The Python Library Reference

发布 *2.7.18*

Guido van Rossum
and the Python development team

五月 20, 2020

Python Software Foundation
Email: docs@python.org

Contents

reference-index 描述了 Python 语言的具体语法和语义，这份库参考则介绍了与 Python 一同发行的标准库。它还描述了通常包含在 Python 发行版中的一些可选组件。

Python 标准库非常庞大，所提供的组件涉及范围十分广泛，正如以下内容目录所显示的。这个库包含了多个内置模块 (以 C 编写)，Python 程序员必须依靠它们来实现系统级功能，例如文件 I/O，此外还有大量以 Python 编写的模块，提供了日常编程中许多问题的标准解决方案。其中有些模块经过专门设计，通过将特定平台功能抽象化为平台中立的 API 来鼓励和加强 Python 程序的可移植性。

Windows 版本的 Python 安装程序通常包含整个标准库，往往还包含许多额外组件。对于类 Unix 操作系统，Python 通常会分成一系列的软件包，因此可能需要使用操作系统所提供的包管理工具来获取部分或全部可选组件。

在这个标准库以外还存在成千上万并且不断增加的其他组件 (从单独的程序、模块、软件包直到完整的应用开发框架)，访问 Python 包索引 即可获取这些第三方包。

CHAPTER 1

概述

“Python 库”中包含了几种不同的组件。

它包含通常被视为语言“核心”中的一部分的数据类型，例如数字和列表。对于这些类型，Python 语言核心定义了文字的形式，并对它们的语义设置了一些约束，但没有完全定义语义。（另一方面，语言核心确实定义了语法属性，如操作符的拼写和优先级。）

这个库也包含了内置函数和异常—不需要 `import` 语句就可以在所有 Python 代码中使用的对象。有一些是由语言核心定义的，但是许多对于核心语义不是必需的，并且仅在这里描述。

不过这个库主要是由一系列的模块组成。这些模块集可以不同方式分类。有些模块是用 C 编写并内置于 Python 解释器中；另一些模块则是用 Python 编写并以源码形式导入。有些模块提供专用于 Python 的接口，例如打印栈追踪信息；有些模块提供专用于特定操作系统的接口，例如操作特定的硬件；另一些模块则提供针对特定应用领域的接口，例如万维网。有些模块在所有更新和移植版本的 Python 中可用；另一些模块仅在底层系统支持或要求时可用；还有些模块则仅当编译和安装 Python 时选择了特定配置选项时才可用。

This manual is organized “from the inside out:” it first describes the built-in data types, then the built-in functions and exceptions, and finally the modules, grouped in chapters of related modules. The ordering of the chapters as well as the ordering of the modules within each chapter is roughly from most relevant to least important.

这意味着如果你从头开始阅读本手册，并在感到厌烦时跳到下一章，你仍能对 Python 库的可用模块和所支持的应用领域有个大致了解。当然，你并非必须如同读小说一样从头读到尾—你也可以先浏览内容目录 (在手册开头)，或在索引 (在手册末尾) 中查找某个特定函数、模块或条目。最后，如果你喜欢随意学习某个主题，你可以选择一个随机页码 (参见 *random* 模块) 并读上一两小节。无论你想以怎样的顺序阅读本手册，还是建议先从内置函数 这一章开始，因为本手册的其余内容都需要你熟悉其中的基本概念。

让我们开始吧！

CHAPTER 2

内置函数

The Python interpreter has a number of functions built into it that are always available. They are listed here in alphabetical order.

		内置函数		
abs()	*divmod()*	*input()*	*open()*	*staticmethod()*
all()	*enumerate()*	*int()*	*ord()*	*str()*
any()	*eval()*	*isinstance()*	*pow()*	*sum()*
basestring()	*execfile()*	*issubclass()*	*print()*	*super()*
bin()	*file()*	*iter()*	*property()*	*tuple()*
bool()	*filter()*	*len()*	*range()*	*type()*
bytearray()	*float()*	*list()*	*raw_input()*	*unichr()*
callable()	*format()*	*locals()*	*reduce()*	*unicode()*
chr()	*frozenset()*	*long()*	*reload()*	*vars()*
classmethod()	*getattr()*	*map()*	*repr()*	*xrange()*
cmp()	*globals()*	*max()*	*reversed()*	*zip()*
compile()	*hasattr()*	*memoryview()*	*round()*	*__import__()*
complex()	*hash()*	*min()*	*set()*	
delattr()	*help()*	*next()*	*setattr()*	
dict()	*hex()*	*object()*	*slice()*	
dir()	*id()*	*oct()*	*sorted()*	

In addition, there are other four built-in functions that are no longer considered essential: *apply()*, *buffer()*, *coerce()*, and *intern()*. They are documented in the *Non-essential Built-in Functions* section.

abs(*x*)

Return the absolute value of a number. The argument may be a plain or long integer or a floating point number. If the argument is a complex number, its magnitude is returned.

all(*iterable*)

如果 *iterable* 的所有元素为真（或迭代器为空），返回 `True` 。等价于:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

2.5 新版功能.

any(*iterable*)

如果 *iterable* 的任一元素为真则返回 `True`。如果迭代器为空，返回 `False`。等价于:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

2.5 新版功能.

basestring()

This abstract type is the superclass for *str* and *unicode*. It cannot be called or instantiated, but it can be used to test whether an object is an instance of *str* or *unicode*. `isinstance(obj, basestring)` is equivalent to `isinstance(obj, (str, unicode))`.

2.3 新版功能.

bin(*x*)

Convert an integer number to a binary string. The result is a valid Python expression. If *x* is not a Python *int* object, it has to define an `__index__()` method that returns an integer.

2.6 新版功能.

class bool([*x*])

Return a Boolean value, i.e. one of `True` or `False`. *x* is converted using the standard truth testing procedure. If *x* is false or omitted, this returns *False*; otherwise it returns *True*. *bool* is also a class, which is a subclass of *int*. Class *bool* cannot be subclassed further. Its only instances are *False* and *True*.

2.2.1 新版功能.

在 2.3 版更改: If no argument is given, this function returns *False*.

class bytearray([*source*[, *encoding*[, *errors*]]])

Return a new array of bytes. The *bytearray* class is a mutable sequence of integers in the range 0 <= x < 256. It has most of the usual methods of mutable sequences, described in 可变序列类型, as well as most methods that the *str* type has, see 字符串的方法.

可选形参 *source* 可以用不同的方式来初始化数组：

- If it is *unicode*, you must also give the *encoding* (and optionally, *errors*) parameters; *bytearray()* then converts the unicode to bytes using `unicode.encode()`.
- 如果是一个 *integer*，会初始化大小为该数字的数组，并使用 null 字节填充。
- 如果是一个符合 *buffer* 接口的对象，该对象的只读 buffer 会用来初始化字节数组。
- 如果是一个 *iterable* 可迭代对象，它的元素的范围必须是 `0 <= x < 256` 的整数，它会被用作数组的初始内容。

如果没有实参，则创建大小为 0 的数组。

2.6 新版功能.

callable(*object*)
Return *True* if the *object* argument appears callable, *False* if not. If this returns true, it is still possible that a call fails, but if it is false, calling *object* will never succeed. Note that classes are callable (calling a class returns a new instance); class instances are callable if they have a `__call__()` method.

chr(*i*)
Return a string of one character whose ASCII code is the integer *i*. For example, `chr(97)` returns the string `'a'`. This is the inverse of *ord()*. The argument must be in the range [0..255], inclusive; *ValueError* will be raised if *i* is outside that range. See also *unichr()*.

classmethod(*function*)
Return a class method for *function*.

一个类方法把类自己作为第一个实参，就像一个实例方法把实例自己作为第一个实参。请用以下习惯来声明类方法:

```
class C(object):
    @classmethod
    def f(cls, arg1, arg2, ...):
        ...
```

`@classmethod` 这样的形式称为函数的*decorator* –详情参阅 function。

类方法的调用可以在类上进行 (例如 `C.f()`) 也可以在实例上进行 (例如 `C().f()`)。其所属类以外的类实例会被忽略。如果类方法在其所属类的派生类上调用，则该派生类对象会被作为隐含的第一个参数被传入。

类方法与 C++ 或 Java 中的静态方法不同。如果你需要后者，请参阅*staticmethod()*。

想了解更多有关类方法的信息，请参阅 types 。

2.2 新版功能.

在 2.4 版更改: Function decorator syntax added.

cmp(*x*, *y*)
Compare the two objects *x* and *y* and return an integer according to the outcome. The return value is negative if `x < y`, zero if `x == y` and strictly positive if `x > y`.

compile(*source*, *filename*, *mode*[, *flags*[, *dont_inherit*]])
Compile the *source* into a code or AST object. Code objects can be executed by an `exec` statement or evaluated by a call to *eval()*. *source* can either be a Unicode string, a *Latin-1* encoded string or an AST object. Refer to the *ast* module documentation for information on how to work with AST objects.

filename 实参需要是代码读取的文件名；如果代码不需要从文件中读取，可以传入一些可辨识的值（经常会使用 `'<string>'`）。

mode 实参指定了编译代码必须用的模式。如果 *source* 是语句序列，可以是 `'exec'`；如果是单一表达式，可以是 `'eval'`；如果是单个交互式语句，可以是 `'single'`。(在最后一种情况下，如果表达式执行结果不是 `None` 将会被打印出来。)

The optional arguments *flags* and *dont_inherit* control which future statements (see **PEP 236**) affect the compilation of *source*. If neither is present (or both are zero) the code is compiled with those future statements that are in effect in the code that is calling *compile()*. If the *flags* argument is given and *dont_inherit* is not (or is zero) then the future statements specified by the *flags* argument are used in addition to those that would be used anyway. If *dont_inherit* is a non-zero integer then the *flags* argument is it –the future statements in effect around the call to compile are ignored.

