式表只有一项**,**那么它就是一个元组**)

If the primary is a sequence, the expression (list) must evaluate to a plain integer. If this value is negative, the length of the sequence is added to it (so that, e.g., `x[-1]` selects the last item of `x`.) The resulting value must be a nonnegative integer less than the number of items in the sequence, and the subscription selects the item whose index is that value (counting from zero).

如果主元是一个有序类型, **表达式表的计算结果应该是一个普通整数**.**如果这个值是负的**, **就加上该主元的长度**(**所以**, **例如**, x[-1]**选择**x**的最后一项**). **计算结果必须是一个小于主元中所含项数的非负整数**, **并且此下标操作选择以该数为索引**(**从**0**开始计**)**的值**.

A string's items are characters. A character is not a separate data type but a string of exactly one character.

字符串的元素是字符, **字符不是单独的数据类型而仅仅是只有一个字符长的字符串**..

5.3.3 片断Slicings

A slicing selects a range of items in a sequence object (e.g., a string, tuple or list). Slicings may be used as expressions or as targets in assignment or del statements. The syntax for a slicing:

一个片断选择某个有序类型对象(**如**, **字符串**, **元组**, **列表**)**一段范围之内的项**. **片断可以作为表达式使用**,**或者是赋值和**del**语句的目标**. **下面是片断的句法**:

```
slicing           ::=  simple_slicing | extended_slicing
simple_slicing    ::=  primary "[" short_slice "]"
extended_slicing  ::=  primary "[" slice_list "]"
slice_list        ::=  slice_item ("," slice_item)* [","]
slice_item        ::=  expression | proper_slice | ellipsis
proper_slice      ::=  short_slice | long_slice
short_slice       ::=  [lower_bound] ":" [upper_bound]
long_slice        ::=  short_slice ":" [stride]
lower_bound       ::=  expression
upper_bound       ::=  expression
stride            ::=  expression
ellipsis          ::=  "..."
```

There is ambiguity in the formal syntax here: anything that looks like an expression list also looks like a slice list, so any subscription can be interpreted as a slicing. Rather than further complicating the syntax, this is disambiguated by defining that in this case the interpretation as a subscription takes priority over the interpretation as a slicing (this is the case if the slice list contains no proper slice nor ellipses). Similarly, when the slice list has exactly one short slice and no trailing comma, the interpretation as a simple slicing takes priority over that as an extended slicing.

在这里形式句法的说明中有点含糊: **任何看起来像表达式表的语法构件也能看作是片断表**, **所以任何下标都可以解释为片断**. **但这样要比更复杂的句法要合适**, **该定义是没有歧义的**, **在这种情况下**(**在片断表中没有包括适当的片断或者省略写法**)**优先将其解释为下标**, **而不是片断**. **类似地**, **如果一个片断表很精确地是一个没有后跟逗号的简短片断**, **则将其优先解释为简单片断**,**而不是扩展片断**.

The semantics for a simple slicing are as follows. The primary must evaluate to a sequence object. The lower and upper bound expressions, if present, must evaluate to plain integers; defaults are zero and the `sys.maxint`, respectively. If either bound is negative, the sequence's length is added to it. The slicing now selects all items with index *k* such that *i* `<=` *k* `<` *j* where *i* and *j* are the specified lower and upper bounds. This may be an empty sequence. It is not an error if *i* or *j* lie outside the range of valid indexes (such items don't exist so they aren't selected).

一个简单片断的语义如下.**主要是要将其导成一个有序类型对象**. **如果给出了下限和上限表达式**, **它们必须是普通整数**; **其默认值为**0**或**sys.maxint, **分别的**, **如果其中一个界限为负数**, **那么将其与该有序类型对象的长度相加得到新的界限值**, **这样片断就是指出了所有的项**k, **其中**i ¡= k ¡ j , i**和**j**是下限和上限**.**这里允许有空有序类型对象**.**如果**i**或**j**超出索引的合法值并不算是错误**(**这样的项不存在**,**所以不被选择**).

The semantics for an extended slicing are as follows. The primary must evaluate to a mapping object, and it is indexed with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of an ellipsis slice item is the built-in `Ellipsis` object. The conversion of a proper slice is a slice object (see section 3.2) whose `start`, `stop` and `step` attributes are the values of the expressions given as lower bound, upper bound and stride, respectively, substituting `None` for missing expressions.

一个扩展片断的语义如下.**主要是要将其导成一个映射对象**, **并且由该片断表构成的键作索引**. **如果一个片断表包括至少一个逗号**, **那么键就是一个包括由片断中的所有项转换而来的元组**; **否则**, **就用独立的片断项作转换成为键**. **为表达式的片断项转换后仍是该表达式**, **包括有省略写法的片断项转换后是一个内建的**Ellipsis**对象**. **正常的片断转换后是一个**start,stop**和**step**属性为给定的下限**,**上限和步长的片断对象**(**见**3.2**节**), **对于缺少的表达式用**None**替代**.

5.3.4 调用Calls

A call calls a callable object (e.g., a function) with a possibly empty series of arguments:

一个调用就是以一系列参数(**可能为空**)**调用一个可调用对象**(**例如**, **函数**):

```
call                 ::=  primary "(" [argument_list [","]] ")"
argument_list        ::=  positional_arguments ["," keyword_arguments]
                           ["," "*" expression]
                           ["," "**" expression]
                          | keyword_arguments ["," "*" expression]
                           ["," "**" expression]
                          | "*" expression ["," "**" expression]
                          | "**" expression
positional_arguments ::=  expression ("," expression)*
keyword_arguments    ::=  keyword_item ("," keyword_item)*
keyword_item         ::=  identifier "=" expression
```

A trailing comma may be present after an argument list but does not affect the semantics.

在参数表后面可以出现一个逗号, **但它在语义上是没有任何作用的**.

The primary must evaluate to a callable object (user-defined functions, built-in functions, methods of built-in objects, class objects, methods of class instances, and certain class instances themselves are callable; extensions may define additional callable object types). All argument expressions are evaluated before the call is attempted. Please refer to section 7.5 for the syntax of formal parameter lists.

首先的工作是导出一个可调用对象(**用户自定义函数**, **内建函数**, **内建方法对象**, **类定义**, **类实例方法**, **某些类实例自身就是可调用的**, **扩展模块可能定义了自己的可调用对象**)**所有的参数表达都在试图调用之前被计算关于形参表的句法参见**7.5.

If keyword arguments are present, they are first converted to positional arguments, as follows. First, a list of unfilled slots is created for the formal parameters. If there are N positional arguments, they are placed in the first N slots. Next, for each keyword argument, the identifier is used to determine the corresponding slot (if the identifier is the same as the first formal parameter name, the first slot is used, and so on). If the slot is already filled, a `TypeError` exception is raised. Otherwise, the value of the argument is placed in the slot, filling it (even if the expression is `None`, it fills the slot). When all arguments have been processed, the slots that are still unfilled are filled with the corresponding default value from the function definition. (Default values are calculated, once, when the function is defined; thus, a mutable object such as a list or dictionary used as default value will be shared by all calls that don't specify an argument value for the corresponding slot; this should usually be avoided.) If there are any unfilled slots for which no default value is specified, a `TypeError` exception is raised. Otherwise, the list of filled slots is used as the argument list for the call.

如果给出了关键字参数, **它们首先被转换为位置参数**.**具体如下**:**第一步**,**根据形参表创建一串空闲槽**, **如果有**N**个位置参数**, **它们就被放在前**N**个槽中**. **然后**, **对于每个关键字参数**, **它的标识符用于检测其对应的槽**(**如果其标识符与第一个形参数名相同**, **它就占用第一个槽**,**以此类推**)**如果发现某个槽已经被占用**, **则引发**TypeError**异常**.**否则将参数的值**(**即使为**None)**放进槽中**. **当所有关键字参数处理完成后**, **所有未填充的槽用在函数定义中的相应的默认值填充**. (**默认值是由函数定义时计算出来的**, **所以**, **像列表和字典这样的可变类型对象作默认值时**, **它们会被那些没有相应槽指定参数值的调用所共享**, **通常要避免这样做**). **如果仍有未填充默认的槽位**, **就会引发一个**TypeError**异常**.**否则**, **所有被填充的槽当作调用的参数表**.

If there are more positional arguments than there are formal parameter slots, a `TypeError` exception is raised, unless a formal parameter using the syntax '`*identifier`' is present; in this case, that formal parameter receives a tuple containing the excess positional arguments (or an empty tuple if there were no excess positional arguments).

如果位置参数的个数比形参槽数多, **并且在未使用**"*identifier"**句法的情况下**,**会引发**TypeError**异常**. **使用该种句法时**, **形参接受一个包括有额外位置参数的元组**(**如果没有额外和位置参数**, **它就为空**).

If any keyword argument does not correspond to a formal parameter name, a `TypeError` exception is raised, unless a formal parameter using the syntax '`**identifier`' is present; in this case, that formal parameter receives a dictionary containing the excess keyword arguments (using the keywords as keys and the argument values as corresponding values), or a (new) empty dictionary if there were no excess keyword arguments.

如果任何一个关键字参数与形参名不匹配, **并且在未使用**"**identifier"**句法的情况下**,**会引发**TypeError**异常**. **使用该种句法时**, **形参接受一个包括有额外关键字参数的字典**(**关键字作为键**, **参数值作为该键对应的值**), **字典如果没有额外和关键字参数**, **它就为空**.

If the syntax '`*expression`' appears in the function call, '`expression`' must evaluate to a sequence. Elements from this sequence are treated as if they were additional positional arguments; if there are postional arguments *x1,...,xN* , and '`expression`' evaluates to a sequence *y1,...,yM*, this is equivalent to a call with M+N positional arguments *x1,...,xN,y1,...,yM*.

如果在函数调用中使用了”*exprsiones”**句法**, **那么**”exprsiones”**的结果必须是有序类型的**. **这个有序类型对象的元素被当作附加的位置参数处理**; **如果存在有位置参数**x1,...,xN , **并且**”*exprsiones”**的计算结果为**y1,...,yM, **那么它与具有**M+N**个参数**x1,...,xN,y1,...,yM **的调用等效**.

A consequence of this is that although the '`*expression`' syntax appears *after* any keyword arguments, it is processed *before* the keyword arguments (and the '`**expression`' argument, if any – see below). So:

由此可以得到一个推论: **尽管**”*expression” **句法出现在任何关键字参数之后，但它在处理关键字参数之前计算．（如果有的话，**”**expression” **也是如此，参见下述），所以：**

```
>>> def f(a, b):
...  print a, b
...
>>> f(b=1, *(2,))
2 1
>>> f(a=1, *(2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: f() got multiple values for keyword argument ’a’
>>> f(1, *(2,))
1 2
```

It is unusual for both keyword arguments and the '`*expression`' syntax to be used in the same call, so in practice this confusion does not arise.

一同使用关键字语法和”*expression” **的情况十分罕见，所以实际上这种混乱是不会发生的．**

If the syntax '`**expression`' appears in the function call, '`expression`' must evaluate to a (subclass of) dictionary, the contents of which are treated as additional keyword arguments. In the case of a keyword appearing in both '`expression`' and as an explicit keyword argument, a `TypeError` exception is raised.

如果在函数调用中使用”**expression”**句法，**”expression” **计算结果必须是一个字典（的子类）．其内容作为附加的关键字参数．如果一个关键字出现在**”expression” **中并且是一个显式关键字参数，就会引发**TypeError **异常．**

Formal parameters using the syntax '`*identifier`' or '`**identifier`' cannot be used as positional argument slots or as keyword argument names. Formal parameters using the syntax '`(sublist)`' cannot be used as keyword argument names; the outermost sublist corresponds to a single unnamed argument slot, and the argument value is assigned to the sublist using the usual tuple assignment rules after all other parameter processing is done.

使用”*identifier”**或**”**identifier”**句法的形参不能作为位置参数或关键字参数名使用．使用**”(sublist)”**句法的形参也不能作为关键字参数名．最外层的**sublist**对应一个匿名参数槽，并在处理完其它参数之后使用通常的元组赋值法则将**sublist**赋为参数值．**

A call always returns some value, possibly `None`, unless it raises an exception. How this value is computed depends on the type of the callable object.

一个元组如果没有引发异常，通常会返回一些值，可能为 None. **怎样计算这个值依赖于可调用对象的类型．**

If it is—

如果它是—-

用户自定义函数a user-defined function: The code block for the function is executed, passing it the argument list. The first thing the code block will do is bind the formal parameters to the arguments; this is described in section 7.5. When the code block executes a `return` statement, this specifies the return value of the function call.

执行此函数的代码块，并把参数传给它．它要做的第一件事就是将形参与实参对应起来．关于这点参见7.5**当代码块执行到**return**语句时，会指定这次函数调用的返回值**.

内建函数或内建方法a built-in function or method: The result is up to the interpreter; see the *Python Library Reference* for the descriptions of built-in functions and methods.

结果依赖于解释器，详见《Python**库参考》**

类对象a class object: A new instance of that class is returned.

返回该类的一个新实例。

类实例的方法a class instance method: The corresponding user-defined function is called, with an argument list that is one longer than the argument list of the call: the instance becomes the first argument.