Future 语句使用比特位来指定，多个语句可以通过按位或来指定。具体特性的比特位可以通过*__future__* 模块中的 `_Feature` 类的实例的 `compiler_flag` 属性来获得。

This function raises *SyntaxError* if the compiled source is invalid, and *TypeError* if the source contains null bytes.

如果您想分析 Python 代码的 AST 表示，请参阅*ast.parse()*。

注解: 在 `'single'` 或 `'eval'` 模式编译多行代码字符串时，输入必须以至少一个换行符结尾。这使*code* 模块更容易检测语句的完整性。

警告: 在将足够大或者足够复杂的字符串编译成 AST 对象时，Python 解释器有可以因为 Python AST 编译器的栈深度限制而崩溃。

在 2.3 版更改: The *flags* and *dont_inherit* arguments were added.

在 2.6 版更改: Support for compiling AST objects.

在 2.7 版更改: Allowed use of Windows and Mac newlines. Also input in `'exec'` mode does not have to end in a newline anymore.

class complex([*real*[, *imag*]])

Return a complex number with the value *real* + *imag**1j or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If *imag* is omitted, it defaults to zero and the function serves as a numeric conversion function like *int()*, *long()* and *float()*. If both arguments are omitted, returns `0j`.

注解: 当从字符串转换时，字符串在 + 或 - 的周围必须不能有空格。例如 `complex('1+2j')` 是合法的，但 `complex('1 + 2j')` 会触发*ValueError* 异常。

Numeric Types —int, float, long, complex 描述了复数类型。

delattr(*object*, *name*)

setattr() 相关的函数。实参是一个对象和一个字符串。该字符串必须是对象的某个属性。如果对象允许，该函数将删除指定的属性。例如 `delattr(x, 'foobar')` 等价于 `del x.foobar` 。

class dict(***kwarg*)
class dict(*mapping*, ***kwarg*)
class dict(*iterable*, ***kwarg*)

创建一个新的字典。*dict* 对象是一个字典类。参见*dict* 和*映射类型—dict* 了解这个类。

其他容器类型，请参见内置的 `list`、*set* 和*tuple* 类，以及*collections* 模块。

dir([*object*])

如果没有实参，则返回当前本地作用域中的名称列表。如果有实参，它会尝试返回该对象的有效属性列表。

如果对象有一个名为 `__dir__()` 的方法，那么该方法将被调用，并且必须返回一个属性列表。这允许实现自定义 `__getattr__()` 或 `__getattribute__()` 函数的对象能够自定义*dir()* 来报告它们的属性。

如果对象不提供 `__dir__()`，这个函数会尝试从对象已定义的*__dict__* 属性和类型对象收集信息。结果列表并不总是完整的，如果对象有自定义 `__getattr__()`，那结果可能不准确。

默认的*dir()* 机制对不同类型的对象行为不同，它会试图返回最相关而不是最全的信息:

- 如果对象是模块对象，则列表包含模块的属性名称。

- 如果对象是类型或类对象，则列表包含它们的属性名称，并且递归查找所有基类的属性。
- 否则，列表包含对象的属性名称，它的类属性名称，并且递归查找它的类的所有基类的属性。

返回的列表按字母表排序。例如：

```
>>> import struct
>>> dir()   # show the names in the module namespace
['__builtins__', '__doc__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape(object):
        def __dir__(self):
            return ['area', 'perimeter', 'location']
>>> s = Shape()
>>> dir(s)
['area', 'perimeter', 'location']
```

注解： 因为*dir()* 主要是为了便于在交互式时使用，所以它会试图返回人们感兴趣的名字集合，而不是试图保证结果的严格性或一致性，它具体的行为也可能在不同版本之间改变。例如，当实参是一个类时，metaclass 的属性不包含在结果列表中。

divmod(*a*, *b*)

Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using long division. With mixed operand types, the rules for binary arithmetic operators apply. For plain and long integers, the result is the same as `(a // b, a % b)`. For floating point numbers the result is `(q, a % b)`, where *q* is usually `math.floor(a / b)` but may be 1 less than that. In any case `q * b + a % b` is very close to *a*, if `a % b` is non-zero it has the same sign as *b*, and `0 <= abs(a % b) < abs(b)`.

在 2.3 版更改: Using *divmod()* with complex numbers is deprecated.

enumerate(*sequence*, *start=0*)

Return an enumerate object. *sequence* must be a sequence, an *iterator*, or some other object which supports iteration. The `next()` method of the iterator returned by *enumerate()* returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *sequence*:

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

等价于:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

2.3 新版功能.

在 2.6 版更改: The *start* parameter was added.

eval(*expression*[, *globals*[, *locals*]])

The arguments are a Unicode or *Latin-1* encoded string and optional globals and locals. If provided, *globals* must

be a dictionary. If provided, *locals* can be any mapping object.

在 2.4 版更改: formerly *locals* was required to be a dictionary.

The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* dictionaries as global and local namespace. If the *globals* dictionary is present and lacks '__builtins__', the current globals are copied into *globals* before *expression* is parsed. This means that *expression* normally has full access to the standard `__builtin__` module and restricted environments are propagated. If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed in the environment where `eval()` is called. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```
>>> x = 1
>>> print eval('x+1')
2
```

这个函数也可以用来执行任何代码对象（如`compile()` 创建的）。这种情况下，参数是代码对象，而不是字符串。如果编译该对象时的 *mode* 实参是 `'exec'` 那么`eval()` 返回值为 `None` 。

Hints: dynamic execution of statements is supported by the `exec` statement. Execution of statements from a file is supported by the `execfile()` function. The `globals()` and `locals()` functions returns the current global and local dictionary, respectively, which may be useful to pass around for use by `eval()` or `execfile()`.

另外可以参阅`ast.literal_eval()`，该函数可以安全执行仅包含文字的表达式字符串。

execfile(*filename*[, *globals*[, *locals*]])

This function is similar to the `exec` statement, but parses a file instead of a string. It is different from the `import` statement in that it does not use the module administration —it reads the file unconditionally and does not create a new module.[1]

The arguments are a file name and two optional dictionaries. The file is parsed and evaluated as a sequence of Python statements (similarly to a module) using the *globals* and *locals* dictionaries as global and local namespace. If provided, *locals* can be any mapping object. Remember that at module level, globals and locals are the same dictionary. If two separate objects are passed as *globals* and *locals*, the code will be executed as if it were embedded in a class definition.

在 2.4 版更改: formerly *locals* was required to be a dictionary.

If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed in the environment where `execfile()` is called. The return value is `None`.

注解: The default *locals* act as described for function `locals()` below: modifications to the default *locals* dictionary should not be attempted. Pass an explicit *locals* dictionary if you need to see effects of the code on *locals* after function `execfile()` returns. `execfile()` cannot be used reliably to modify a function's locals.

---

file(*name*[, *mode*[, *buffering*]])

Constructor function for the `file` type, described further in section *File Objects*. The constructor's arguments are the same as those of the `open()` built-in function described below.

When opening a file, it's preferable to use `open()` instead of invoking this constructor directly. `file` is more suited to type testing (for example, writing `isinstance(f, file)`).

2.2 新版功能.

filter(*function*, *iterable*)

Construct a list from those elements of *iterable* for which *function* returns true. *iterable* may be either a sequence, a container which supports iteration, or an iterator. If *iterable* is a string or a tuple, the result also has that type;

[1] It is used relatively rarely so does not warrant being made into a statement.

otherwise it is always a list. If *function* is `None`, the identity function is assumed, that is, all elements of *iterable* that are false are removed.

Note that `filter(function, iterable)` is equivalent to `[item for item in iterable if function(item)]` if function is not `None` and `[item for item in iterable if item]` if function is `None`.

See *itertools.ifilter()* and *itertools.ifilterfalse()* for iterator versions of this function, including a variation that filters for elements where the *function* returns false.

class float([*x*])
返回从数字或字符串 *x* 生成的浮点数。

If the argument is a string, it must contain a possibly signed decimal or floating point number, possibly embedded in whitespace. The argument may also be [+|-]nan or [+|-]inf. Otherwise, the argument may be a plain or long integer or a floating point number, and a floating point number with the same value (within Python's floating point precision) is returned. If no argument is given, returns `0.0`.

注解: When passing in a string, values for NaN and Infinity may be returned, depending on the underlying C library. Float accepts the strings nan, inf and -inf for NaN and positive or negative infinity. The case and a leading + are ignored as well as a leading - is ignored for NaN. Float always represents NaN and infinity as nan, inf or -inf.

Numeric Types —int, float, long, complex 描述了浮点类型。

format(*value*[, *format_spec*])
将 *value* 转换为 *format_spec* 控制的“格式化”表示。*format_spec* 的解释取决于 *value* 实参的类型，但是大多数内置类型使用标准格式化语法：格式规格迷你语言。

注解: `format(value, format_spec)` merely calls `value.__format__(format_spec)`.

2.6 新版功能.

class frozenset([*iterable*])
返回一个新的*frozenset* 对象，它包含可选参数 *iterable* 中的元素。`frozenset` 是一个内置的类。有关此类的文档，请参阅*frozenset* 和集合类型—*set, frozenset*。

请参阅内建的*set*、`list`、*tuple* 和*dict* 类，以及*collections* 模块来了解其它的容器。

2.4 新版功能.

getattr(*object*, *name*[, *default*])
返回对象命名属性的值。*name* 必须是字符串。如果该字符串是对象的属性之一，则返回该属性的值。例如，`getattr(x, 'foobar')` 等同于 `x.foobar`。如果指定的属性不存在，且提供了 *default* 值，则返回它，否则触发*AttributeError*。

globals()
返回表示当前全局符号表的字典。这总是当前模块的字典（在函数或方法中，不是调用它的模块，而是定义它的模块）。

hasattr(*object*, *name*)
The arguments are an object and a string. The result is `True` if the string is the name of one of the object's attributes, `False` if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an exception or not.)

hash(*object*)
Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

help([*object*])
启动内置的帮助系统（此函数主要在交互式中使用）。如果没有实参，解释器控制台里会启动交互式帮助系统。如果实参是一个字符串，则在模块、函数、类、方法、关键字或文档主题中搜索该字符串，并在控制台上打印帮助信息。如果实参是其他任意对象，则会生成该对象的帮助页。

该函数通过*site*模块加入到内置命名空间。

2.2 新版功能.

hex(*x*)
Convert an integer number (of any size) to a lowercase hexadecimal string prefixed with "0x", for example:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
>>> hex(1L)
'0x1L'
```

If x is not a Python *int* or *long* object, it has to define a __hex__() method that returns a string.

另请参阅*int()*将十六进制字符串转换为以 16 为基数的整数。

注解: 如果要获取浮点数的十六进制字符串形式，请使用*float.hex()*方法。

在 2.4 版更改: Formerly only returned an unsigned literal.

id(*object*)
Return the "identity" of an object. This is an integer (or long integer) which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same *id()* value.

CPython implementation detail: This is the address of the object in memory.

input([*prompt*])
Equivalent to `eval(raw_input(prompt))`.

This function does not catch user errors. If the input is not syntactically valid, a *SyntaxError* will be raised. Other exceptions may be raised if there is an error during evaluation.

如果加载了*readline*模块，*input()*将使用它来提供复杂的行编辑和历史记录功能。

Consider using the *raw_input()* function for general input from users.

class int(*x=0*)
class int(*x*, *base=10*)
Return an integer object constructed from a number or string *x*, or return `0` if no arguments are given. If *x* is a number, it can be a plain integer, a long integer, or a floating point number. If *x* is floating point, the conversion truncates towards zero. If the argument is outside the integer range, the function returns a long object instead.

If *x* is not a number or if *base* is given, then *x* must be a string or Unicode object representing an integer literal in radix *base*. Optionally, the literal can be preceded by + or − (with no space in between) and surrounded by whitespace. A base-n literal consists of the digits 0 to n-1, with `a` to `z` (or `A` to `Z`) having values 10 to 35. The default *base* is 10. The allowed values are 0 and 2–36. Base-2, -8, and -16 literals can be optionally prefixed with `0b`/`0B`, `0o`/`0O`/`0`, or `0x`/`0X`, as with integer literals in code. Base 0 means to interpret the string exactly as an integer literal, so that the actual base is 2, 8, 10, or 16.

整数类型定义请参阅*Numeric Types —int, float, long, complex* 。

isinstance(*object*, *classinfo*)
Return true if the *object* argument is an instance of the *classinfo* argument, or of a (direct, indirect or *virtual*)

subclass thereof. Also return true if *classinfo* is a type object (new-style class) and *object* is an object of that type or of a (direct, indirect or *virtual*) subclass thereof. If *object* is not a class instance or an object of the given type, the function always returns false. If *classinfo* is a tuple of class or type objects (or recursively, other such tuples), return true if *object* is an instance of any of the classes or types. If *classinfo* is not a class, type, or tuple of classes, types, and such tuples, a `TypeError` exception is raised.

在 2.2 版更改: Support for a tuple of type information was added.

issubclass(*class*, *classinfo*)
Return true if *class* is a subclass (direct, indirect or *virtual*) of *classinfo*. A class is considered a subclass of itself. *classinfo* may be a tuple of class objects, in which case every entry in *classinfo* will be checked. In any other case, a `TypeError` exception is raised.

在 2.3 版更改: Support for a tuple of type information was added.

iter(*o*[, *sentinel*])
Return an *iterator* object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, *o* must be a collection object which supports the iteration protocol (the `__iter__()` method), or it must support the sequence protocol (the `__getitem__()` method with integer arguments starting at `0`). If it does not support either of those protocols, `TypeError` is raised. If the second argument, *sentinel*, is given, then *o* must be a callable object. The iterator created in this case will call *o* with no arguments for each call to its `next()` method; if the value returned is equal to *sentinel*, `StopIteration` will be raised, otherwise the value will be returned.

One useful application of the second form of `iter()` is to read lines of a file until a certain line is reached. The following example reads a file until the `readline()` method returns an empty string:

```
with open('mydata.txt') as fp:
    for line in iter(fp.readline, ''):
        process_line(line)
```

2.2 新版功能.

len(*s*)
返回对象的长度（元素个数）。实参可以是序列（如 string、bytes、tuple、list 或 range 等）或集合（如 dictionary、set 或 frozen set 等）。

class list([*iterable*])
Return a list whose items are the same and in the same order as *iterable*' s items. *iterable* may be either a sequence, a container that supports iteration, or an iterator object. If *iterable* is already a list, a copy is made and returned, similar to `iterable[:]`. For instance, `list('abc')` returns `['a', 'b', 'c']` and `list( (1, 2, 3) )` returns `[1, 2, 3]`. If no argument is given, returns a new empty list, `[]`.

`list` is a mutable sequence type, as documented in *Sequence Types —str, unicode, list, tuple, bytearray, buffer, xrange*. For other containers see the built in `dict`, `set`, and `tuple` classes, and the `collections` module.

locals()
Update and return a dictionary representing the current local symbol table. Free variables are returned by `locals()` when it is called in function blocks, but not in class blocks.

注解: 不要更改此字典的内容；更改不会影响解释器使用的局部变量或自由变量的值。

---

class long(*x=0*)
class long(*x*, *base=10*)
Return a long integer object constructed from a string or number *x*. If the argument is a string, it must contain a possibly signed number of arbitrary size, possibly embedded in whitespace. The *base* argument is interpreted in the same way as for `int()`, and may only be given when *x* is a string. Otherwise, the argument may be a plain or

long integer or a floating point number, and a long integer with the same value is returned. Conversion of floating point numbers to integers truncates (towards zero). If no arguments are given, returns `0L`.

The long type is described in *Numeric Types —int, float, long, complex*.

map(*function*, *iterable*, ...)
Apply *function* to every item of *iterable* and return a list of the results. If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. If one iterable is shorter than another it is assumed to be extended with `None` items. If *function* is `None`, the identity function is assumed; if there are multiple arguments, *map()* returns a list consisting of tuples containing the corresponding items from all iterables (a kind of transpose operation). The *iterable* arguments may be a sequence or any iterable object; the result is always a list.

max(*iterable*[, *key*])
max(*arg1*, *arg2*, **args*[, *key*])
返回可迭代对象中最大的元素，或者返回两个及以上实参中最大的。

If one positional argument is provided, *iterable* must be a non-empty iterable (such as a non-empty string, tuple or list). The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

The optional *key* argument specifies a one-argument ordering function like that used for `list.sort()`. The *key* argument, if supplied, must be in keyword form (for example, `max(a,b,c,key=func)`).

在 2.5 版更改: Added support for the optional *key* argument.

memoryview(*obj*)
返回由给定实参创建的“内存视图”对象。有关详细信息，请参阅*memoryview type*。

min(*iterable*[, *key*])
min(*arg1*, *arg2*, **args*[, *key*])
返回可迭代对象中最小的元素，或者返回两个及以上实参中最小的。

If one positional argument is provided, *iterable* must be a non-empty iterable (such as a non-empty string, tuple or list). The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned.

The optional *key* argument specifies a one-argument ordering function like that used for `list.sort()`. The *key* argument, if supplied, must be in keyword form (for example, `min(a,b,c,key=func)`).

在 2.5 版更改: Added support for the optional *key* argument.

next(*iterator*[, *default*])
Retrieve the next item from the *iterator* by calling its *next()* method. If *default* is given, it is returned if the iterator is exhausted, otherwise *StopIteration* is raised.

2.6 新版功能.

class object
Return a new featureless object. *object* is a base for all new style classes. It has the methods that are common to all instances of new style classes.

2.2 新版功能.

在 2.3 版更改: This function does not accept any arguments. Formerly, it accepted arguments but ignored them.

oct(*x*)
Convert an integer number (of any size) to an octal string. The result is a valid Python expression.

在 2.4 版更改: Formerly only returned an unsigned literal.

open(*name*[, *mode*[, *buffering*]])
Open a file, returning an object of the *file* type described in section *File Objects*. If the file cannot be opened,

IOError is raised. When opening a file, it's preferable to use *open()* instead of invoking the *file* constructor directly.

The first two arguments are the same as for `stdio`'s `fopen()`: *name* is the file name to be opened, and *mode* is a string indicating how the file is to be opened.