调用对应的用户自定义函数，其参数个数比普通的函数调用多一：该实例成为方法的第一个参数。

类实例a class instance: The class must define a `__call__()` method; the effect is then the same as if that method was called.

类实例必须定义方法__call__()**；效果与调用该方法相同。**

5.4 幂运算符The power operator

The power operator binds more tightly than unary operators on its left; it binds less tightly than unary operators on its right. The syntax is:

幂运算符比在操作数左边的一元运算符有更高的优先级; **但比右面的一元运算符要低**.**句法为**:

```
power  ::=  primary ["**" u_expr]
```

Thus, in an unparenthesized sequence of power and unary operators, the operators are evaluated from right to left (this does not constrain the evaluation order for the operands).

因此, **在一串没有括号的由幂运算符和一元运算符组成的序列**, **会从左到右面求值**(**没有强制改变求值顺序**).

The power operator has the same semantics as the built-in `pow()` function, when called with two arguments: it yields its left argument raised to the power of its right argument. The numeric arguments are first converted to a common type. The result type is that of the arguments after coercion.

当以两个参数调用pow() **时**, **幂运算符与内建函数**pow()**有相同的语义**: **生成左边参数的右边参数次方**.**数值型参数首先转换成通用类型**. **结果类型是参数经强制规则转换后的结果**; **若结果不能以该类型表达**(**计算整数的幂结果为负数**, **或计算负浮点数的幂为无效的值**), **则引发一个**TypeError **异常**.

With mixed operand types, the coercion rules for binary arithmetic operators apply. For int and long int operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns `100`, but `10**-2` returns `0.01`. (This last feature was added in Python 2.2. In Python 2.1 and before, if both arguments were of integer types and the second argument was negative, an exception was raised).

Raising `0.0` to a negative power results in a `ZeroDivisionError`. Raising a negative number to a fractional power results in a `ValueError`.

5.5 一元算术运算符Unary arithmetic operations

All unary arithmetic (and bit-wise) operations have the same priority:

所有一元算术运算符(**和位运算符**)**有相同的优先级**:

```
u_expr  ::=  power | "-" u_expr | "+" u_expr | "~" u_expr
```

The unary `-` (minus) operator yields the negation of its numeric argument.

一元运算符-(**减**)**对其数值型操作数取负**.

The unary `+` (plus) operator yields its numeric argument unchanged.

一元运算符+(**加**)**不改变其数值型操作数**.

The unary ~ (invert) operator yields the bit-wise inversion of its plain or long integer argument. The bit-wise inversion of `x` is defined as `-(x+1)`. It only applies to integral numbers.

一元运算符 (**取反**)**对其普通整数或长整数参数求逆**(**比特级**).x**的比特级求逆运算定义为**-(x+1).**它仅仅用于整数型的操作数**.

In all three cases, if the argument does not have the proper type, a `TypeError` exception is raised.

在以上所有的三种情况下, **如果参数的类型不合法**, **就会引发一个**TypeError**异常**.

5.6 二元算术运算符Binary arithmetic operations

The binary arithmetic operations have the conventional priority levels. Note that some of these operations also apply to certain non-numeric types. Apart from the power operator, there are only two levels, one for multiplicative operators and one for additive operators:

二元算术运算符的优先级符合我们的正常习惯.**注意其中有些运算符也可以应用于非数值型操作数**.**除了幂运算符**, **它们只分两个优先级**: **一个是乘法类运算**,**一个是加法类运算**.

```
m_expr  ::=  u_expr | m_expr "*" u_expr | m_expr "//" u_expr | m_expr "/" u_expr
             | m_expr "%" u_expr
a_expr  ::=  m_expr | a_expr "+" m_expr | a_expr "-" m_expr
```

The `*` (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be an integer (plain or long) and the other must be a sequence. In the former case, the numbers are converted to a common type and then multiplied together. In the latter case, sequence repetition is performed; a negative repetition factor yields an empty sequence.

*(**乘**)**运算符计算其操作数的积**, **其两个参数必须是数值型的**, **或一个是整数**(**普通或长整数**)**另一个是有序类型**.**第一种情况下**, **数值参数被转换成通用类型然后计算积**. **后一种情况下**, **重复连接有序类型对象**. **一个负连接因子产生一个有空的有序类型对象**.

The `/` (division) and `//` (floor division) operators yield the quotient of their arguments. The numeric arguments are first converted to a common type. Plain or long integer division yields an integer of the same type; the result is that of mathematical division with the 'floor' function applied to the result. Division by zero raises the `ZeroDivisionError` exception.

/(**除**)**运算符生成参数的商**.**数值型参数首先转换成通用类型**,**普通整数或长整数的除法计算结果是相同类型整数**,**结果就是对商的精确结果执行**floor()**函数的返回在值**.**除以零会引发**ZeroDivisionError **异常**.

The `%` (modulo) operator yields the remainder from the division of the first argument by the second. The numeric arguments are first converted to a common type. A zero right argument raises the `ZeroDivisionError` exception. The arguments may be floating point numbers, e.g., `3.14%0.7` equals `0.34` (since `3.14` equals `4*0.7 + 0.34`.) The modulo operator always yields a result with the same sign as its second operand (or zero); the absolute value of the result is strictly smaller than the absolute value of the second operand[1].

%(**取模**)**计算第一个参数除以第二参数得到的余数**.**数值型参数首先转换成通用类型**. **右面的参数为零**, **会引发**ZeroDivisionError**异常**. **参数可以浮点数**,**例如**3.14

The integer division and modulo operators are connected by the following identity: `x == (x/y)*y + (x%y)`. Integer division and modulo are also connected with the built-in function `divmod()`: `divmod(x, y) == (x/y, x%y)`. These identities don't hold for floating point numbers; there similar identities hold approximately where `x/y` is replaced by `floor(x/y)` or `floor(x/y) - 1`[2].

整除和取模运算可以用以下等式联系起来: x == (x/y)*y + (x

[1]While `abs(x%y) < abs(y)` is true mathematically, for floats it may not be true numerically due to roundoff. For example, and assuming a platform on which a Python float is an IEEE 754 double-precision number, in order that `-1e-100 %1e100` have the same sign as `1e100`, the computed result is `-1e-100 + 1e100`, which is numerically exactly equal to `1e100`. Function `fmod()` in the `math` module returns a result whose sign matches the sign of the first argument instead, and so returns `-1e-100` in this case. Which approach is more appropriate depends on the application.

[2]If x is very close to an exact integer multiple of y, it's possible for `floor(x/y)` to be one larger than `(x-x%y)/y` due to rounding. In such cases, Python returns the latter result, in order to preserve that `divmod(x,y)[0] * y + x % y` be very close to `x`.

Deprecated since release 2.3. The floor division operator, the modulo operator, and the `divmod()` function are no longer defined for complex numbers. Instead, convert to a floating point number using the `abs()` function if appropriate.

The `+` (addition) operator yields the sum of its arguments. The arguments must either both be numbers or both sequences of the same type. In the former case, the numbers are converted to a common type and then added together. In the latter case, the sequences are concatenated.

+(**加**)**运算符计算参数的和**.**参数必须都是数值型的**, **或都是相同有序类型的对象**. **对于前一种情况**, **它们先转换成通用类型**, **然后相加**. **后一种情况下**, **所有有序类型对象被连接起来**

The `-` (subtraction) operator yields the difference of its arguments. The numeric arguments are first converted to a common type.

-(**减**)**计算参数的差**, **数值型的参数首先转换成通用类型**.

5.7 移位运算符Shifting operations

The shifting operations have lower priority than the arithmetic operations:

移位运算符的优先级比算术运算符低.

```
shift_expr  ::=  a_expr | shift_expr ( "<<" | ">>" ) a_expr
```

These operators accept plain or long integers as arguments. The arguments are converted to a common type. They shift the first argument to the left or right by the number of bits given by the second argument.

这些运算符接受普通整数和长整数作为参数. **参数都被转换通用类型**.**它们将第一个参数向左或向右移动第二个参数指出的位数**.

A right shift by *n* bits is defined as division by `pow(2,n)`. A left shift by *n* bits is defined as multiplication with `pow(2,n)`; for plain integers there is no overflow check so in that case the operation drops bits and flips the sign if the result is not less than `pow(2,31)` in absolute value. Negative shift counts raise a `ValueError` exception.

右移n**位可以定义为除以**pow(2,n),**左移**n**位可以定义为乘以**pow(2,n); **对于普通整数是没有溢出检查的**,**因此若结果的绝对值不小于**pow(2,31), **这个运算会截掉相应的位并且符号位也在移位处理之列**.

5.8 二元位运算符Binary bit-wise operations

Each of the three bitwise operations has a different priority level:

三个二元位运算符具有各不相同的优先级:

```
and_expr  ::=  shift_expr | and_expr "&" shift_expr
xor_expr  ::=  and_expr | xor_expr "^" and_expr
or_expr   ::=  xor_expr | or_expr "|" xor_expr
```

The & operator yields the bitwise AND of its arguments, which must be plain or long integers. The arguments are converted to a common type.

& **运算符进行比特级的**AND(**与**)**运算**, **参数必须是普通整数或长整数**.**参数转换成通用类型**.

The ˆ operator yields the bitwise XOR (exclusive OR) of its arguments, which must be plain or long integers. The arguments are converted to a common type.

ˆ **运算符进行比特级的**XOR(**异或**)**运算**, **参数必须是普通整数或长整数**.**参数转换成通用类型**.

The `|` operator yields the bitwise (inclusive) OR of its arguments, which must be plain or long integers. The arguments are converted to a common type.

— **运算符进行比特级的**OR(**同或**)**运算**, **参数必须是普通整数或长整数**.**参数转换成通用类型**.

5.9 比较Comparisons

Unlike C, all comparison operations in Python have the same priority, which is lower than that of any arithmetic, shifting or bitwise operation. Also unlike C, expressions like `a < b < c` have the interpretation that is conventional in mathematics:

不像C**语言**, **在**Python**中所有比较运算具有相同的优先级**: **比所有算术运算符**,**移位运算符**,**和位运算符都要低**.**并且**, **表达式**a ¡ b ¡ c **具有和数学上一样的含义**:

```
comparison     ::=  or_expr ( comp_operator or_expr )*
comp_operator  ::=  "<" | ">" | "==" | ">=" | "<=" | "<>" | "!="
                    | "is" ["not"] | ["not"] "in"
```

Comparisons yield boolean values: `True` or `False`.

比较运算生成逻辑值: True **意味着结果为真**, False **意味着结果为假**,

Comparisons can be chained arbitrarily, e.g., `x < y <= z` is equivalent to `x < y and y <= z`, except that `y` is evaluated only once (but in both cases `z` is not evaluated at all when `x < y` is found to be false).

比较可以任意的连接, **例如**, x ¡ y ¡= z **等价于**x ¡ y and y ¡= z,**除了**y**只求值一次**(**但在这两种情况下**, **只要发现**x ¡ y **为假**, z**就不会被计算**) .

Formally, if *a*, *b*, *c*, . . . , *y*, *z* are expressions and *opa*, *opb*, . . . , *opy* are comparison operators, then *a opa b opb c* . . .*y opy z* is equivalent to *a opa b* `and` *b opb c* `and` . . .*y opy z*, except that each expression is evaluated at most once.

形式上, **如果**a, b, c, ..., y, z **为表达式**, opa, opb, ..., opy **为比较运算符**,**则**a opa b opb c ...y opy z **等价于**a opa b and b opb c and ... y opy z, **除了每个表达式最多只求值一次**.

Note that *a opa b opb c* doesn't imply any kind of comparison between *a* and *c*, so that, e.g., `x < y > z` is perfectly legal (though perhaps not pretty).

注意a opa b opb c **并没有隐式地规定**a **和**c **之间的比较运算种类**, **所以**x ¡ y ¿ z **是完全合法的**(**虽然可能不太漂亮**).

The forms <> and != are equivalent; for consistency with C, != is preferred; where != is mentioned below <> is also accepted. The <> spelling is considered obsolescent.

¡¿**和**!=**是等价的**, **考虑对**C**语言的连贯性**, **推荐使用**!=, **以下对**!=**的讨论对**¡¿**也是成立的**.¡¿**写法已经考虑废除了**.

The operators <, >, ==, >=, <=, and != compare the values of two objects. The objects need not have the same type. If both are numbers, they are converted to a common type. Otherwise, objects of different types *always* compare unequal, and are ordered consistently but arbitrarily.

运算符¡, ¿, ==, ¿=, ¡=, **和**!= **比较两个对象的值**, **它们不需要具有相同的的类型**.**如果两个都是数值型的**, **它们都转换成通用类型**. **否则**, **不同类型的对象之间的比较通常是不等的**.**并且顺序通常是固定的**, **但顺序是任意的**.

(This unusual definition of comparison was used to simplify the definition of operations like sorting and the `in` and `not in` operators. In the future, the comparison rules for objects of different types are likely to change.)

(**这个不太自然的比较规则用于简化像排序操作和**in**或**not in**运算符**.**以后**, **关于不同类型的对象的比较规则很可能会改变**.)

Comparison of objects of the same type depends on the type:

相同类型的对象的比较法则依赖于该类型:

- Numbers are compared arithmetically.

  数值型按大小比较.

- Strings are compared lexicographically using the numeric equivalents (the result of the built-in function `ord()`) of their characters. Unicode and 8-bit strings are fully interoperable in this behavior.

  串按字典序比较(**每个字符的序数用内建函数**(ord()**得到**). Unicode**和八位长字符完全可以同时使用**.

- Tuples and lists are compared lexicographically using comparison of corresponding elements. This means that to compare equal, each element must compare equal and the two sequences must be of the same type and have the same length.

  元组和按列表字典序比较((通过比较对应的项).

  If not equal, the sequences are ordered the same as their first differing elements. For example, `cmp([1,2,x], [1,2,y])` returns the same as `cmp(x,y)`. If the corresponding element does not exist, the shorter sequence is ordered first (for example, `[1,2] < [1,2,3]`).

- Mappings (dictionaries) compare equal if and only if their sorted (key, value) lists compare equal.[3] Outcomes other than equality are resolved consistently, but are not otherwise defined.[4]

  映射(**字典**)**仅当它们存储**(**键值对**)**表一样时相等**. [5] **这类相等不同于通常意义上的相等**, **但还没有定义其它的比较方法**. [6]

- Most other types compare unequal unless they are the same object; the choice whether one object is considered smaller or larger than another one is made arbitrarily but consistently within one execution of a program.

  对大多数其它类型对象进行比较, **如果不是相同的对象则结果就是不等的**.**一个对象被看作比另一个对象小或大**, **是不可以预知的**, **但在相同的程序其结果是前后一致的**.

The operators `in` and `not in` test for set membership. *x* `in` *s* evaluates to true if *x* is a member of the set *s*, and false otherwise. *x* `not in` *s* returns the negation of *x* `in` *s*. The set membership test has traditionally been bound to sequences; an object is a member of a set if the set is a sequence and contains an element equal to that object. However, it is possible for an object to support membership tests without being a sequence. In particular, dictionaries support memership testing as a nicer way of spelling *key* `in` *dict*; other mapping types may follow suit.

in**运算符和**not in **运算符用于测试集合成员**. **如果**x**是集合**s**的成员**, **那么**x in s**的结果为真**, **否则为假**. x not in s**的结果与上相反**. **集合成员测试运算通常用在有序类型对象中**; **某对象是集合的一个成员**, **即这个有序类型对象包括有与该对象相等的元素**.**但是也允许不是有序类型的对象支持集合成员测试运算**; **特别地**, **支持集合成员测试的字典提供了一个不错的方法测试**dict**中的**key; **其它映射类型可能提供类似的机制**.

For the list and tuple types, *x* `in` *y* is true if and only if there exists an index *i* such that *x* `==` *y*`[`*i*`]` is true.

对于列表和元组类型, x in y**当且仅当**y**具有合法的索引**i, **并且**x == y[i]**为真**.

For the Unicode and string types, *x* `in` *y* is true if and only if *x* is a substring of *y*. An equivalent test is `y.find(x) != -1`. Note, *x* and *y* need not be the same type; consequently, `u'ab' in 'abc'` will return `True`. Empty strings are always considered to be a substring of any other string, so `"" in "abc"` will return `True`. Changed in version 2.3: Previously, *x* was required to be a string of length `1`.

对于Unicode**和串类型**, x in y**当且仅当**y**具有合法的索引**i, **并且**x == y[i]**为真**.**如果**x**不是长度为**1**的字符串或**Unicode**串**, **就会引发**TypeError **异常**.

For user-defined classes which define the `__contains__()` method, *x* `in` *y* is true if and only if *y*`.__contains__(`*x*`)` is true.

对于定义了__contains__()**方法的用户自定义类**, x in y**为真仅当**y.__contains__(x)**返回**true.

For user-defined classes which do not define `__contains__()` and do define `__getitem__()`, *x* `in` *y* is true if and only if there is a non-negative integer index *i* such that *x* `==` *y*`[`*i*`]`, and all lower integer indices do not raise `IndexError` exception. (If any other exception is raised, it is as if `in` raised that exception).

对于没有定义__contains__()**方法但定义有**__getitem__()**方法的用户自定义类**, x in y**当且仅当有一个非负的索引**i, **使**x == y[i]**满足**, **并且所有小于该数的索引不能引发**IndexError**异常**(**如果引发了任何其它异常**, **就好像是该运算引发的一样**).

The operator `not in` is defined to have the inverse true value of `in`.

[3]The implementation computes this efficiently, without constructing lists or sorting.

[4]Earlier versions of Python used lexicographic comparison of the sorted (key, value) lists, but this was very expensive for the common case of comparing for equality. An even earlier version of Python compared dictionaries by identity only, but this caused surprises because people expected to be able to test a dictionary for emptiness by comparing it to `{}`.

[5]**目前实现的计算方法并没有构造中间列表和排序**, **因而比较有效率**.

[6]**早些版本的**Python**对排序后的**(**键值对**)**表比较**, **但是这样对于一般等价比较的应用来说非常低效**.**更早的**Python**版本按仅仅依靠字典的标识比较**, **但这时候在你试图用一个空字典和 比较时会得出不期望的结果**.

运算符not in**可计算与运算符**in**相反的结果**.

The operators `is` and `is not` test for object identity: *x* `is` *y* is true if and only if *x* and *y* are the same object. *x* `is not` *y* yields the inverse truth value.

运算符is**和**is not**用于测试对象标识**: x is y **为真**, **当且仅当**x**和**y**是相同的对象**, x is not y**取其反值**.

5.10 布尔运算Boolean operations

Boolean operations have the lowest priority of all Python operations:

布尔运算符在所有Python**运算符中有最低的优先级**:

```
expression   ::=  or_test | lambda_form
or_test      ::=  and_test | or_test "or" and_test
and_test     ::=  not_test | and_test "and" not_test
not_test     ::=  comparison | "not" not_test
lambda_form  ::=  "lambda" [parameter_list]:  expression
```

In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: `None`, numeric zero of all types, empty sequences (strings, tuples and lists), and empty mappings (dictionaries). All other values are interpreted as true.

在布尔运算的上下文, **和控制流语句使用的表达式中**, **以下值解释为假**:None, **所有类型的数值零**,**空的有序类型对象**(**串**, **元组和列表**),**空的映射对象**(**字典**). **所有其它值解释为真**.

The operator `not` yields `1` if its argument is false, `0` otherwise.

如果运算符not**的参数为假**, **它返回**1, **否则返回**0.

The expression *x* `and` *y* first evaluates *x*; if *x* is false, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

表达式x and y**首先计算**x; **如果**x**为假**, **就返回它的值**; **否则**, **计算**y**的值**, **并返回其结果**.

The expression *x* `or` *y* first evaluates *x*; if *x* is true, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

表达式x or y**首先计算**x; **如果**x**为真**, **就返回它的值**; **否则**, **计算**y**的值**, **并返回其结果**.

(Note that neither `and` nor `or` restrict the value and type they return to `0` and `1`, but rather return the last evaluated argument. This is sometimes useful, e.g., if `s` is a string that should be replaced by a default value if it is empty, the expression `s or 'foo'` yields the desired value. Because `not` has to invent a value anyway, it does not bother to return a value of the same type as its argument, so e.g., `not 'foo'` yields `0`, not `''`.)

(**注意**and**和**or**都没有限制它的结果的值和类型必须是**0**或**1, **仅仅是最后一个计算的参数**. **这在某些情况下是有用的**,**例如**, **如果**s**是一个若为空就应该为默认值所替换的串**, **表达式**s or 'foo'**就会得到希望的结果**. **因为**not**根本不会生成一个值**, **就没必要让它的返回值类型与其参数的相同**, **所以**, **例如**, not 'foo' **返回**0, **而不是**")

5.11 Lambda形式(lambda表达式) Lambdas

Lambda forms (lambda expressions) have the same syntactic position as expressions. They are a shorthand to create anonymous functions; the expression `lambda` *arguments*`:` *expression* yields a function object. The unnamed object behaves like a function object defined with

Lambda**形式**(lambda**表达式**)**在句法上与表达式有相同的位置**. **这是一个创建类型函数的快捷方法**;**表达式**lambda arguments: expression**生成一个行为与下面定义的函数一致的函数对象**:

```
def name(arguments):
    return expression
```

See section 7.5 for the syntax of parameter lists. Note that functions created with lambda forms cannot contain statements.

对于参数表句法,**参见**7.5**节**.**注意由**lambda**形式创建的函数不能包括语句**.

5.12 表达式表Expression lists

```
expression_list  ::=  expression ( "," expression )* [","]
```

An expression list containing at least one comma yields a tuple. The length of the tuple is the number of expressions in the list. The expressions are evaluated from left to right.

一个表达式表是一个包括至少一个逗号的元组, **它的长是表中表达式的个数**.**其中表达式从左到右按顺序计算**.

The trailing comma is required only to create a single tuple (a.k.a. a *singleton*); it is optional in all other cases. A single expression without a trailing comma doesn't create a tuple, but rather yields the value of that expression. (To create an empty tuple, use an empty pair of parentheses: `()`.)

最后的逗号仅仅在创建单元素元组(**又称为**"**独元**")**时才需要**; **在其它情况下**, **它是可选的**. **一个没有后缀逗号的表达式不会创建元组**, **但仍会计算该表达式的值**.(**可以使用一对空括号**() **创建一个空元组**).

5.13 Evaluation order

Python evaluates expressions from left to right. Notice that while evaluating an assignment, the right-hand side is evaluated before the left-hand side.

In the following lines, expressions will be evaluated in the arithmetic order of their suffixes:

```
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
func(expr1, expr2, *expr3, **expr4)
expr3, expr4 = expr1, expr2
```

5.14 总结Summary

The following table summarizes the operator precedences in Python, from lowest precedence (least binding) to highest precedence (most binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for comparisons, including tests, which all have the same precedence and chain from left to right — see section 5.9 – and exponentiation, which groups from right to left).

下表总结了Python**中运算符的优先级**, **从低优先级**(**弱捆绑**)**到高优先级**(**强捆绑**), **在同一格子中的运算符具有相同的优先级**.**如果没有特殊的句法指定**, **运算符是二元的**. **同一格子内的运算符都从左至右结合**(**比较运算符是个例外**, **它们可以从左到右连接起来**—- **见上文**, **并且幂运算符也是从左至右结合**).

Operator	Description
`lambda`	Lambda**表达式**Lambda expression
`or`	**布尔**OR Boolean OR
`and`	**布尔**AND Boolean AND
`not` *x*	**布尔**NOT Boolean NOT
`in`, `not in`	**成员测试**Membership tests
`is`, `is not`	**标识测试**Identity tests
<, <=, >, >=, <>, `!=`, `==`	**比较**Comparisons
`|`	**比特级**OR Bitwise OR
`^`	**比特级**XOR Bitwise XOR
`&`	**比特级**AND Bitwise AND
`<<`, `>>`	**移位**Shifts
`+`, `-`	**加减**Addition and subtraction
`*`, `/`, `%`	**乘**,**除**,**取余**Multiplication, division, remainder
`+`*x*, `-`*x*	**正运算**,**负运算**Positive, negative
`~`*x*	**比特级**not Bitwise not
`**`	**幂**Exponentiation
x`.`*attribute*	**属性引用**Attribute reference
x`[`*index*`]`	**下标**Subscription
x`[`*index*`:`*index*`]`	**片断**Slicing
f`(`*arguments*`...)`	**函数调用**Function call
`(`*expressions*`...)`	**表达式分组或元组**Binding or tuple display
`[`*expressions*`...]`	**列表**List display
`{`*key*`:`*datum*`...}`	**字典**Dictionary display
‘*expressions*...‘	**串转换**String conversion

第六章

简单语句Simple statements

Simple statements are comprised within a single logical line. Several simple statements may occur on a single line separated by semicolons. The syntax for simple statements is:

简单语句可以在一个逻辑行中表达. **某些简单语句可以用分号分隔而占据一行**.**简单语句句法如下**:

```
simple_stmt  ::=  expression_stmt
                  | assert_stmt
                  | assignment_stmt
                  | augmented_assignment_stmt
                  | pass_stmt
                  | del_stmt
                  | print_stmt
                  | return_stmt
                  | yield_stmt
                  | raise_stmt
                  | break_stmt
                  | continue_stmt
                  | import_stmt
                  | global_stmt
                  | exec_stmt
```

6.1 表达式语句Expression statements

Expression statements are used (mostly interactively) to compute and write a value, or (usually) to call a procedure (a function that returns no meaningful result; in Python, procedures return the value `None`). Other uses of expression statements are allowed and occasionally useful. The syntax for an expression statement is:

表达式语句用于计算和写一个值(**多用在交互方式下**), **或调用一个过程**(**一个返回没有意义的结果的函数**; **在**Python**中**, **过程返回值**None). **其它表达式语句的使用方法也允许**, **有时也用的到**.**表达式语句的句法如下**:

```
expression_stmt  ::=  expression_list
```

An expression statement evaluates the expression list (which may be a single expression).

一个表达式语句要对该表达表(**可能只是一个表达式**)**求值**.

In interactive mode, if the value is not `None`, it is converted to a string using the built-in `repr()` function and the resulting string is written to standard output (see section 6.6) on a line by itself. (Expression statements yielding `None` are not written, so that procedure calls do not cause any output.)