The most commonly-used values of *mode* are `'r'` for reading, `'w'` for writing (truncating the file if it already exists), and `'a'` for appending (which on *some* Unix systems means that *all* writes append to the end of the file regardless of the current seek position). If *mode* is omitted, it defaults to `'r'`. The default is to use text mode, which may convert `'\n'` characters to a platform-specific representation on writing and back on reading. Thus, when opening a binary file, you should append `'b'` to the *mode* value to open the file in binary mode, which will improve portability. (Appending `'b'` is useful even on systems that don't treat binary and text files differently, where it serves as documentation.) See below for more possible values of *mode*.

The optional *buffering* argument specifies the file's desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size (in bytes). A negative *buffering* means to use the system default, which is usually line buffered for tty devices and fully buffered for other files. If omitted, the system default is used.[2]

Modes `'r+'`, `'w+'` and `'a+'` open the file for updating (reading and writing); note that `'w+'` truncates the file. Append `'b'` to the mode to open the file in binary mode, on systems that differentiate between binary and text files; on systems that don't have this distinction, adding the `'b'` has no effect.

In addition to the standard `fopen()` values *mode* may be `'U'` or `'rU'`. Python is usually built with *universal newlines* support; supplying `'U'` opens the file as a text file, but lines may be terminated by any of the following: the Unix end-of-line convention `'\n'`, the Macintosh convention `'\r'`, or the Windows convention `'\r\n'`. All of these external representations are seen as `'\n'` by the Python program. If Python is built without universal newlines support a *mode* with `'U'` is the same as normal text mode. Note that file objects so opened also have an attribute called `newlines` which has a value of `None` (if no newlines have yet been seen), `'\n'`, `'\r'`, `'\r\n'`, or a tuple containing all the newline types seen.

Python enforces that the mode, after stripping `'U'`, begins with `'r'`, `'w'` or `'a'`.

Python provides many file handling modules including *fileinput*, *os*, *os.path*, *tempfile*, and *shutil*.

在 2.5 版更改: Restriction on first letter of mode string introduced.

ord(*c*)

Given a string of length one, return an integer representing the Unicode code point of the character when the argument is a unicode object, or the value of the byte when the argument is an 8-bit string. For example, `ord('a')` returns the integer `97`, `ord(u'\u2020')` returns `8224`. This is the inverse of *chr()* for 8-bit strings and of *unichr()* for unicode objects. If a unicode argument is given and Python was built with UCS2 Unicode, then the character's code point must be in the range [0..65535] inclusive; otherwise the string length is two, and a *TypeError* will be raised.

pow(*x*, *y*[, *z*])

返回 *x* 的 *y* 次幂；如果 *z* 存在，则对 *z* 取余（比直接 `pow(x, y) % z` 计算更高效）。两个参数形式的 `pow(x, y)` 等价于幂运算符：`x**y`。

The arguments must have numeric types. 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.) If the second argument is negative, the third argument must be omitted. If *z* is present, *x* and *y* must be

[2] Specifying a buffer size currently has no effect on systems that don't have `setvbuf()`. The interface to specify the buffer size is not done using a method that calls `setvbuf()`, because that may dump core when called after any I/O has been performed, and there's no reliable way to determine whether this is the case.

of integer types, and *y* must be non-negative. (This restriction was added in Python 2.2. In Python 2.1 and before, floating 3-argument `pow()` returned platform-dependent results depending on floating-point rounding accidents.)

print(**objects*, *sep=' '*, *end='\n'*, *file=sys.stdout*)

Print *objects* to the stream *file*, separated by *sep* and followed by *end*. *sep*, *end* and *file*, if present, must be given as keyword arguments.

所有非关键字参数都会被转换为字符串，就像是执行了*str()* 一样，并会被写入到流，以 *sep* 且在末尾加上 *end*。*sep* 和 *end* 都必须为字符串；它们也可以为 None，这意味着使用默认值。如果没有给出 *objects*，则*print()* 将只写入 *end*。

The *file* argument must be an object with a `write(string)` method; if it is not present or None, *sys.stdout* will be used. Output buffering is determined by *file*. Use `file.flush()` to ensure, for instance, immediate appearance on a screen.

注解： This function is not normally available as a built-in since the name `print` is recognized as the `print` statement. To disable the statement and use the *print()* function, use this future statement at the top of your module:

```
from __future__ import print_function
```

2.6 新版功能.

class property([*fget*[, *fset*[, *fdel*[, *doc*]]]])

Return a property attribute for *new-style class*es (classes that derive from *object*).

fget 是获取属性值的函数。*fset* 是用于设置属性值的函数。*fdel* 是用于删除属性值的函数。并且 *doc* 为属性对象创建文档字符串。

一个典型的用法是定义一个托管属性 x:

```
class C(object):
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

如果 *c* 是 *C* 的实例，`c.x` 将调用 getter，`c.x = value` 将调用 setter，`del c.x` 将调用 deleter。

如果给出，*doc* 将成为该 property 属性的文档字符串。否则该 property 将拷贝 *fget* 的文档字符串（如果存在）。这令使用*property()* 作为*decorator* 来创建只读的特征属性可以很容易地实现:

```
class Parrot(object):
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
```

(下页继续)

(续上页)

```
        """Get the current voltage."""
        return self._voltage
```

以上 `@property` 装饰器会将 `voltage()` 方法转化为一个具有相同名称的只读属性的“getter”，并将 *voltage* 的文档字符串设置为“Get the current voltage.”

特征属性对象具有 `getter`, `setter` 以及 `deleter` 方法，它们可用作装饰器来创建该特征属性的副本，并将相应的访问函数设为所装饰的函数。这最好是用一个例子来解释:

```
class C(object):
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

上述代码与第一个例子完全等价。注意一定要给附加函数与原始的特征属性相同的名称 (在本例中为 `x`。)

返回的特征属性对象同样具有与构造器参数相对应的属性 `fget`, `fset` 和 `fdel`。

2.2 新版功能.

在 2.5 版更改: Use *fget*’s docstring if no *doc* given.

在 2.6 版更改: The `getter`, `setter`, and `deleter` attributes were added.

range(*stop*)
range(*start*, *stop*[, *step*])

This is a versatile function to create lists containing arithmetic progressions. It is most often used in `for` loops. The arguments must be plain integers. If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`. The full form returns a list of plain integers `[start, start + step, start + 2 * step, ...]`. If *step* is positive, the last element is the largest `start + i * step` less than *stop*; if *step* is negative, the last element is the smallest `start + i * step` greater than *stop*. *step* must not be zero (or else *ValueError* is raised). Example:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
```

(下页继续)

(续上页)

```
>>> range(1, 0)
[]
```

raw_input([*prompt*])

If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, *EOFError* is raised. Example:

```
>>> s = raw_input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

If the *readline* module was loaded, then *raw_input()* will use it to provide elaborate line editing and history features.

reduce(*function*, *iterable*[, *initializer*])

Apply *function* of two arguments cumulatively to the items of *iterable*, from left to right, so as to reduce the iterable to a single value. For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates `((((1+2)+3)+4)+5)`. The left argument, *x*, is the accumulated value and the right argument, *y*, is the update value from the *iterable*. If the optional *initializer* is present, it is placed before the items of the iterable in the calculation, and serves as a default when the iterable is empty. If *initializer* is not given and *iterable* contains only one item, the first item is returned. Roughly equivalent to:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        try:
            initializer = next(it)
        except StopIteration:
            raise TypeError('reduce() of empty sequence with no initial value')
    accum_value = initializer
    for x in it:
        accum_value = function(accum_value, x)
    return accum_value
```

reload(*module*)

Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (the same as the *module* argument).

When `reload(module)` is executed:

- Python modules' code is recompiled and the module-level code reexecuted, defining a new set of objects which are bound to names in the module's dictionary. The `init` function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does

not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects —with a `try` statement it can test for the table's presence and skip its initialization if desired:

```
try:
    cache
except NameError:
    cache = {}
```

It is generally not very useful to reload built-in or dynamically loaded modules. Reloading *sys*, *__main__*, `builtins` and other key modules is not recommended. In many cases extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from …import …`, calling *reload()* for the other module does not redefine the objects imported from it —one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (*module*.*name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances —they continue to use the old class definition. The same is true for derived classes.

repr(*object*)
Return a string containing a printable representation of an object. This is the same value yielded by conversions (reverse quotes). It is sometimes useful to be able to access this operation as an ordinary function. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to *eval()*, otherwise the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a `__repr__()` method.

reversed(*seq*)
返回一个反向的*iterator*。*seq* 必须是一个具有 `__reversed__()` 方法的对象或者是支持该序列协议(具有从 `0` 开始的整数类型参数的 `__len__()` 方法和 `__getitem__()` 方法)。

2.4 新版功能.

在 2.6 版更改: Added the possibility to write a custom `__reversed__()` method.

round(*number*[, *ndigits*])
Return the floating point value *number* rounded to *ndigits* digits after the decimal point. If *ndigits* is omitted, it defaults to zero. The result is a floating point number. Values are rounded to the closest multiple of 10 to the power minus *ndigits*; if two multiples are equally close, rounding is done away from 0 (so, for example, `round(0.5)` is `1.0` and `round(-0.5)` is `-1.0`).

注解: 对浮点数执行*round()* 的行为可能会令人惊讶: 例如, `round(2.675, 2)` 将给出 `2.67` 而不是期望的 `2.68`。这不算是程序错误: 这一结果是由于大多数十进制小数实际上都不能以浮点数精确地表示。请参阅 tut-fp-issues 了解更多信息。

---

class set([*iterable*])
返回一个新的*set* 对象, 可以选择带有从 *iterable* 获取的元素。`set` 是一个内置类型。请查看*set* 和集合类型—*set, frozenset* 获取关于这个类的文档。

有关其他容器请参看内置的*frozenset*, `list`, *tuple* 和*dict* 类, 以及*collections* 模块。

2.4 新版功能.

setattr(*object*, *name*, *value*)
此函数与*getattr()* 两相对应。其参数为一个对象、一个字符串和一个任意值。字符串指定一个现有属性或者新增属性。函数会将值赋给该属性, 只要对象允许这种操作。例如, `setattr(x, 'foobar', 123)` 等价于 `x.foobar = 123`。

class slice(*stop*)
class slice(*start*, *stop*[, *step*])
返回一个表示由 `range(start, stop, step)` 所指定索引集的*slice* 对象。其中 *start* 和 *step* 参数默认为 `None`。切片对象具有仅会返回对应参数值（或其默认值）的只读数据属性 `start`, `stop` 和 `step`。它们没有其他的显式功能；不过它们会被 NumPy 以及其他第三方扩展所使用。切片对象也会在使用扩展索引语法时被生成。例如: `a[start:stop:step]` 或 `a[start:stop, i]`。请参阅*itertools.islice()* 了解返回迭代器的一种替代版本。

sorted(*iterable*[, *cmp*[, *key*[, *reverse*]]])
根据 *iterable* 中的项返回一个新的已排序列表。

The optional arguments *cmp*, *key*, and *reverse* have the same meaning as those for the `list.sort()` method (described in section 可变序列类型).

cmp specifies a custom comparison function of two arguments (iterable elements) which should return a negative, zero or positive number depending on whether the first argument is considered smaller than, equal to, or larger than the second argument: `cmp=lambda x,y: cmp(x.lower(), y.lower())`. The default value is `None`.

key specifies a function of one argument that is used to extract a comparison key from each list element: `key=str.lower`. The default value is `None` (compare the elements directly).

reverse 为一个布尔值。如果设为 `True`，则每个列表元素将按反向顺序比较进行排序。

In general, the *key* and *reverse* conversion processes are much faster than specifying an equivalent *cmp* function. This is because *cmp* is called multiple times for each list element while *key* and *reverse* touch each element only once. Use *functools.cmp_to_key()* to convert an old-style *cmp* function to a *key* function.

内置的*sorted()* 确保是稳定的。如果一个排序确保不会改变比较结果相等的元素的相对顺序就称其为稳定的—这有利于进行多重排序（例如先按部门、再按薪级排序）。

有关排序示例和简要排序教程，请参阅 sortinghowto 。

2.4 新版功能.

staticmethod(*function*)
Return a static method for *function*.

静态方法不会接收隐式的第一个参数。要声明一个静态方法，请使用此语法

```
class C(object):
    @staticmethod
    def f(arg1, arg2, ...):
        ...
```

`@staticmethod` 这样的形式称为函数的*decorator* –详情参阅 function。

静态方法的调用可以在类上进行 (例如 `C.f()`) 也可以在实例上进行 (例如 `C().f()`)。

Python 中的静态方法与 Java 或 C++ 中的静态方法类似。另请参阅*classmethod()* ，用于创建备用类构造函数的变体。

想了解更多有关静态方法的信息，请参阅 types 。

2.2 新版功能.

在 2.4 版更改: Function decorator syntax added.

class str(*object=''*)
Return a string containing a nicely printable representation of an object. For strings, this returns the string itself. The difference with `repr(object)` is that `str(object)` does not always attempt to return a string that is acceptable to *eval()*; its goal is to return a printable string. If no argument is given, returns the empty string, `''`.

For more information on strings see *Sequence Types —str, unicode, list, tuple, bytearray, buffer, xrange* which describes sequence functionality (strings are sequences), and also the string-specific methods described in the 字符串的方法 section. To output formatted strings use template strings or the `%` operator described in the *String Formatting Operations* section. In addition see the *String Services* section. See also `unicode()`.

sum(*iterable*[, *start*])
从 *start* 开始自左向右对 *iterable* 中的项求和并返回总计值。*start* 默认为 `0`。*iterable* 的项通常为数字，开始值则不允许为字符串。

对某些用例来说，存在`sum()` 的更好替代。拼接字符串序列的更好更快方式是调用 `''.join(sequence)`。要以扩展精度对浮点值求和，请参阅`math.fsum()`。要拼接一系列可迭代对象，请考虑使用`itertools.chain()`。

2.3 新版功能.

super(*type*[, *object-or-type*])
返回一个代理对象，它会将方法调用委托给 *type* 指定的父类或兄弟类。这对于访问已在类中被重载的继承方法很有用。搜索顺序与`getattr()` 所使用的相同，只是 *type* 指定的类型本身会被跳过。

type 的`__mro__` 属性列出了`getattr()` 和`super()` 所使用的方法解析顺序。该属性是动态的，可以在继承层级结构更新的时候任意改变。

如果省略第二个参数，则返回的超类对象是未绑定的。如果第二个参数为一个对象，则 `isinstance(obj, type)` 必须为真值。如果第二个参数为一个类型，则 `issubclass(type2, type)` 必须为真值（这适用于类方法）。

注解： `super()` only works for *new-style class*es.

---

super 有两个典型用例。在具有单继承的类层级结构中，*super* 可用来引用父类而不必显式地指定它们的名称，从而令代码更易维护。这种用法与其他编程语言中 *super* 的用法非常相似。

第二个用例是在动态执行环境中支持协作多重继承。此用例为 Python 所独有，在静态编译语言或仅支持单继承的语言中是不存在的。这使得实现“菱形图”成为可能，在这时会有多个基类实现相同的方法。好的设计强制要求这种方法在每个情况下具有相同的调用签名（因为调用顺序是在运行时确定的，也因为该顺序要适应类层级结构的更改，还因为该顺序可能包含在运行时之前未知的兄弟类）。

对于以上两个用例，典型的超类调用看起来是这样的:

```
class C(B):
    def method(self, arg):
        super(C, self).method(arg)
```

请注意`super()` 是作为显式加点属性查找的绑定过程的一部分来实现的，例如 `super().__getitem__(name)`。它做到这一点是通过实现自己的 `__getattribute__()` 方法，这样就能以可预测的顺序搜索类，并且支持协作多重继承。对应地，`super()` 在像 `super()[name]` 这样使用语句或操作符进行隐式查找时则未被定义。

Also note that `super()` is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references.

对于有关如何使用`super()` 来如何设计协作类的实用建议，请参阅 使用 super() 的指南。

2.2 新版功能.

tuple([*iterable*])
Return a tuple whose items are the same and in the same order as *iterable*’s items. *iterable* may be a sequence, a container that supports iteration, or an iterator object. If *iterable* is already a tuple, it is returned unchanged. For instance, `tuple('abc')` returns `('a', 'b', 'c')` and `tuple([1, 2, 3])` returns `(1, 2, 3)`. If no argument is given, returns a new empty tuple, `()`.

tuple is an immutable sequence type, as documented in *Sequence Types —str, unicode, list, tuple, bytearray, buffer, xrange*. For other containers see the built in *dict*, list, and *set* classes, and the *collections* module.

class type(*object*)
class type(*name*, *bases*, *dict*)
With one argument, return the type of an *object*. The return value is a type object. The *isinstance()* built-in function is recommended for testing the type of an object.

With three arguments, return a new type object. This is essentially a dynamic form of the class statement. The *name* string is the class name and becomes the *__name__* attribute; the *bases* tuple itemizes the base classes and becomes the *__bases__* attribute; and the *dict* dictionary is the namespace containing definitions for class body and becomes the *__dict__* attribute. For example, the following two statements create identical *type* objects:

```
>>> class X(object):
...     a = 1
...
>>> X = type('X', (object,), dict(a=1))
```

2.2 新版功能.

unichr(*i*)
Return the Unicode string of one character whose Unicode code is the integer *i*. For example, unichr(97) returns the string u'a'. This is the inverse of *ord()* for Unicode strings. The valid range for the argument depends how Python was configured –it may be either UCS2 [0..0xFFFF] or UCS4 [0..0x10FFFF]. *ValueError* is raised otherwise. For ASCII and 8-bit strings see *chr()*.

2.0 新版功能.

unicode(*object*='')
unicode(*object*[, *encoding*[, *errors*]])
Return the Unicode string version of *object* using one of the following modes:

If *encoding* and/or *errors* are given, unicode() will decode the object which can either be an 8-bit string or a character buffer using the codec for *encoding*. The *encoding* parameter is a string giving the name of an encoding; if the encoding is not known, *LookupError* is raised. Error handling is done according to *errors*; this specifies the treatment of characters which are invalid in the input encoding. If *errors* is 'strict' (the default), a *ValueError* is raised on errors, while a value of 'ignore' causes errors to be silently ignored, and a value of 'replace' causes the official Unicode replacement character, U+FFFD, to be used to replace input characters which cannot be decoded. See also the *codecs* module.

If no optional parameters are given, unicode() will mimic the behaviour of str() except that it returns Unicode strings instead of 8-bit strings. More precisely, if *object* is a Unicode string or subclass it will return that Unicode string without any additional decoding applied.

For objects which provide a __unicode__() method, it will call this method without arguments to create a Unicode string. For all other objects, the 8-bit string version or representation is requested and then converted to a Unicode string using the codec for the default encoding in 'strict' mode.

For more information on Unicode strings see *Sequence Types —str, unicode, list, tuple, bytearray, buffer, xrange* which describes sequence functionality (Unicode strings are sequences), and also the string-specific methods described in the *字符串的方法* section. To output formatted strings use template strings or the % operator described in the *String Formatting Operations* section. In addition see the *String Services* section. See also *str()*.

2.0 新版功能.

在 2.2 版更改: Support for __unicode__() added.

vars([*object*])
返回模块、类、实例或任何其它具有*__dict__* 属性的对象的*__dict__* 属性。

Objects such as modules and instances have an updateable *__dict__* attribute; however, other objects may have write restrictions on their *__dict__* attributes (for example, new-style classes use a dictproxy to prevent direct dictionary updates).

不带参数时，*vars()* 的行为类似*locals()*。请注意，locals 字典仅对于读取起作用，因为对 locals 字典的更新会被忽略。

xrange(*stop*)
xrange(*start*, *stop*[, *step*])

This function is very similar to *range()*, but returns an *xrange object* instead of a list. This is an opaque sequence type which yields the same values as the corresponding list, without actually storing them all simultaneously. The advantage of *xrange()* over *range()* is minimal (since *xrange()* still has to create the values when asked for them) except when a very large range is used on a memory-starved machine or when all of the range's elements are never used (such as when the loop is usually terminated with `break`). For more information on xrange objects, see *XRange Type* and *Sequence Types —str, unicode, list, tuple, bytearray, buffer, xrange*.

CPython implementation detail: *xrange()* is intended to be simple and fast. Implementations may impose restrictions to achieve this. The C implementation of Python restricts all arguments to native C longs ("short" Python integers), and also requires that the number of elements fit in a native C long. If a larger range is needed, an alternate version can be crafted using the *itertools* module: `islice(count(start, step), (stop-start+step-1+2*(step<0))//step)`.

zip([*iterable*, ...])

This function returns a list of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables. The returned list is truncated in length to the length of the shortest argument sequence. When there are multiple arguments which are all of the same length, *zip()* is similar to *map()* with an initial argument of `None`. With a single sequence argument, it returns a list of 1-tuples. With no arguments, it returns an empty list.

The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using `zip(*[iter(s)]*n)`.

zip() 与 * 运算符相结合可以用来拆解一个列表:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> zipped
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zipped)
>>> x == list(x2) and y == list(y2)
True
```

2.0 新版功能.

在 2.4 版更改: Formerly, *zip()* required at least one argument and `zip()` raised a *TypeError* instead of returning an empty list.

__import__(*name*[, *globals*[, *locals*[, *fromlist*[, *level*]]]])

注解: 与*importlib.import_module()* 不同，这是一个日常 Python 编程中不需要用到的高级函数。

This function is invoked by the `import` statement. It can be replaced (by importing the *__builtin__* module and assigning to `__builtin__.__import__`) in order to change semantics of the `import` statement, but nowadays it is usually simpler to use import hooks (see **PEP 302**). Direct use of *__import__()* is rare, except in cases where you want to import a module whose name is only known at runtime.

该函数会导入 *name* 模块，有可能使用给定的 *globals* 和 *locals* 来确定如何在包的上下文中解读名称。*fromlist* 给出了应该从由 *name* 指定的模块导入对象或子模块的名称。标准实现完全不使用其 *locals* 参数，而仅使用 *globals* 参数来确定 `import` 语句的包上下文。

level specifies whether to use absolute or relative imports. The default is `-1` which indicates both absolute and relative imports will be attempted. `0` means only perform absolute imports. Positive values for *level* indicate the number of parent directories to search relative to the directory of the module calling *__import__()*.

当 *name* 变量的形式为 `package.module` 时，通常将会返回最高层级的包（第一个点号之前的名称），而 不是以 *name* 命名的模块。但是，当给出了非空的 *fromlist* 参数时，则将返回以 *name* 命名的模块。

例如，语句 `import spam` 的结果将为与以下代码作用相同的字节码:

```
spam = __import__('spam', globals(), locals(), [], -1)
```

语句 `import spam.ham` 的结果将为以下调用:

```
spam = __import__('spam.ham', globals(), locals(), [], -1)
```