在交互方式下, **如果其值不为**None, **就使用内建函数**repr()**并将结果串写入标准输出**(**见**6.6**节**), **返回**None**的过程不回写**, **所有过程调用没有任何输出**.)

6.2 断言语句Assert statements

Assert statements are a convenient way to insert debugging assertions into a program:

断言语句是一个在程序中插入调试断言的常用方法:

```
assert_stmt  ::=  "assert" expression ["," expression]
```

The simple form, '`assert expression`', is equivalent to

简单形式的, "assert expression", **等价于**:

```
if __debug__:
   if not expression: raise AssertionError
```

The extended form, '`assert expression1, expression2`', is equivalent to

扩展形式的, "assert expression", **等价于**:

```
if __debug__:
   if not expression1: raise AssertionError, expression2
```

These equivalences assume that `__debug__` and `AssertionError` refer to the built-in variables with those names. In the current implementation, the built-in variable `__debug__` is 1 under normal circumstances, 0 when optimization is requested (command line option -O). The current code generator emits no code for an assert statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be displayed as part of the stack trace.

这些等价式假定了存在__debug__ **和**AssertionError , **而不是具有相同名字的相应内建变量**.**在当前实现**, **内建变量**__debug__**在普通情况下为**1, **在要求优化的情况下为**0(**命令行选项**-O) **在编译要求优化时**, **当前的代码生成器不产生任何断言语句的代码**.**注意在错误信息包括源代码的作法是多余的**; **因为它们会作为跟踪回溯对象的一部分显示**.

Assignments to `__debug__` are illegal. The value for the built-in variable is determined when the interpreter starts.

给__debug__**赋值是非法的**,**解释器是在启动时读取内建变量的值的**.

6.3 赋值语句Assignment statements

Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects:

赋值语句用来把名字(**重新**)**捆绑到值，以及修改可变对象的属性或者项目：**

```
assignment_stmt  ::=  (target_list "=")+ expression_list
target_list      ::=  target ("," target)* [","]
target           ::=  identifier
                      | "(" target_list ")"
                      | "[" target_list "]"
                      | attributeref
                      | subscription
                      | slicing
```

(See section 5.3 for the syntax definitions for the last three symbols.)

(**上面后三项符号的语法定义参看**5.3**节**)

An assignment statement evaluates the expression list (remember that this can be a single expression or a comma-separated list, the latter yielding a tuple) and assigns the single resulting object to each of the target lists, from left to right.

一个赋值语句对表达式序列求值(还记得这可以是单个表达式或者一个逗号分隔的序列，后者导出一个元组)，然后从左到右地将对象结果一一地赋给目的序列的每个对象。

Assignment is defined recursively depending on the form of the target (list). When a target is part of a mutable object (an attribute reference, subscription or slicing), the mutable object must ultimately perform the assignment and decide about its validity, and may raise an exception if the assignment is unacceptable. The rules observed by various types and the exceptions raised are given with the definition of the object types (see section 3.2).

依赖于目标(序列)的形式，赋值被递归地定义。当目标是一个可变对象的一部分(属性引用，下标，片断)的时候，该可变对象必须最终执行该赋值，决定其有效性，并且如果该赋值不可接受可能会抛出一个例外。不同类型以及抛出例外所遵循的规则在该对象类型的定义中给出(见3.2节)。

Assignment of an object to a target list is recursively defined as follows.

一个对象向一个目的序列的赋值递归地定义如下.

- If the target list is a single target: The object is assigned to that target.

  如果目标序列是单个目标，该对象就赋予该目标。

- If the target list is a comma-separated list of targets: The object must be a sequence with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets. (This rule is relaxed as of Python 1.5; in earlier versions, the object had to be a tuple. Since strings are sequences, an assignment like '`a, b = "xy"`' is now legal as long as the string has the right length.)

  如果目标序列是一组用逗号分隔的目标：该对象必须是一个其子项个数与目标序列中的目标个数一样多的有序类型对象，且其子项，从左到右地，逐个赋予相应目标。(这个规则从Python 1.5开始放宽了；在早期版本中，对象必须是一个元组。既然字符串是有序类型对象，像”a,b = ”xy””这样的赋值现在就是合法的，只要该字符串有正确的长度。

Assignment of an object to a single target is recursively defined as follows.

一个对象向单个目标的赋值递归地定义如下。

- If the target is an identifier (name):

  如果该目标是一个标志符（名字）:

  – If the name does not occur in a `global` statement in the current code block: the name is bound to the object in the current local namespace.

    如果该名字不出现于当前代码块的global语句当中：该名字就约束到当前局部名字空间的对象上。

  – Otherwise: the name is bound to the object in the current global namespace.

    否则：该名字约束到当前全局名字空间中的对象。

  The name is rebound if it was already bound. This may cause the reference count for the object previously bound to the name to reach zero, causing the object to be deallocated and its destructor (if it has one) to be called.

  如果名字已经被约束了它就被重新约束。这可能导致早先约束到该名字的对象的引用计数降为零，导致释放该对象的分配空间并调用其析构器，如果它有一个的话。

- If the target is a target list enclosed in parentheses or in square brackets: The object must be a sequence with the same number of items as there are targets in the target list, and its items are assigned, from left to right, to the corresponding targets.

  如果目标是一个用括号或者方括号括起来的目标序列：该对象必须是具有和目标序列中目标个数同样数目的有序类型，且其子项从左到右地赋值给相应目标。

- If the target is an attribute reference: The primary expression in the reference is evaluated. It should yield an object with assignable attributes; if this is not the case, `TypeError` is raised. That object is then asked to assign the assigned object to the given attribute; if it cannot perform the assignment, it raises an exception (usually but not necessarily `AttributeError`).

  如果目标是一个属性引用：引用中的主元表达式被求值。它应该给出一个带可赋值属性的对象；如果不是这种情况，就会抛出TypeError例外。然后那个对象就被要求将被赋值的对象赋值给给定的属性；如果它无法执行该赋值，就会抛出一个例外（通常但是不必然是AttributeError异常）。

- If the target is a subscription: The primary expression in the reference is evaluated. It should yield either a mutable sequence object (e.g., a list) or a mapping object (e.g., a dictionary). Next, the subscript expression is evaluated.

  如果目标是一个下标：引用中的主元表达式被求值。这应给出或者一个可变有序对象(比如，一个列表)或者一个映射对象(比如，一个字典)。接着，下标表达式被求值。

  If the primary is a mutable sequence object (e.g., a list), the subscript must yield a plain integer. If it is negative, the sequence's length is added to it. The resulting value must be a nonnegative integer less than the sequence's length, and the sequence is asked to assign the assigned object to its item with that index. If the index is out of range, `IndexError` is raised (assignment to a subscripted sequence cannot add new items to a list).

  如果主元是可变有序对象(例如列表)，下标必须给出一个普通整数。如果是负数，序列的长度就被加上。最后的值必须是一个小于该序列长度的非负整数，然后该序列就被请求将被赋对象赋值给它带那个指标的项。如果指标超出范围，就会抛出 IndexError**例外(给一个用下标引用的有序对象赋值不会给列表增添新项)。**

  If the primary is a mapping object (e.g., a dictionary), the subscript must have a type compatible with the mapping's key type, and the mapping is then asked to create a key/datum pair which maps the subscript to the assigned object. This can either replace an existing key/value pair with the same key value, or insert a new key/value pair (if no key with the same value existed).

  如果主元是一个映射对象（比如字典），下标的类型必须和映射的键类型兼容，接着该映射就被要求创建一个把下标映射到被赋对象的键/数据对。这(操作)要不用新键值取代已存在的具有相同键的键/值对，要不插入一个新的键/值对(如果不存在相同的键)。

- If the target is a slicing: The primary expression in the reference is evaluated. It should yield a mutable sequence object (e.g., a list). The assigned object should be a sequence object of the same type. Next, the lower and upper bound expressions are evaluated, insofar they are present; defaults are zero and the sequence's length. The bounds should evaluate to (small) integers. If either bound is negative, the sequence's length is added to it. The resulting bounds are clipped to lie between zero and the sequence's length, inclusive. Finally, the sequence object is asked to replace the slice with the items of the assigned sequence. The length of the slice may be different from the length of the assigned sequence, thus changing the length of the target sequence, if the object allows it.

  如果目标是一片断：引用中的主元表达式被求值。这应给出一个可变有序对象(比如列表)。被赋值对象应该是同一类型的有序对象。下一步，在它们所出现的范围内，对上下限表达式求值；缺省值是零和序列长度。限值应求值为(小)整数。如果任一限是负的，就加上序列的长度。结果限值被修整至零和序列长度之间(含零和序列长度)。最后，有序对象被要求用被赋有序对象的子项替换该片断。片断的长度可能和被赋序列的长度不同，那就改变目标序列的长度，如果该对象允许的话。

(In the current implementation, the syntax for targets is taken to be the same as for expressions, and invalid syntax is rejected during the code generation phase, causing less detailed error messages.)

(在当前的实现中，目标对象的语法被认为和表达式的语法相同，并且非法语法在代码生成期被拒绝，这导致缺少详细的错误信息)

WARNING: Although the definition of assignment implies that overlaps between the left-hand side and the right-hand side are 'safe' (e.g., '`a, b = b, a`' swaps two variables), overlaps *within* the collection of assigned-to variables are not safe! For instance, the following program prints '`[0, 2]`':

警告：虽然赋值的定义隐含着左手边和右手边之间的重叠是“安全的”(比如，”a, b = b, a”**交换两个变量)，在所赋值变量间的重叠却是不安全的！例如，下面的程序打印出**”[0, 2]”**：**

```
x = [0, 1]
i = 0
i, x[i] = 1, 2
print x
```

6.3.1 增量赋值语句Augmented assignment statements

Augmented assignment is the combination, in a single statement, of a binary operation and an assignment statement:

增量赋值就是在单条语句内合并一个二元运算和一个赋值语句。

```
augmented_assignment_stmt  ::=  target augop expression_list
augop                      ::=  "+=" | "-=" | "*=" | "/=" | "%=" | "**="
                                | ">>=" | "<<=" | "&=" | "^=" | "|="
```

(See section 5.3 for the syntax definitions for the last three symbols.)

(**最后三项符号的语法定义见**5.3**节**)

An augmented assignment evaluates the target (which, unlike normal assignment statements, cannot be an unpacking) and the expression list, performs the binary operation specific to the type of assignment on the two operands, and assigns the result to the original target. The target is only evaluated once.

一条增量赋值语句对目标(和一般的赋值语句不同，它不能是展开的对象)和表达式列表求值，执行特定于两个操作数的赋值类型的二元运算，并将结果赋值给原先的目标。目标仅求值一次。

An augmented assignment expression like `x += 1` can be rewritten as `x = x + 1` to achieve a similar, but not exactly equal effect. In the augmented version, `x` is only evaluated once. Also, when possible, the actual operation is performed *in-place*, meaning that rather than creating a new object and assigning that to the target, the old object is modified instead.

一条赋值语句，比如x+= 1, **可以重写为**x = x + 1**，效果是类似的，但并不完全一样。在增量版本中，**x**仅求值一次。而且，只要可能，实际的操作是就地进行的，意思是并非创建一个新对象然后将其赋值给目标，而是修改老的对象。**

With the exception of assigning to tuples and multiple targets in a single statement, the assignment done by augmented assignment statements is handled the same way as normal assignments. Similarly, with the exception of the possible *in-place* behavior, the binary operation performed by augmented assignment is the same as the normal binary operations.

除了在一条语句中赋值给元组和多个对象的情况，增量赋值语句所完成的赋值用与普通赋值同样的方式处理。类似地，除了可能的就地方式，由增量赋值执行的二元运算和普通的二元运算也是一样的。

For targets which are attribute references, the initial value is retrieved with a `getattr()` and the result is assigned with a `setattr()`. Notice that the two methods do not necessarily refer to the same variable. When `getattr()` refers to a class variable, `setattr()` still writes to an instance variable. For example:

```
class A:
    x = 3    # class variable
a = A()
a.x += 1     # writes a.x as 4 leaving A.x as 3
```

6.4 pass语句The pass statement

```
pass_stmt  ::=  "pass"
```

`pass` is a null operation — when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed, for example:

pass **是一个空操作**—- **当执行它**, **什么也不做**. **它在句法上要求有一个语句时但不需要写代码时用到**,**例如**:

```
def f(arg): pass    # a function that does nothing (yet)

class C: pass       # a class with no methods (yet)
```

6.5 del 语句The del statement

```
del_stmt  ::=  "del" target_list
```

Deletion is recursively defined very similar to the way assignment is defined. Rather that spelling it out in full details, here are some hints.

删除与赋值的定义方法类似, 也是递归的. 下面是一些的说明:

Deletion of a target list recursively deletes each target, from left to right.

一个目标表的递归删除操作会从左到右地删除其中的每个对象地.

Deletion of a name removes the binding of that name from the local or global namespace, depending on whether the name occurs in a `global` statement in the same code block. If the name is unbound, a `NameError` exception will be raised.

删除名字就是在局部名字空间和全局名字空间删除掉该名字的绑定(必须存在), 从哪个名字空间删除决定于该名字是否出现在其代码块的globals**语句中.**

It is illegal to delete a name from the local namespace if it occurs as a free variable in a nested block.

Deletion of attribute references, subscriptions and slicings is passed to the primary object involved; deletion of a slicing is in general equivalent to assignment of an empty slice of the right type (but even this is determined by the sliced object).

对于属性引用, 下标和片断的删除会作用到相关的主元对象, 对片断的删除一般等价于对该片断赋予相应类型的空片断(但这也受被截为片断的对象的限制).

6.6 print语句The print statement

```
print_stmt  ::=  "print" ( [expression ("," expression)* [","]]
                 | ">>" expression [("," expression)+ [","]] )
```

`print` evaluates each expression in turn and writes the resulting object to standard output (see below). If an object is not a string, it is first converted to a string using the rules for string conversions. The (resulting or original) string is then written. A space is written before each object is (converted and) written, unless the output system believes it is positioned at the beginning of a line. This is the case (1) when no characters have yet been written to standard output, (2) when the last character written to standard output is '`\n`', or (3) when the last write operation on standard output was not a `print` statement. (In some cases it may be functional to write an empty string to standard output for this reason.) **Note:** Objects which act like file objects but which are not the built-in file objects often do not properly emulate this aspect of the file object's behavior, so it is best not to rely on this.

print**依次对每个表达式求值，然后把结果对象写入标准输出(见下面)。如果一个对象不是一个字符串，首先用字符串转换法则将其转成一个字符串。然后(结果的或者原来的)字符串就被写入。在每个对象被(转换和)写入以前，输出系统会先写一个空格，除非它认为当前处在一行的起首位置。这里是(起首)情形：**(1)**当尚未有字符写到标准输出,**(2)**当上一个写入标准输出的字符是**'\n'**，或者**(3)**当前一次作用于标准输出上的操作不是**print**语句。注意：那些行为类似**file**对象但是并非内置**file**对象的对象常常不能正确地模拟**file**对象这方面的行为，所以最好不要依赖于此。**

A '`\n`' character is written at the end, unless the `print` statement ends with a comma. This is the only action if the statement contains just the keyword `print`.

A '`\n`' character is written at the end, unless the print statement ends with a comma. This is the only action if the statement contains just the keyword print.

Standard output is defined as the file object named `stdout` in the built-in module `sys`. If no such object exists, or if it does not have a `write()` method, a `RuntimeError` exception is raised.

标准输出定义为内置模块sys**中名为**stdout**的**file**文件对象。如果该对象不存在，或者它没有**write()**方法，会抛出一个**RuntimeError**异常。**

`print` also has an extended form, defined by the second portion of the syntax described above. This form is sometimes referred to as "`print` chevron." In this form, the first expression after the >> must evaluate to a "file-like” object, specifically an object that has a `write()` method as described above. With this extended form, the subsequent expressions are printed to this file object. If the first expression evaluates to `None`, then `sys.stdout` is used as the file for output.

print**也有一个扩展形式，由上面描述的语法的第二部分定义。这个形式有时被称为“返身打印**(print chevron)**”。在该形式中，跟在**”¿¿”**后的第一个表达式必须取值为一个**”**类文件**”**的对象，确切地说就是一个具有上面所描述的**write()**方法的对象。以这个扩展形式，后续表达式被打印到该文件对象上。如果第一个表达式取值为**None**，**sys.stdout**对象就被用来作为输出文件。**

6.7 return**语句**The `return` statement

```
return_stmt  ::=  "return" [expression_list]
```

`return` may only occur syntactically nested in a function definition, not within a nested class definition.

return**在句法上仅可以出现在嵌套的函数定义中**, **不能出现在嵌套的类定义中**.

If an expression list is present, it is evaluated, else `None` is substituted.

如果给出了表达式表, **就计算其值**, **否则就代以**None.

`return` leaves the current function call with the expression list (or `None`) as return value.

return**的作用是离开当前函数调用**, **并以表达式表的值**(**或**None)**为返回值**.

When `return` passes control out of a `try` statement with a `finally` clause, that `finally` clause is executed before really leaving the function.

当return**放在具有**finally**子句的**try**语句中**, finally**子句中的语句会在函数真正退出之前执行一次**.

In a generator function, the `return` statement is not allowed to include an `expression_list`. In that context, a bare `return` indicates that the generator is done and will cause `StopIteration` to be raised.

在生成器函数中, return**语句不允许包括**expression_list. **在该种情况下**, **空的**return**语句指出生成器结束**,**并引发一个**StopIteration**异常**.

6.8 yield**语句**The `yield` statement

```
yield_stmt  ::=  "yield" expression_list
```

The `yield` statement is only used when defining a generator function, and is only used in the body of the generator function. Using a `yield` statement in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

yield**语句仅当定义生成器函数的时候使用，也只能用于生成器函数体中。在一个函数定义中使用**yield**语句足以导致该定义产生一个生成器函数而不是普通函数。**

When a generator function is called, it returns an iterator known as a generator iterator, or more commonly, a generator. The body of the generator function is executed by calling the generator’s `next()` method repeatedly until it raises an exception.

当生成器函数被调用的时候，它返回一个迭代器，称为生成器迭代器，或者更常用的，生成器。通过重复地调用生成器的next()**方法来运行生成器的函数体，直到抛出一个异常。**

When a `yield` statement is executed, the state of the generator is frozen and the value of `expression_list` is returned to `next()`’s caller. By “frozen” we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, and the internal evaluation stack: enough information is saved so that the next time `next()` is invoked, the function can proceed exactly as if the `yield` statement were just another external call.

当一个yield**语句被执行的时刻，该生成器的状态就被冻结起来，而表达式表的值则被返回给**next()**方法的调用者。所谓“冻结”我们指的是所有局部状态都被保持，包括局部变量的当前约束，**(**指向下条的**)**指令指针，内部的求值堆栈：保留了充分多的信息，使得当下次激活**next()**的时候，函数执行起来完全好像**yield**语句不过是另外一个外部调用。**

The `yield` statement is not allowed in the `try` clause of a `try` ... `finally` construct. The difficulty is that there's no guarantee the generator will ever be resumed, hence no guarantee that the `finally` block will ever get executed.

yield语句不允许出现于try ... finally**结构的**try**子句当中。困难之处在于没有保证**generator**会被继续执行**,**于是也就没有保证**finally**块会被执行。**

Note: In Python 2.2, the `yield` statement is only allowed when the `generators` feature has been enabled. It will always be enabled in Python 2.3. This `__future__` import statment can be used to enable the feature:

注意：在 Python 2.2**，**yield**语句只有在生成器特性被激活以后才成立。在**Python 2.3**中它将总是激活的。下面这个**__future__ import**语句可用于激活该特性**:

```
from __future__ import generators
```

See Also:

PEP 0255, "***简单生成器**Simple Generators*"
  为Python**添加生成器和**yield**语句的提案**The proposal for adding generators and the `yield` statement to Python.

6.9 raise**语句**The `raise` statement

```
raise_stmt  ::=  "raise" [expression ["," expression ["," expression]]]
```

If no expressions are present, `raise` re-raises the last expression that was active in the current scope. If no exception is active in the current scope, an exception is raised indicating this error.

如果不给出表达式(expressions)**，**raise**重新引发当前范围内上次引发的表达式。**

Otherwise, `raise` evaluates the expressions to get three objects, using `None` as the value of omitted expressions. The first two objects are used to determine the *type* and *value* of the exception.

否则，raise**对其后的第一个表达式求值，该值必须给出字符串，类，或者实例对象。如果还有第二个表达式，也被求值，否则以**None**代之。如果第一表达式是一个类对象，那么第二表达式可以是该类或者其衍生类的一个实例，然后该实例就被引发。如果第二表达式不是这样的实例，那么给出的类就会被实例化。实例化的参量列表按下列方式决定：如果第二表达式是一元组，就用来当作参量列表；如果是**None**，参量列表就为空；否则，参量列表仅包含单个参量，就是第二个表达式。如果第一表达式是一个实例对象，第二表达式必须是**None**。**

If the first object is an instance, the type of the exception is the class of the instance, the instance itself is the value, and the second object must be `None`.

如果第一对象是一个字符串，它就引发一个由第一对象标示的，以第二对象(**或**None)**为参数的异常。如果第一对象是一个类或者实例，就会引发由前述所决定的实例的类所标示的，以该实例为参数的异常。**

If the first object is a class, it becomes the type of the exception. The second object is used to determine the exception value: If it is an instance of the class, the instance becomes the exception value. If the second object is a tuple, it is used as the argument list for the class constructor; if it is `None`, an empty argument list is used, and any other object is treated as a single argument to the constructor. The instance so created by calling the constructor is used as the exception value.

If a third object is present and not `None`, it must be a traceback object (see section 3.2), and it is substituted instead of the current location as the place where the exception occurred. If the third object is present and not a traceback object or `None`, a `TypeError` exception is raised. The three-expression form of `raise` is useful to re-raise an exception transparently in an except clause, but `raise` with no expressions should be preferred if the exception to be re-raised was the most recently active exception in the current scope.

如果给出了第三对象，且它不是 None**，它就应该为一个回溯对象**(**见**3.2**节**)**，且用其替代当前位置作为异常发生的地点。这可用于透明地在异常子句中重新引发一个异常。**

Additional information on exceptions can be found in section 4.2, and information about handling exceptions is in section 7.4.

6.10 break语句The `break` statement

```
break_stmt  ::=  "break"
```

`break` may only occur syntactically nested in a `for` or `while` loop, but not nested in a function or class definition within that loop.

break在句法上只能出现在for或while循环中, 但不能出现在循环中的函数定义或类定义中.

It terminates the nearest enclosing loop, skipping the optional `else` clause if the loop has one.

它中断最内层的循环, 跳过其可选的else语句(如果有的话).

If a `for` loop is terminated by `break`, the loop control target keeps its current value.

如果for循环被break中断, 它的循环控制对象还保持当前值,

When `break` passes control out of a `try` statement with a `finally` clause, that `finally` clause is executed before really leaving the loop.

当break放在具有finally子句的try语句中, finally子句中的语句会在循环真正退出之前执行一次.

6.11 continue语句The `continue` statement

```
continue_stmt  ::=  "continue"
```

`continue` may only occur syntactically nested in a `for` or `while` loop, but not nested in a function or class definition or `try` statement within that loop.[1] It continues with the next cycle of the nearest enclosing loop.

continue在句法上只能出现在for或while循环中, 但不能出现在循环中的函数定义或类定义中.6.1它重新开始最内层的循环. [2]

6.12 import 语句The `import` statement

```
import_stmt  ::=  "import" module ["as" name] ( "," module ["as" name] )*
                  | "from" module "import" identifier ["as" name]
                   ( "," identifier ["as" name] )*
                  | "from" module "import" "*"
module       ::=  (identifier ".")* identifier
```

Import statements are executed in two steps: (1) find a module, and initialize it if necessary; (2) define a name or names in the local namespace (of the scope where the `import` statement occurs). The first form (without `from`) repeats these steps for each identifier in the list. The form with `from` performs step (1) once, and then performs step (2) repeatedly.

import语句分两步执行：(1) 找到模块，如果需要则进行初始化；(2) 在(import语句所发生的范围内)局部名字空间中定义一个或者多个名字。(import语句的)第一形式(不带import那个)对列表中的每个标示符重复这些步骤。带import的形式只执行步骤(1)一次，然后重复地执行步骤(2)。

In this context, to "initialize" a built-in or extension module means to call an initialization function that the module must provide for the purpose (in the reference implementation, the function's name is obtained by prepending string "init" to the module's name); to "initialize" a Python-coded module means to execute the module's body.

The system maintains a table of modules that have been or are being initialized, indexed by module name. This table is accessible as `sys.modules`. When a module name is found in this table, step (1) is finished. If not, a search for a module definition is started. When a module is found, it is loaded. Details of the module searching

[1]It may occur within an `except` or `else` clause. The restriction on occurring in the `try` clause is implementor's laziness and will eventually be lifted.

[2]它可以出现在子句中, 限制在try中的出现是由于实现者的缓慢并且最终会实现的.

and loading process are implementation and platform specific. It generally involves searching for a “built-in” module with the given name and then searching a list of locations given as `sys.path`.

系统维护一个已初始化的，由其名字索引的模块表。这个表可经由sys.modules**访问。如果一个模块的名字可于该表找到，**(**该模块的**)**步骤**(1)**就已经结束了。如果没有，就开始搜索该模块的定义。一旦找到模块，它就会被载入**(**系统**)**。模块搜索和加载的详细过程取决于特定的实现和平台。一般地它涉及搜索给定名字的内置模块然后搜索由**sys.path**给出的位置列表。**

If a built-in module is found, its built-in initialization code is executed and step (1) is finished. If no matching file is found, `ImportError` is raised. If a file is found, it is parsed, yielding an executable code block. If a syntax error occurs, `SyntaxError` is raised. Otherwise, an empty module of the given name is created and inserted in the module table, and then the code block is executed in the context of this module. Exceptions during this execution terminate step (1).

如果找到内置模块，就执行其内置的预置代码，然后步骤(1)**就完成了。如果找不到匹配的文件，就引发**ImportError**异常。如果文件找到，它就会被分析，导致一段可执行代码块。如果有语法错误，就引发**SyntaxError**异常。否则，就用给定的名字创建一个空模块并插入模块表中，然后在这个模块的环境中运行所得代码块。在这一运行过程中引发的异常将中止步骤**(1)**。**

When step (1) finishes without raising an exception, step (2) can begin.

如果步骤(1)**无任何异常地结束了，步骤**(2)**就可以开始了**

The first form of `import` statement binds the module name in the local namespace to the module object, and then goes on to import the next identifier, if any. If the module name is followed by `as`, the name following `as` is used as the local name for the module.

import**语句的第一形式于当前局部名字空间中将模块名字约束到该模块对象上去，然后继续导入下个标识符，如果还有的话。如果模块名字后面带**as**，**as**后面跟着的名字就被用作该模块的局部名。为避免混淆，你不能把带点的模块名字导入为另外一个不同的名字。所以**import module as m**是合法的，但是**import module.submod as s**就不是。后者应写成**from module import submod as s**。参看下面所述。**

The `from` form does not bind the module name: it goes through the list of identifiers, looks each one of them up in the module found in step (1), and binds the name in the local namespace to the object thus found. As with the first form of `import`, an alternate local name can be supplied by specifying ”`as` localname”. If a name is not found, `ImportError` is raised. If the list of identifiers is replaced by a star (‘`*`’), all public names defined in the module are bound in the local namespace of the `import` statement..

带from**的导入形式不约束模块的名字：它遍历其标识符列表，在由步骤**(1)**找到的模块中查找其中的每一个**(**所对应的对象**)**，然后把局部空间中的名字约束到找到的对象上去。类似**import**的第一式，通过指定**”as**局部名**”**的形式，可以给出一个替代名字。如果某个名字找不到，就会引发**ImportError**异常。如果标识符列表用一个星号**(”*”)**取代，所有定义于该模块中的公共名字都在**import**语句所在的局部名字空间中被约束。**

The *public names* defined by a module are determined by checking the module’s namespace for a variable named `__all__`; if defined, it must be a sequence of strings which are names defined or imported by that module. The names given in `__all__` are all considered public and are required to exist. If `__all__` is not defined, the set of public names includes all names found in the module’s namespace which do not begin with an underscore character (‘_’). `__all__` should contain the entire public API. It is intended to avoid accidentally exporting items that are not part of the API (such as library modules which were imported and used within the module).

一个模块所定义的“公共名字”通过检查该模块的名字空间中的名为__all__**的变量决定。如果**(**该变量**)**有定义，它必须是一个字符串的有序序列，这些字符串是由该模块定义或者导入的名字。在**__all__**中给出的名字都被认为是公共的且要求其存在。如果**__all__**没有定义，**(**该模块的**)**公共名字的集合就包含所有在该模块的名字空间中找到的，不以下划线**(”_”)**起首的所有名字。**

The `from` form with ‘`*`’ may only occur in a module scope. If the wild card form of import — ‘`import *`’ — is used in a function and the function contains or is a nested block with free variables, the compiler will raise a `SyntaxError`.

带*的**from**形式只能在模块的范围中发生。**

Hierarchical module names: when the module names contains one or more dots, the module search path is carried out differently. The sequence of identifiers up to the last dot is used to find a “package”; the final identifier is then searched inside the package. A package is generally a subdirectory of a directory on `sys.path` that has a file '__init__.py'. [XXX Can't be bothered to spell this out right now; see the URL http://www.python.org/doc/essays/packages.html for more details, also about how the module search works from inside a package.]

有层次的模块名字：当模块名字包含一个或多个小数点时，模块的搜索路径执行起来有所不同。(从头一直)到最后一个小数点的标识符序列被用来找到一个"包"；然后在该包中搜索最末的那个标识符。包一般而言是在sys.path中的目录的一个有文件__init__.py的子目录。【XXX 现在在这里不能把这些都详细地写出来，更多的细节，以及关于包内模块搜索如何工作的详情，请见URLhttp://www.python.org/doc/essays/packages.html】

The built-in function `__import__()` is provided to support applications that determine which modules need to be loaded dynamically; refer to Built-in Functions in the *Python Library Reference* for additional information.

提供了内置函数__import__()使得应用程序可动态地决定需要加载哪些模块，更多信息请参考Python 库参考中的内建函数。

6.12.1 Future statements

A *future statement* is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python. The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language. It allows use of the new features on a per-module basis before the release in which the feature becomes standard.

```
future_statement  ::=  "from" "__future__" "import" feature ["as" name]
                       ("," feature ["as" name])*
feature           ::=  identifier
name              ::=  identifier
```

A future statement must appear near the top of the module. The only lines that can appear before a future statement are:

- the module docstring (if any),
- comments,
- blank lines, and
- other future statements.

The features recognized by Python 2.3 are '`generators`', '`division`' and '`nested_scopes`'. '`generators`' and '`nested_scopes`' are redundant in 2.3 because they are always enabled.

A future statement is recognized and treated specially at compile time: Changes to the semantics of core constructs are often implemented by generating different code. It may even be the case that a new feature introduces new incompatible syntax (such as a new reserved word), in which case the compiler may need to parse the module differently. Such decisions cannot be pushed off until runtime.

For any given release, the compiler knows which feature names have been defined, and raises a compile-time error if a future statement contains a feature not known to it.

The direct runtime semantics are the same as for any import statement: there is a standard module `__future__`, described later, and it will be imported in the usual way at the time the future statement is executed.

The interesting runtime semantics depend on the specific feature enabled by the future statement.

Note that there is nothing special about the statement:

```
import __future__ [as name]
```

That is not a future statement; it's an ordinary import statement with no special semantics or syntax restrictions.

Code compiled by an exec statement or calls to the builtin functions `compile()` and `execfile()` that occur in a module `M` containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can, starting with Python 2.2 be controlled by optional arguments to `compile()` — see the documentation of that function in the library reference for details.

A future statement typed at an interactive interpreter prompt will take effect for the rest of the interpreter session. If an interpreter is started with the **-i** option, is passed a script name to execute, and the script includes a future statement, it will be in effect in the interactive session started after the script is executed.

6.13 global**语句**The `global` statement

```
global_stmt  ::=  "global" identifier ("," identifier)*
```

The `global` statement is a declaration which holds for the entire current code block. It means that the listed identifiers are to be interpreted as globals. It would be impossible to assign to a global variable without `global`, although free variables may refer to globals without being declared global.

global**语句是对整个代码块都有作用的一个声明**. **它指出其所列的标识符要解释为全局的**.**如果某名字在局部名字空间中没有定义**, **就自动使用相应的全局名字**. **没有**global**是不可能手动指定一个名字是全局的**.

Names listed in a `global` statement must not be used in the same code block textually preceding that `global` statement.

在global **中出现的名字不能在**global **之前的代码中使用**.

Names listed in a `global` statement must not be defined as formal parameters or in a `for` loop control target, `class` definition, function definition, or `import` statement.

在global **中出现的名字不能作为形参**, **不能作为循环的控制对象**, **不能在类定义**, **函数定义**, import**语句中出现**.

(The current implementation does not enforce the latter two restrictions, but programs should not abuse this freedom, as future implementations may enforce them or silently change the meaning of the program.)

(**当前实现不强制执行后两种限制**, **但程序不能滥用这种自由**, **以后的实现可能会对其强行限制或者可能会改变程序行为而没有任何提示**.)

Programmer's note: the `global` is a directive to the parser. It applies only to code parsed at the same time as the `global` statement. In particular, a `global` statement contained in an `exec` statement does not affect the code block *containing* the `exec` statement, and code contained in an `exec` statement is unaffected by `global` statements in the code containing the `exec` statement. The same applies to the `eval()`, `execfile()` and `compile()` functions.

程序员注意: global**是一个解析器的指示字**. **它仅仅会对和**global**一起解析的代码有效**.**特别地**, exec**语句中的**global**语句不会对包括有该**exec**语句的代码块产生影响**. **并且**exec**语句中的**global**语句也不对**exec**语句中的代码产生影响**. **相同的机制也应用于**eval(), execfile() **和**compile() **函数**.

6.14 exec **语句**The `exec` statement

```
exec_stmt  ::=  "exec" expression ["in" expression ["," expression]]
```

This statement supports dynamic execution of Python code. The first expression should evaluate to either a string, an open file object, or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs). If it is an open file, the file is parsed until EOF and executed. If it is a code object, it is simply executed.

该语句支持Python **代码的动态执行。第一个表达式应求值为一个字符串**, **一个打开的文件对象**, **或是一个代码对象**. **如果它是字符串**, **该字符串被分析为一组**Python**的语句，并随后运行**(**除非发生语法错误**)**。如果是一打开的文件**, **则分析该文件直到文件末尾，然后运行。如果是一代码对象，则简单地运行它。**

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only the first expression after `in` is specified, it should be a dictionary, which will be used for both the global and the local variables. If two expressions are given, both must be dictionaries and they are used for the global and local variables, respectively.

在所有情况下，如果省略了可选部分，代码就在当前的范围中运行。如果只给出了in **后面的表达式，它**

应为一个字典，并用于全局和局部的变量。如果给出了两个表达式, 两者都应是字典，且它们分别用于全局和局部变量。

As a side effect, an implementation may insert additional keys into the dictionaries given besides those corresponding to variable names set by the executed code. For example, the current implementation may add a reference to the dictionary of the built-in module `__builtin__` under the key `__builtins__` (!).

作为一种副作用，除了那些对应于由被执行代码所设置的变量的对象外，一种实现可能会向字典中加入额外的键。例如，当前的实现可能会向字典中加入__builtin__ **键名下的一个指向内建模块字典**__builtin__ **的指针**(!)**。**

Programmer's hints: dynamic evaluation of expressions is supported by the built-in function `eval()`. The built-in functions `globals()` and `locals()` return the current global and local dictionary, respectively, which may be useful to pass around for use by `exec`.

程序员提示: **表达式的动态执行由内建函数**eval() **支持。内建函数**globals() **和**locals() **分别返回当前的全局和局部字典，可用于传递给**exec**使用。**

第七章

复合语句Compound statements

Compound statements contain (groups of) other statements; they affect or control the execution of those other statements in some way. In general, compound statements span multiple lines, although in simple incarnations a whole compound statement may be contained in one line.

复合语句由其它语句(组)构成; 它某种方式影响或控制其它语句的执行.一般地, 复合语句跨越多个行, 但是一个完整的复合语句也可以简化在一行中.

The `if`, `while` and `for` statements implement traditional control flow constructs. `try` specifies exception handlers and/or cleanup code for a group of statements. Function and class definitions are also syntactically compound statements.

if ,while**和**for**语句实现的传统的控制流机制**. try**语句为一组语句指定异常处理器和/或清除代码.函数定义和类定义在句法上也被看作复合语句**.

Compound statements consist of one or more 'clauses.' A clause consists of a header and a 'suite.' The clause headers of a particular compound statement are all at the same indentation level. Each clause header begins with a uniquely identifying keyword and ends with a colon. A suite is a group of statements controlled by a clause. A suite can be one or more semicolon-separated simple statements on the same line as the header, following the header's colon, or it can be one or more indented statements on subsequent lines. Only the latter form of suite can contain nested compound statements; the following is illegal, mostly because it wouldn't be clear to which `if` clause a following `else` clause would belong:

复合语句由一个或多个"子句"组成.一个子句由一个头和一个"代码序列"组成.一个具体的复合语句的所有子句具有相同的缩进层次.每个子句头以一个唯一的标识关键字开始, 并以一个冒号结果.一个语句序列, 是由该子句所控制的一组语句, 一个语句序列可以包括一个或多个分号— 与子句头同行的一串简单语句;或者它可以是以随后的各行中缩进的语句. 仅在后者的情况下子句序列允许包括有嵌套的复合语句, 下面这样是非法的, 这样处理大部分原因是如果其后面有else**子句的话, 语义就不太清晰了**.

```
if test1: if test2: print x
```

Also note that the semicolon binds tighter than the colon in this context, so that in the following example, either all or none of the `print` statements are executed:

也要注意在这样的上下文中, 分号的优先级比冒号的高, 所以在下面的例子中, 要么执行全部的print**语句, 要么一个也不执行**:

```
if x < y < z: print x; print y; print z
```

Summarizing:

总结:

```
compound_stmt  ::=  if_stmt
                    | while_stmt
                    | for_stmt
                    | try_stmt
                    | funcdef
                    | classdef
suite          ::=  stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement      ::=  stmt_list NEWLINE | compound_stmt
stmt_list      ::=  simple_stmt (";" simple_stmt)* [";"]
```

Note that statements always end in a `NEWLINE` possibly followed by a `DEDENT`. Also note that optional continuation clauses always begin with a keyword that cannot start a statement, thus there are no ambiguities (the 'dangling `else`' problem is solved in Python by requiring nested `if` statements to be indented).

注意以NEWLINE**结尾的语句可能后缀一个**DEDENT. **同时注意到可选的续行子句通常以不能开始一个语句的某个关键字开始**, **因此这里没有歧义**(”**不定的**else”**问题已经由**Python**对嵌套的语句要求缩进而解决掉了**).

The formatting of the grammar rules in the following sections places each clause on a separate line for clarity.

为了叙述清楚, **以下章节中的每个子句的语法规则格式都被分行说明**.

7.1 if **语句**The `if` statement

The `if` statement is used for conditional execution:

if **语句用于条件性执行**:

```
if_stmt  ::=  "if" expression ":" suite
              ( "elif" expression ":" suite )*
              ["else" ":" suite]
```

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true (see section 5.10 for the definition of true and false); then that suite is executed (and no other part of the `if` statement is executed or evaluated). If all expressions are false, the suite of the `else` clause, if present, is executed.

它对表达式逐个求值, **直到其中一个为真时**, **准确地选择相应的一个语句序列**.(**对于真和假的定义参见**5.10**节**); **然后该执行语句序列**(if**语句的其它部分不会被执行和计算**).**如果所有表达式都为假**, **并且给出了**else**子句**,**那么将执行它包括的语句序列**.

7.2 while**语句**The `while` statement

The `while` statement is used for repeated execution as long as an expression is true:

while**用于控制重复执行**, **只要满足条件表达式为真**:

```
while_stmt  ::=  "while" expression ":" suite
                 ["else" ":" suite]
```

This repeatedly tests the expression and, if it is true, executes the first suite; if the expression is false (which may be the first time it is tested) the suite of the `else` clause, if present, is executed and the loop terminates.

while**会重复地计算表达式的值**, **并且如果为真**,**就执行第一个语句序列**; **如果为假**(**可能在第一次比较时**),**就执行**else**子句**(**如果给出**), **并退出循环**.

A `break` statement executed in the first suite terminates the loop without executing the `else` clause's suite. A `continue` statement executed in the first suite skips the rest of the suite and goes back to testing the expression.

在第一个语句序列中的break**语句可以实现不执行**else**子句而退出循环**.**在第一个语句序列中的**continue**语句可以跳过该子句的其余部分**, **直接进行下次的表达式的测试**.

7.3 for语句The `for` statement

The `for` statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

for语句用于迭代有序类型或其它可迭代对象的元素(像串, 元组或列表):

```
for_stmt  ::=  "for" target_list "in" expression_list ":" suite
               ["else" ":" suite]
```

The expression list is evaluated once; it should yield a sequence. The suite is then executed once for each item in the sequence, in the order of ascending indices. Each item in turn is assigned to the target list using the standard rules for assignments, and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty), the suite in the `else` clause, if present, is executed, and the loop terminates.

表达式表仅被计算一次, 它应该生成一个有序类型对象.语句序列对每个有序类型对象的元素按索引升序执行一次.每个元素使用标准的赋值规则依次赋给循环控制对象表, 然后执行语句序列.当迭代完毕后(当有序类型对象为空立即结束循环), 就执行else子句(如果给出), 然后循环结束.

A `break` statement executed in the first suite terminates the loop without executing the `else` clause's suite. A `continue` statement executed in the first suite skips the rest of the suite and continues with the next item, or with the `else` clause if there was no next item.

在第一个语句序列中的break语句可以实现不执行else子句而退出循环.在第一个语句序列中的continue语句可以跳过该子句的其余部分, 直接进行下个元素的计算, 或者当迭代完毕后进入else子句.

The suite may assign to the variable(s) in the target list; this does not affect the next item assigned to it.

语句序列可以对循环控制对象表中的变量赋值, 这不影响for语句赋下一项元素给它.

The target list is not deleted when the loop is finished, but if the sequence is empty, it will not have been assigned to at all by the loop. Hint: the built-in function `range()` returns a sequence of integers suitable to emulate the effect of Pascal's `for i := a to b do`; e.g., `range(3)` returns the list `[0, 1, 2]`.

在循环结束后,这个循环控制对象表没有删除, 但是如果有序类型对象为空, 它在循环根本就不会被该有序类型对象赋值.小技巧: 内建函数range()返回一个整数列表, 可以用于模拟Pascal语言中的for i := a to b的行为, 例如range返回在列表[0, 1, 2].

Warning: There is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, i.e. lists). An internal counter is used to keep track of which item is used next, and this is incremented on each iteration. When this counter has reached the length of the sequence the loop terminates. This means that if the suite deletes the current (or a previous) item from the sequence, the next item will be skipped (since it gets the index of the current item which has already been treated). Likewise, if the suite inserts an item in the sequence before the current item, the current item will be treated again the next time through the loop. This can lead to nasty bugs that can be avoided by making a temporary copy using a slice of the whole sequence, e.g.,

警告:如果在循环要修改有序类型对象(仅对可变类型而言, 即列表)的话, 这里有一些要注意的地方.有一个内部计数器用于跟踪下一轮循环使用哪一个元素, 并且每次迭代就增加一次.当这个计数器到达有序类型对象的长度时该循环就结束了.这意味着如果语句序删除一个当前元素(或前一个元素)时, 下一个元素会被跳过去(因为当前索引值的元素已经处理过了).另一方面, 如果在当前元素前插入一个元素, 则当前元素会下一轮循环被再次重复处理.这可能会导致难以觉察的错误.但可以通过使用含有整个有序类型对象的片断而生成的临时拷贝避免这个问题, 例如,

```
for x in a[:]:
    if x < 0: a.remove(x)
```

7.4 try语句The `try` statement

The `try` statement specifies exception handlers and/or cleanup code for a group of statements:

try语句为一组语句指定异常处理器和/或清除代码:

```
try_stmt      ::=  try_exc_stmt | try_fin_stmt
try_exc_stmt  ::=  "try" ":" suite
                   ("except" [expression ["," target]] ":" suite)+
                   ["else" ":" suite]
try_fin_stmt  ::=  "try" ":" suite "finally" ":" suite
```

There are two forms of `try` statement: `try`...`except` and `try`...`finally`. These forms cannot be mixed (but they can be nested in each other).

有两种形式的try**语句**: try...except **和**try...finally. **它们不能混合使用**(**但它们可以互相嵌套**).

The `try`...`except` form specifies one or more exception handlers (the `except` clauses). When no exception occurs in the `try` clause, no exception handler is executed. When an exception occurs in the `try` suite, a search for an exception handler is started. This search inspects the except clauses in turn until one is found that matches the exception. An expression-less except clause, if present, must be last; it matches any exception. For an except clause with an expression, that expression is evaluated, and the clause matches the exception if the resulting object is "compatible" with the exception. An object is compatible with an exception if it is either the object that identifies the exception, or (for exceptions that are classes) it is a base class of the exception, or it is a tuple containing an item that is compatible with the exception. Note that the object identities must match, i.e. it must be the same object, not just an object with the same value.

try...except**形式指定一个或多个异常处理器**(**异常子句**). **当在**try**子句中没有异常发生时**, **异常处理器将不被执行**. **当在**try**子句中有异常发生时**, **就会开始搜索异常处理器**.**它会按顺序搜索直到第一个匹配的处理器找到为止**. **如果存在一个没有指定异常的**except**语句**, **它必须放在最后**, **它会匹配任何异常**.**当一个**except**匹配**, **相应表达式会被计算**. **如果结果对象与该异常**"**兼容**", **那么该子句就匹配了这个异常**.**如果这个对象是标识这个异常的对象**, **或**(**异常类**)**是该异常的基类**, **或者它是一个包括与该异常兼容的对象的元组就称为这个对象是兼容的**. **注意对象的标识必须匹配**, **那就是说**, **它必须是相同的对象**, **不仅是具有相同值的对象**.

If no except clause matches the exception, the search for an exception handler continues in the surrounding code and on the invocation stack.

如果没有except**子句匹配异常**, **异常处理器的搜索工作将继续在调用栈的外层代码中进行**.

If the evaluation of an expression in the header of an except clause raises an exception, the original search for a handler is canceled and a search starts for the new exception in the surrounding code and on the call stack (it is treated as if the entire `try` statement raised the exception).

如果在except**子句头部计算表达式时就引发了异常**, **原来的异常处理器搜索工作就中断**, **并在外层代码搜索新的异常处理器**(**就好像处理整个**try**语句发生了异常一样**).

When a matching except clause is found, the exception's parameter is assigned to the target specified in that except clause, if present, and the except clause's suite is executed. All except clauses must have an executable block. When the end of this block is reached, execution continues normally after the entire try statement. (This means that if two nested handlers exist for the same exception, and the exception occurs in the try clause of the inner handler, the outer handler will not handle the exception.)

当找到了一个匹配的except**子句时**, **异常的参数被赋给了**except**子句中指定的对象**(**如果给出**), **并且执行其后的语句序列**. **所有的**except**子句必须一个可执行代码块**.**当执行该代码块末尾时**, **会转到整个**try**语句之后继续正常执行**(**这意味着**, **对于一个异常如果有两个嵌套的异常处理器**, **并且异常由内层的处理器处理**, **那么外层处理器就不会响应这个异常**).

Before an except clause's suite is executed, details about the exception are assigned to three variables in the `sys` module: `sys.exc_type` receives the object identifying the exception; `sys.exc_value` receives the exception's parameter; `sys.exc_traceback` receives a traceback object (see section 3.2) identifying the point in the program where the exception occurred. These details are also available through the `sys.exc_info()` function, which returns a tuple `(`*exc_type*`, `*exc_value*`, `*exc_traceback*`)`. Use of the corresponding variables is deprecated in favor of this function, since their use is unsafe in a threaded program. As of Python 1.5, the variables are restored to their previous values (before the call) when returning from a function that handled an exception.

在某个except**子句的语句序列被执行前**, **会将这个异常的详细情况记录在**sys**模块的三个变量中**:sys.exc_type**记录标识异常的对象**;sys.exc_vaule**记录异常的参数**; sys.exc_traceback**记录标识程序中异常发生点的跟踪回溯对象**(**见**3.2**节**).**这些信息可以通过函数**sys.exc_info()**得到**, **它会返回一个元组**(exc_type, exc_value, exc_traceback). **两者比较而言应该使用函数方法**, **因为在线程式化程序使用变量是不安全的**. **自**Python1.5**开始**, **这些变量在从返回发生异常的函数时会恢复它们之前的值**.

The optional `else` clause is executed if and when control flows off the end of the `try` clause.[1] Exceptions in the `else` clause are not handled by the preceding `except` clauses.

当控制从try**子句尾部中结束时**, **就执行可选的**else**子句**. [2] **在**else**子句中引发的异常不会在前面的**except**子句得到处理**.

The `try`...`finally` form specifies a 'cleanup' handler. The `try` clause is executed. When no exception occurs, the `finally` clause is executed. When an exception occurs in the `try` clause, the exception is temporarily saved, the `finally` clause is executed, and then the saved exception is re-raised. If the `finally` clause raises another exception or executes a `return` or `break` statement, the saved exception is lost. A `continue` statement is illegal in the `finally` clause. (The reason is a problem with the current implementation – this restriction may be lifted in the future). The exception information is not available to the program during execution of the `finally` clause.

try...finally**形式指定一个清除处理器**. **在执行**try**语句块没有异常发生时**, finally**子句被执行**.**在异常引发时**, **该异常就被临时保存起来**, finally**也被执行**, **然后暂存的异常被重新引发**.**如果执行**finally**子句时引发了另一个异常或执行了**return**或**break**语句**, **就会抛弃保存的异常**,**在**finally**子句中的**continue**语句是非法的**(**这么做的原因是当前实现的原因**—- **这个限制可能也会保留下去**)**在执行**finally**子句时异常信息是无效的**.

When a `return`, `break` or `continue` statement is executed in the `try` suite of a `try`...`finally` statement, the `finally` clause is also executed 'on the way out.' A `continue` statement is illegal in the `finally` clause. (The reason is a problem with the current implementation — this restriction may be lifted in the future).

当在try...finally**语句中的**try**语句序列的**return, break **或**continue**执行后**,finally**子句也会执行**.**在**finally**子句中的**continue**语句是非法的**(**这么做的原因是当前实现的原因**—- **这个限制可能也会保留下去**)

Additional information on exceptions can be found in section 4.2, and information on using the `raise` statement to generate exceptions may be found in section 6.9.

7.5 函数定义Function definitions

A function definition defines a user-defined function object (see section 3.2):

一个函数定义定义了一个用户自定义函数对象(**见**3.2**节**).

```
funcdef          ::=  "def" funcname "(" [parameter_list] ")" ":" suite
parameter_list   ::=  (defparameter ",")*
                      ("*" identifier [, "**" identifier]
                      | "**" identifier | defparameter [","])
defparameter     ::=  parameter ["=" expression]
sublist          ::=  parameter ("," parameter)* [","]
parameter        ::=  identifier | "(" sublist ")"
funcname         ::=  identifier
```

A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global namespace as the global namespace to be used when the function is called.

函数定义是一个可执行语句.**它在当前局部名字空间中将函数名字与函数对象**(**函数的可执行代码的包装**)**捆绑在一起**.**这个函数对象包括着一个全局名字空间的引用**, **以便在调用时使用**.

The function definition does not execute the function body; this gets executed only when the function is called.

函数定义不执行函数体, **它们仅仅在调用时执行**.

When one or more top-level parameters have the form *parameter* = *expression*, the function is said to have "default parameter values." For a parameter with a default value, the corresponding argument may be omitted from a call, in which case the parameter's default value is substituted. If a parameter has a default value, all following parameters must also have a default value — this is a syntactic restriction that is not expressed by the grammar.

当一个或多个参数以形式出现时, **这个函数就称为具有**"**默认参数值**". **对于有默认参数值的参数**, **在调用时它们可以省略**, **此时他们被赋予默认值**.**如果某参数具有默认值**, **则所有以后的参数都必须有默认值**—-

[1]Currently, control "flows off the end" except in the case of an exception or the execution of a `return`, `continue`, or `break` statement.
[2]**现在**, "**控制从**try**子句尾部中结束**"**除了出现异常情况和执行中有**return, continue, **或**break**语句的时候**.

这是一个在上述语法说明中没有提到的限制.

Default parameter values are evaluated when the function definition is executed. This means that the expression is evaluated once, when the function is defined, and that that same "pre-computed" value is used for each call. This is especially important to understand when a default parameter is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified. This is generally not what was intended. A way around this is to use `None` as the default, and explicitly test for it in the body of the function, e.g.:

默认参数值在函数定义被执行时计算.**这意味着这个表达式仅仅求值一次**, **时间是函数定义时**, **在并且在每次调用时都使用相同的**"**预计算**"**的值**.**这在理解默认参数值是一个像列表**,**字典这样的可变对象时特别值得注意**.**如果修改了这个对象**(**例如给列表追加了一项**).**默认值也随之修改**.**这通常不是想要发生的**.**一个避免这个麻烦的方法就是使用**None**作默认值**, **并且在函数体是作显式的测试**, **例如**:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section 5.3.4. A function call always assigns values to all parameters mentioned in the parameter list, either from position arguments, from keyword arguments, or from default values. If the form "`*identifier`" is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form "`**identifier`" is present, it is initialized to a new dictionary receiving any excess keyword arguments, defaulting to a new empty dictionary.

函数调用语义的详细说明,**参见**5.3.4**节**. **通常一个函数调用会给所有出现在参数表中的参数赋一个值**,**要么是位置参数**, **或者关键字参数**, **或是默认值**.**如果使用的**"*identifier"**形式**,**它就被初始化一个接受所有额外位置参数的元组**, **默认为空元组**.**如果使用的**"**identifier"**形式**,**它就被初始化一个接受所有额外关键字参数的字典**, **默认为空元组**.

It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions. This uses lambda forms, described in section 5.11. Note that the lambda form is merely a shorthand for a simplified function definition; a function defined in a "`def`" statement can be passed around or assigned to another name just like a function defined by a lambda form. The "`def`" form is actually more powerful since it allows the execution of multiple statements.

也可以创建匿名函数(**没和名字捆绑的函数**), **能直接在表达式中使用**.**这是通过**lambda**表达式实现的**,**详见**5.10**节**.**注意**lambda**仅仅是一个简单函数的简写形式**; **以**"def"**定义的函数可以传递**, **可以赋予另一个名字**, **就和以**lambda**定义的函数一样**.**以**"def"**定义的函数功能要更强大些**, **因为它允许执行多条语句**.

Programmer's note: Functions are first-class objects. A "`def`" form executed inside a function definition defines a local function that can be returned or passed around. Free variables used in the nested function can access the local variables of the function containing the def. See section 4.1 for details.

程序员注意: **在函数定义中执行的**"def", **可以返回和传递**.**在嵌套函数中名字的语义会在**Python 2.2**中有所改变**,**可以参考附录中对新语义的描述**.

7.6 类定义Class definitions

A class definition defines a class object (see section 3.2):

一个类定义定义一个类对象(**见**3.2**节**)

```
classdef     ::=  "class" classname [inheritance] ":" suite
inheritance  ::=  "(" [expression_list] ")"
classname    ::=  identifier
```

A class definition is an executable statement. It first evaluates the inheritance list, if present. Each item in the inheritance list should evaluate to a class object or class type which allows subclassing. The class's suite is then executed in a new execution frame (see section 4.1), using a newly created local namespace and the original global namespace. (Usually, the suite contains only function definitions.) When the class's suite finishes execution, its execution frame is discarded but its local namespace is saved. A class object is then created using the inheritance list for the base classes and the saved local namespace for the attribute dictionary. The class name is bound to this class object in the original local namespace.

一个类定义是一个可以执行语句. 它首先考察继承关系表,如果存在. 在继承关系表中的每个对象都生成一个类对象.然后类的语句序列在新的堆栈结构(见4.1节)使用新创建的局部名字空间和原来的全局名字空间执行.(通常这个语句序列仅仅包括函数定义)当该类的语句序执行结束后就丢弃掉这个堆栈结构但该局部名字空间会被保存下来.然后便使用其继承关系表创建基类和保存下来的名字空间作为属性字典创建新的类对象.在最初的名字空间中, 类的名字绑定在这个类对象上.

Programmer's note: Variables defined in the class definition are class variables; they are shared by all instances. To define instance variables, they must be given a value in the `__init__()` method or in another method. Both class and instance variables are accessible through the notation "`self.name`", and an instance variable hides a class variable with the same name when accessed in this way. Class variables with immutable values can be used as defaults for instance variables. For new-style classes, descriptors can be used to create instance variables with different implementation details.

程序员注意:在类定义中定义的变量是类变量; 它们共享所有的类实例, 为了定义变量, 它们必须在__init__()**或其它方法中被指定一个值.所有类变量和实例变量都可以通过**"self.name"**记号访问, 并且实例变量如果和类变量重名, 此时会隐藏类变量.具有不可变值的类变量可以作为实例变量的默认值.**

第八章

顶层构件Top-level components

The Python interpreter can get its input from a number of sources: from a script passed to it as standard input or as program argument, typed in interactively, from a module source file, etc. This chapter gives the syntax used in these cases.

Python解释器可以从几种来源得到输入：从作为标准输入或作为程序参数传入的脚本，交互方式下的输入，从模块源文件，等等。本章给出在这些情况下使用的语法。

8.1 完整的Python程序Complete Python programs

While a language specification need not prescribe how the language interpreter is invoked, it is useful to have a notion of a complete Python program. A complete Python program is executed in a minimally initialized environment: all built-in and standard modules are available, but none have been initialized, except for `sys` (various system services), `__builtin__` (built-in functions, exceptions and `None`) and `__main__`. The latter is used to provide the local and global namespace for execution of the complete program.

尽管语法规格说明不需要指明语言的解释器是如何执行的，对一个完整的Python程序的了解也是有用的。一个完整的Python程序运行在一个最低限度的初始环境中：所有内置和标准模块都是可用的，但都没有被初始化，除了sys(各种系统服务)模块、__builtin__(内置函数、异常和None)模块和__main__模块。__main__被用来为完整程序的运行提供局部和全局名字空间。

The syntax for a complete Python program is that for file input, described in the next section.

针对于文件输入来说的完整的Python程序语法，在下一节给出描述。

The interpreter may also be invoked in interactive mode; in this case, it does not read and execute a complete program but reads and executes one statement (possibly compound) at a time. The initial environment is identical to that of a complete program; each statement is executed in the namespace of `__main__`.

解释器也可以以交互方式运行；在这种情况下，它并不读取和运行一个完整程序，而是一次读取和运行一条语句(可能是复合语句)。这种初始环境与完整程序环境是相同的；每条语句都是在__main__名字空间下运行。

Under UNIX, a complete program can be passed to the interpreter in three forms: with the **-c** *string* command line option, as a file passed as the first command line argument, or as standard input. If the file or standard input is a tty device, the interpreter enters interactive mode; otherwise, it executes the file as a complete program.

在Unix上，一个完整程序可以以三种形式传给解释器：使用-c字符串命令行选项，以一个文件作为命令行的第一个参数，或作为标准输入。如果文件或标准输入是一个tty(终端)设备，解释器进行交互模式；否则，它把文件作为一个完整程序来运行。

8.2 文件输入File input

All input read from non-interactive files has the same form:

所有从非交互文件的输入读取具有相同的形式：

```
file_input  ::=  (NEWLINE | statement)*
```

This syntax is used in the following situations:

这个语法用于以下的情况：

- when parsing a complete Python program (from a file or from a string); **当解析一个完整**Python**程序时**(**从文件或字符串中**)**；**
- when parsing a module; **当解析一个模块时；**
- when parsing a string passed to the `exec` statement; **当解析一个传给**exec**语句的字符串时；**

8.3 交互式输入Interactive input

Input in interactive mode is parsed using the following grammar:

交互模式输入使用以下语法进行解析：

```
interactive_input  ::=  [stmt_list] NEWLINE | compound_stmt NEWLINE
```

Note that a (top-level) compound statement must be followed by a blank line in interactive mode; this is needed to help the parser detect the end of the input.

请注意一个(**顶层**)**复合语句后面在交互模式下必须跟着一个空行；需要用它来帮助解释器检测输入的结束。**

8.4 表达式输入Expression input

There are two forms of expression input. Both ignore leading whitespace. The string argument to `eval()` must have the following form:

有两种表达式输入形式。两种都忽略掉前导空白。对eval()**的字符串参数必须有以下形式：**

```
eval_input  ::=  expression_list NEWLINE*
```

The input line read by `input()` must have the following form:

通过input()**读入的输入行必须有以下形式：**

```
input_input  ::=  expression_list NEWLINE
```

Note: to read 'raw' input line without interpretation, you can use the built-in function `raw_input()` or the `readline()` method of file objects.

注意：想要读出未经处理的'**原始**'**输入行，你可以使用内置函数**raw_input()**或**file**对象的**readline()**方法。**

附录 A

History and License

A.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.2	2.3.1	2003	PSF	yes

Note: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

A.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.3.3

1. This LICENSE AGREEMENT is between the Python Software Foundation (“PSF”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 2.3.3 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.3.3 alone or in any derivative version, provided, however, that PSF’s License Agreement and PSF’s notice of copyright, i.e., “Copyright © 2001-2003 Python Software Foundation; All Rights Reserved” are retained in Python 2.3.3 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.3.3 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.3.3.

4. PSF is making Python 2.3.3 available to Licensee on an “AS IS” basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.3.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.3.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.3.3, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.3.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0
BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com (“BeOpen”), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization (“Licensee”) accessing and otherwise using this software in source or binary form and its associated documentation (“the Software”).

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an “AS IS” basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the “BeOpen Python” logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (“CNRI”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI’s License Agreement and CNRI’s notice of copyright, i.e., “Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved” are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI’s License Agreement, Licensee may substitute the following text (omitting the quotes): “Python 1.6.1 is made available subject to the terms and conditions in CNRI’s License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013.”

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an “AS IS” basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia’s conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the “ACCEPT” button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

附录 B

修正记录

日期	修正位置	建议人	备注
2002.03.10	5.2.5 字典的表示	smille	
2002.03.09	5.2.1 标识符(名字)	limodou,smille	
2002.03.07	2.1.8 符号间的空白	limodou	
2002.03.07	2.1.7 缩进	limodou	
2002.03.07	2.词法分析	limodou	
2002.03.07	2.1.6 空行	limodou	
2002.03.06	2.1.3 注释	limodou	
2002.03.06	2.1.4 显式行连接	limodou	
2002.03.06	2.1.5 隐式行连接	limodou	
2002.03.06	5.2.1 标识符(名字)	limodou	
2002.03.05	2.词法分析	limodou	
2002.03.02	4.2 异常	smille	
2002.02.23	2.3.2 保留的名字	smille	
2002.02.23	4.1 代码块, 运行结构框架和命名空间	smille	
2002.02.23	2.1.3 符号间的空白	smille	
2002.02.23	2.2 其它符号	smille	
2002.02.23	3.2 标准类型层次	smille	
2002.02.14	1.1 记法	limodou, smille	token待改
2002.02.13	前言	limodou	
2002.02.10	3.3.6 模拟数值类型	smille	

注意：

*** 本修正记录不包括对笔误的修改，因为这可能使用本文件快速膨胀，也没有实际意义。有关具体的修改位置和变动内容可以参见中国Linux论坛中Python编程区中关于本文档的专栏。**

*** 本修正记录只记录署名作者之外的贡献者对本文档的修改。**

附录 C

翻译团队

Python **参考手册译者列表**(**字母序**):

- * Limodou:
  o 8. **顶层构件**(Top-level components)
- * Raise li Sail:
  o 1. **介绍**(Introduction)
  o 2. **词法分析（**Lexical analysis**）**
  o 3. **数据模型**(Data model)
  o 4. **运行模型**(Execution model)
  o 5. **表达式**(Expressions)
  o 6.1 **表达式语句**(Expression statements)
  o 6.2 **断言语句**(Assert statements)
  o 6.4 pass **语句**
  o 6.5 del **语句**
  o 6.7 return **语句**
  o 6.10 break **语句**
  o 6.11 continue **语句**
  o 6.13 global **语句**
  o 7. **复合语句**(Compound statements)
- * Smille:
  o 6.3 **赋值语句**(Assignment statements)
  o 6.3.1 **增量赋值语句**(Augmented Assignment statements)
  o 6.6 print**语句**(The print statement)
  o 6.8 yield**语句**(The yield statement)
  o 6.9 raise**语句**(The raise statement)
  o 6.12 import **语句**(The import statement)
  o 6.14 exec **语句**(The exec statement)