请注意在这里*__import__()* 是如何返回顶层模块的，因为这是通过 `import` 语句被绑定到特定名称的对象。

另一方面，语句 `from spam.ham import eggs, sausage as saus` 的结果将为

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], -1)
eggs = _temp.eggs
saus = _temp.sausage
```

在这里，`spam.ham` 模块会由*__import__()* 返回。要导入的对象将从此对象中提取并赋值给它们对应的名称。

如果您只想按名称导入模块（可能在包中），请使用*importlib.import_module()*

在 2.5 版更改: The level parameter was added.

在 2.5 版更改: Keyword support for parameters was added.

CHAPTER 3

Non-essential Built-in Functions

There are several built-in functions that are no longer essential to learn, know or use in modern Python programming. They have been kept here to maintain backwards compatibility with programs written for older versions of Python.

Python programmers, trainers, students and book writers should feel free to bypass these functions without concerns about missing something important.

apply(*function*, *args*[, *keywords*])
The *function* argument must be a callable object (a user-defined or built-in function or method, or a class object) and the *args* argument must be a sequence. The *function* is called with *args* as the argument list; the number of arguments is the length of the tuple. If the optional *keywords* argument is present, it must be a dictionary whose keys are strings. It specifies keyword arguments to be added to the end of the argument list. Calling *apply()* is different from just calling `function(args)`, since in that case there is always exactly one argument. The use of *apply()* is equivalent to `function(*args, **keywords)`.

2.3 版后已移除: Use `function(*args, **keywords)` instead of `apply(function, args, keywords)` (see tut-unpacking-arguments).

buffer(*object*[, *offset*[, *size*]])
The *object* argument must be an object that supports the buffer call interface (such as strings, arrays, and buffers). A new buffer object will be created which references the *object* argument. The buffer object will be a slice from the beginning of *object* (or from the specified *offset*). The slice will extend to the end of *object* (or will have a length given by the *size* argument).

coerce(*x*, *y*)
Return a tuple consisting of the two numeric arguments converted to a common type, using the same rules as used by arithmetic operations. If coercion is not possible, raise *TypeError*.

intern(*string*)
Enter *string* in the table of "interned" strings and return the interned string –which is *string* itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup –if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys.

在 2.3 版更改: Interned strings are not immortal (like they used to be in Python 2.2 and before); you must keep a reference to the return value of *intern()* around to benefit from it.

备注

CHAPTER 4

内置常量

有少数的常量存在于内置命名空间中。它们是：

False
The false value of the *bool* type.

2.3 新版功能.

True
The true value of the *bool* type.

2.3 新版功能.

None
The sole value of *types.NoneType*. None is frequently used to represent the absence of a value, as when default arguments are not passed to a function.

在 2.4 版更改: Assignments to None are illegal and raise a *SyntaxError*.

NotImplemented
Special value which can be returned by the "rich comparison" special methods (__eq__(), __lt__(), and friends), to indicate that the comparison is not implemented with respect to the other type.

Ellipsis
Special value used in conjunction with extended slicing syntax.

__debug__
如果 Python 没有以 -O 选项启动，则此常量为真值。另请参见 assert 语句。

注解: The names *None* and *__debug__* cannot be reassigned (assignments to them, even as an attribute name, raise *SyntaxError*), so they can be considered "true" constants.

在 2.7 版更改: Assignments to __debug__ as an attribute became illegal.

4.1 由 site 模块添加的常量

site 模块（在启动期间自动导入，除非给出 -S 命令行选项）将几个常量添加到内置命名空间。它们对交互式解释器 shell 很有用，并且不应在程序中使用。

quit([*code=None*])
exit([*code=None*])
当打印此对象时，会打印出一条消息，例如 “Use quit() or Ctrl-D (i.e. EOF) to exit”，当调用此对象时，将使用指定的退出代码来引发*SystemExit*。

copyright
credits
打印或调用的对象分别打印版权或作者的文本。

license
当打印此对象时，会打印出一条消息 “Type license() to see the full license text”，当调用此对象时，将以分页形式显示完整的许可证文本（每次显示一屏）。

CHAPTER 5

内置类型

以下部分描述了解释器中内置的标准类型。

注解： Historically (until release 2.2), Python’s built-in types have differed from user-defined types because it was not possible to use the built-in types as the basis for object-oriented inheritance. This limitation no longer exists.

The principal built-in types are numerics, sequences, mappings, files, classes, instances and exceptions.

Some operations are supported by several object types; in particular, practically all objects can be compared, tested for truth value, and converted to a string (with the *repr()* function or the slightly different `str()` function). The latter function is implicitly used when an object is written by the `print()` function.

5.1 逻辑值检测

Any object can be tested for truth value, for use in an `if` or `while` condition or as operand of the Boolean operations below. The following values are considered false:

- `None`
- `False`
- zero of any numeric type, for example, `0`, `0L`, `0.0`, `0j`.
- any empty sequence, for example, `''`, `()`, `[]`.
- any empty mapping, for example, `{}`.
- instances of user-defined classes, if the class defines a `__nonzero__()` or `__len__()` method, when that method returns the integer zero or `bool` value `False`.[1]

[1] 有关这些特殊方法的额外信息可参看 Python 参考指南 (customization)。

All other values are considered true —so objects of many types are always true.

产生布尔值结果的运算和内置函数总是返回 0 或 False 作为假值，1 或 True 作为真值，除非另行说明。(重要例外：布尔运算 or 和 and 总是返回其中一个操作数。)

5.2 Boolean Operations —and, or, not

这些属于布尔运算，按优先级升序排列:

运算	结果:	注释
x or y	if *x* is false, then *y*, else *x*	(1)
x and y	if *x* is false, then *x*, else *y*	(2)
not x	if *x* is false, then True, else False	(3)

注释:

(1) 这是个短路运算符，因此只有在第一个参数为假值时才会对第二个参数求值。

(2) 这是个短路运算符，因此只有在第一个参数为真值时才会对第二个参数求值。

(3) not 的优先级比非布尔运算符低，因此 not a == b 会被解读为 not (a == b) 而 a == not b 会引发语法错误。

5.3 比较运算

Comparison operations are supported by all objects. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, 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).

此表格汇总了比较运算:

运算	含义	注释
<	严格小于	
<=	小于或等于	
>	严格大于	
>=	大于或等于	
==	等于	
!=	不等于	(1)
is	对象标识	
is not	否定的对象标识	

注释:

(1) != can also be written <>, but this is an obsolete usage kept for backwards compatibility only. New code should always use !=.

Objects of different types, except different numeric types and different string types, never compare equal; such objects are ordered consistently but arbitrarily (so that sorting a heterogeneous array yields a consistent result). Furthermore, some types (for example, file objects) support only a degenerate notion of comparison where any two objects of that type are unequal. Again, such objects are ordered arbitrarily but consistently. The <, <=, > and >= operators will raise a *TypeError* exception when any operand is a complex number.

Non-identical instances of a class normally compare as non-equal unless the class defines the `__eq__()` method or the `__cmp__()` method.

Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines either enough of the rich comparison methods (`__lt__()`, `__le__()`, `__gt__()`, and `__ge__()`) or the `__cmp__()` method.

CPython implementation detail: Objects of different types except numbers are ordered by their type names; objects of the same types that don't support proper comparison are ordered by their address.

Two more operations with the same syntactic priority, `in` and `not in`, are supported only by sequence types (below).

5.4 Numeric Types — `int`, `float`, `long`, `complex`

There are four distinct numeric types: *plain integers*, *long integers*, *floating point numbers*, and *complex numbers*. In addition, Booleans are a subtype of plain integers. Plain integers (also just called *integers*) are implemented using `long` in C, which gives them at least 32 bits of precision (`sys.maxint` is always set to the maximum plain integer value for the current platform, the minimum value is `-sys.maxint - 1`). Long integers have unlimited precision. Floating point numbers are usually implemented using `double` in C; information about the precision and internal representation of floating point numbers for the machine on which your program is running is available in *`sys.float_info`*. Complex numbers have a real and imaginary part, which are each a floating point number. To extract these parts from a complex number *z*, use `z.real` and `z.imag`. (The standard library includes additional numeric types, *`fractions`* that hold rationals, and *`decimal`* that hold floating-point numbers with user-definable precision.)

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including binary, hex, and octal numbers) yield plain integers unless the value they denote is too large to be represented as a plain integer, in which case they yield a long integer. Integer literals with an `'L'` or `'l'` suffix yield long integers (`'L'` is preferred because `1l` looks too much like eleven!). Numeric literals containing a decimal point or an exponent sign yield floating point numbers. Appending `'j'` or `'J'` to a numeric literal yields an imaginary number (a complex number with a zero real part) which you can add to an integer or float to get a complex number with real and imaginary parts.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the "narrower" type is widened to that of the other, where plain integer is narrower than long integer is narrower than floating point is narrower than complex. Comparisons between numbers of mixed type use the same rule.[2] The constructors *`int()`*, *`long()`*, *`float()`*, and *`complex()`* can be used to produce numbers of a specific type.

All built-in numeric types support the following operations. See power and later sections for the operators' priorities.

[2] 作为结果，列表 `[1, 2]` 与 `[1.0, 2.0]` 是相等的，元组的情况也类似。

运算	结果:	注释
`x + y`	*x* 和 *y* 的和	
`x - y`	*x* 和 *y* 的差	
`x * y`	*x* 和 *y* 的乘积	
`x / y`	*x* 和 *y* 的商	(1)
`x // y`	(floored) quotient of *x* and *y*	(4)(5)
`x % y`	remainder of `x / y`	(4)
`-x`	*x* 取反	
`+x`	*x* 不变	
`abs(x)`	*x* 的绝对值或大小	(3)
`int(x)`	将 *x* 转换为整数	(2)
`long(x)`	*x* converted to long integer	(2)
`float(x)`	将 *x* 转换为浮点数	(6)
`complex(re,im)`	一个带有实部 *re* 和虚部 *im* 的复数。*im* 默认为 0。	
`c.conjugate()`	conjugate of the complex number *c*. (Identity on real numbers)	
`divmod(x, y)`	`(x // y, x % y)`	(3)(4)
`pow(x, y)`	*x* 的 *y* 次幂	(3)(7)
`x ** y`	*x* 的 *y* 次幂	(7)

注释:

(1) For (plain or long) integer division, the result is an integer. The result is always rounded towards minus infinity: 1/2 is 0, (-1)/2 is -1, 1/(-2) is -1, and (-1)/(-2) is 0. Note that the result is a long integer if either operand is a long integer, regardless of the numeric value.

(2) Conversion from floats using *int()* or *long()* truncates toward zero like the related function, *math.trunc()*. Use the function *math.floor()* to round downward and *math.ceil()* to round upward.

(3) See 内置函数 for a full description.

(4) 2.3 版后已移除: 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.

(5) Also referred to as integer division. The resultant value is a whole integer, though the result' s type is not necessarily int.

(6) float 也接受字符串 "nan" 和附带可选前缀 "+" 或 "-" 的 "inf" 分别表示非数字 (NaN) 以及正或负无穷。

2.6 新版功能.

(7) Python 将 `pow(0, 0)` 和 `0 ** 0` 定义为 `1`，这是编程语言的普遍做法。

All *numbers.Real* types (*int*, *long*, and *float*) also include the following operations:

运算	结果:
math.trunc(x)	*x* 截断为*Integral*
round(x[, n])	*x* rounded to *n* digits, rounding ties away from zero. If *n* is omitted, it defaults to 0.
math.floor(x)	the greatest integer as a float <= *x*
math.ceil(x)	the least integer as a float >= *x*

5.4.1 整数类型的按位运算

Bitwise operations only make sense for integers. Negative numbers are treated as their 2's complement value (this assumes a sufficiently large number of bits that no overflow occurs during the operation).

二进制按位运算的优先级全都低于数字运算，但又高于比较运算；一元运算 ~ 具有与其他一元算术运算 (+ and −) 相同的优先级。

此表格是以优先级升序排序的按位运算列表:

运算	结果:	注释
`x	y`	*x* 和 *y* 按位 或	
`x ^ y`	*x* 和 *y* 按位 异或	
`x & y`	*x* 和 *y* 按位 与	
`x << n`	*x* 左移 *n* 位	(1)(2)
`x >> n`	*x* 右移 *n* 位	(1)(3)
`~x`	*x* 逐位取反	

注释:

(1) 负的移位数是非法的，会导致引发*`ValueError`*。

(2) A left shift by *n* bits is equivalent to multiplication by `pow(2, n)`. A long integer is returned if the result exceeds the range of plain integers.

(3) A right shift by *n* bits is equivalent to division by `pow(2, n)`.

5.4.2 整数类型的附加方法

The integer types implement the *`numbers.Integral` abstract base class*. In addition, they provide one more method:

`int`.**`bit_length`**()

`long`.**`bit_length`**()

返回以二进制表示一个整数所需要的位数，不包括符号位和前面的零:

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

更准确地说，如果 `x` 非零，则 `x.bit_length()` 是使得 `2**(k-1) <= abs(x) < 2**k` 的唯一正整数 `k`。同样地，当 `abs(x)` 小到足以具有正确的舍入对数时，则 `k = 1 + int(log(abs(x), 2))`。如果 `x` 为零，则 `x.bit_length()` 返回 `0`。

等价于:

```
def bit_length(self):
    s = bin(self)       # binary representation:  bin(-37) --> '-0b100101'
    s = s.lstrip('-0b') # remove leading zeros and minus sign
    return len(s)       # len('100101') --> 6
```

2.7 新版功能.

5.4.3 浮点类型的附加方法

float 类型实现了*numbers.Real abstract base class*。float 还具有以下附加方法。

`float.`**`as_integer_ratio`**`()`
返回一对整数，其比率正好等于原浮点数并且分母为正数。无穷大会引发*OverflowError* 而 NaN 则会引发*ValueError*。

2.6 新版功能.

`float.`**`is_integer`**`()`
如果 float 实例可用有限位整数表示则返回 `True`，否则返回 `False`:

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

2.6 新版功能.

两个方法均支持与十六进制数字符串之间的转换。由于 Python 浮点数在内部存储为二进制数，因此浮点数与十进制数字符串之间的转换往往会导致微小的舍入错误。而十六进制数字符串却允许精确地表示和描述浮点数。这在进行调试和数值工作时非常有用。

`float.`**`hex`**`()`
以十六进制字符串的形式返回一个浮点数表示。对于有限浮点数，这种表示法将总是包含前导的 `0x` 和尾随的 `p` 加指数。

2.6 新版功能.

`float.`**`fromhex`**`(`*s*`)`
返回以十六进制字符串 *s* 表示的浮点数的类方法。字符串 *s* 可以带有前导和尾随的空格。

2.6 新版功能.

请注意*float.hex()* 是实例方法，而*float.fromhex()* 是类方法。

十六进制字符串采用的形式为:

```
[sign] ['0x'] integer ['.' fraction] ['p' exponent]
```

可选的 `sign` 可以是 `+` 或 `-`，`integer` 和 `fraction` 是十六进制数码组成的字符串，`exponent` 是带有可选前导符的十进制整数。大小写没有影响，在 integer 或 fraction 中必须至少有一个十六进制数码。此语法类似于 C99 标准的 6.4.4.2 小节中所描述的语法，也是 Java 1.5 以上所使用的语法。特别地，*float.hex()* 的输出可以用作 C 或 Java 代码中的十六进制浮点数字面值，而由 C 的 `%a` 格式字符或 Java 的 `Double.toHexString` 所生成的十六进制数字符串由为*float.fromhex()* 所接受。

请注意 exponent 是十进制数而非十六进制数，它给出要与系数相乘的 2 的幂次。例如，十六进制数字符串 `0x3.a7p10` 表示浮点数 `(3 + 10./16 + 7./16**2) * 2.0**10` 即 `3740.0`:

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

对 `3740.0` 应用反向转换会得到另一个代表相同数值的十六进制数字符串:

```
>>> float.hex(3740.0)
'0x1.d380000000000p+11'
```

5.5 迭代器类型

2.2 新版功能.

Python 支持在容器中进行迭代的概念。这是通过使用两个单独方法来实现的；它们被用于允许用户自定义类对迭代的支持。将在下文中详细描述的序列总是支持迭代方法。

容器对象要提供迭代支持，必须定义一个方法:

`container.__iter__()`
返回一个迭代器对象。该对象需要支持下文所述的迭代器协议。如果容器支持不同的迭代类型，则可以提供额外的方法来专门地请求不同迭代类型的迭代器。(支持多种迭代形式的对象的例子有同时支持广度优先和深度优先遍历的树结构。) 此方法对应于 Python/C API 中 Python 对象类型结构体的 `tp_iter` 槽位。

迭代器对象自身需要支持以下两个方法，它们共同组成了 *迭代器协议*:

`iterator.__iter__()`
返回迭代器对象本身。这是同时允许容器和迭代器配合 `for` 和 `in` 语句使用所必须的。此方法对应于 Python/C API 中 Python 对象类型结构体的 `tp_iter` 槽位。

`iterator.next()`
从容器中返回下一项。如果已经没有项可返回，则会引发 *StopIteration* 异常。此方法对应于 Python/C API 中 Python 对象类型结构体的 `tp_iternext` 槽位。

Python 定义了几种迭代器对象以支持对一般和特定序列类型、字典和其他更特别的形式进行迭代。除了迭代器协议的实现，特定类型的其他性质对迭代操作来说都不重要。

The intention of the protocol is that once an iterator's *next()* method raises *StopIteration*, it will continue to do so on subsequent calls. Implementations that do not obey this property are deemed broken. (This constraint was added in Python 2.3; in Python 2.2, various iterators are broken according to this rule.)

5.5.1 生成器类型

Python's *generator*s provide a convenient way to implement the iterator protocol. If a container object's `__iter__()` method is implemented as a generator, it will automatically return an iterator object (technically, a generator object) supplying the *__iter__()* and *next()* methods. More information about generators can be found in the documentation for the yield expression.

5.6 Sequence Types —str, unicode, list, tuple, bytearray, buffer, xrange

There are seven sequence types: strings, Unicode strings, lists, tuples, bytearrays, buffers, and xrange objects.

For other containers see the built in *dict* and *set* classes, and the *collections* module.

String literals are written in single or double quotes: `'xyzzy'`, `"frobozz"`. See strings for more about string literals. Unicode strings are much like strings, but are specified in the syntax using a preceding `'u'` character: `u'abc'`, `u"def"`. In addition to the functionality described here, there are also string-specific methods described in the 字符串的方法 section. Lists are constructed with square brackets, separating items with commas: `[a, b, c]`. Tuples are constructed by the comma operator (not within square brackets), with or without enclosing parentheses, but an empty tuple must have the enclosing parentheses, such as `a, b, c` or `()`. A single item tuple must have a trailing comma, such as `(d,)`.

Bytearray objects are created with the built-in function *bytearray()*.

Buffer objects are not directly supported by Python syntax, but can be created by calling the built-in function *buffer()*. They don't support concatenation or repetition.

Objects of type xrange are similar to buffers in that there is no specific syntax to create them, but they are created using the *xrange()* function. They don't support slicing, concatenation or repetition, and using `in`, `not in`, *min()* or *max()* on them is inefficient.

Most sequence types support the following operations. The `in` and `not in` operations have the same priorities as the comparison operations. The `+` and `*` operations have the same priority as the corresponding numeric operations.[3] Additional methods are provided for 可变序列类型.

This table lists the sequence operations sorted in ascending priority. In the table, *s* and *t* are sequences of the same type; *n*, *i* and *j* are integers:

运算	结果:	注释
`x in s`	如果 *s* 中的某项等于 *x* 则结果为 `True`，否则为 `False`	(1)
`x not in s`	如果 *s* 中的某项等于 *x* 则结果为 `False`，否则为 `True`	(1)
`s + t`	*s* 与 *t* 相拼接	(6)
`s * n, n * s`	相当于 *s* 与自身进行 *n* 次拼接	(2)
`s[i]`	*s* 的第 *i* 项，起始为 0	(3)
`s[i:j]`	*s* 从 *i* 到 *j* 的切片	(3)(4)
`s[i:j:k]`	*s* 从 *i* 到 *j* 步长为 *k* 的切片	(3)(5)
`len(s)`	*s* 的长度	
`min(s)`	*s* 的最小项	
`max(s)`	*s* 的最大项	
`s.index(x)`	index of the first occurrence of *x* in *s*	
`s.count(x)`	*x* 在 *s* 中出现的总次数	

Sequence types also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see comparisons in the language reference.)

注释:

(1) When *s* is a string or Unicode string object the `in` and `not in` operations act like a substring test. In Python versions before 2.3, *x* had to be a string of length 1. In Python 2.3 and beyond, *x* may be a string of any length.

(2) Values of *n* less than `0` are treated as `0` (which yields an empty sequence of the same type as *s*). Note that items in the sequence *s* are not copied; they are referenced multiple times. This often haunts new Python programmers; consider:

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

What has happened is that `[[]]` is a one-element list containing an empty list, so all three elements of `[[]] * 3` are references to this single empty list. Modifying any of the elements of `lists` modifies this single list. You can create a list of different lists this way:

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
```

(下页继续)

[3] 它们必须如此，因为解析器无法区分这些操作数的类型。

(续上页)

```
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

进一步的解释可以在 FAQ 条目 faq-multidimensional-list 中查看。

(3) 如果 *i* 或 *j* 为负值，则索引顺序是相对于序列 *s* 的末尾: 索引号会被替换为 `len(s) + i` 或 `len(s) + j`。但要注意 `-0` 仍然为 `0`。

(4) *s* 从 *i* 到 *j* 的切片被定义为所有满足 `i <= k < j` 的索引号 *k* 的项组成的序列。如果 *i* 或 *j* 大于 `len(s)`，则使用 `len(s)`。如果 *i* 被省略或为 `None`，则使用 `0`。如果 *j* 被省略或为 `None`，则使用 `len(s)`。如果 *i* 大于等于 *j*，则切片为空。

(5) *s* 从 *i* 到 *j* 步长为 *k* 的切片被定义为所有满足 `0 <= n < (j-i)/k` 的索引号 `x = i + n*k` 的项组成的序列。换句话说，索引号为 `i`, `i+k`, `i+2*k`, `i+3*k`，以此类推，当达到 *j* 时停止 (但一定不包括 *j*)。当 *k* 为正值时，*i* 和 *j* 会被减至不大于 `len(s)`。当 *k* 为负值时，*i* 和 *j* 会被减至不大于 `len(s) - 1`。如果 *i* 或 *j* 被省略或为 `None`，它们会成为“终止”值 (是哪一端的终止值则取决于 *k* 的符号)。请注意，*k* 不可为零。如果 *k* 为 `None`，则当作 `1` 处理。

(6) **CPython implementation detail:** If *s* and *t* are both strings, some Python implementations such as CPython can usually perform an in-place optimization for assignments of the form `s = s + t` or `s += t`. When applicable, this optimization makes quadratic run-time much less likely. This optimization is both version and implementation dependent. For performance sensitive code, it is preferable to use the *`str.join()`* method which assures consistent linear concatenation performance across versions and implementations.

在 2.4 版更改: Formerly, string concatenation never occurred in-place.

5.6.1 字符串的方法

Below are listed the string methods which both 8-bit strings and Unicode objects support. Some of them are also available on *`bytearray`* objects.

In addition, Python’s strings support the sequence type methods described in the *Sequence Types —str, unicode, list, tuple, bytearray, buffer, xrange* section. To output formatted strings use template strings or the `%` operator described in the *String Formatting Operations* section. Also, see the *`re`* module for string functions based on regular expressions.

`str.`**`capitalize`**`()`
返回原字符串的副本，其首个字符大写，其余为小写。

For 8-bit strings, this method is locale-dependent.

`str.`**`center`**`(`*width*`[,` *fillchar*`])`
Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is a space).

在 2.4 版更改: Support for the *fillchar* argument.

`str.`**`count`**`(`*sub*`[,` *start*`[,` *end*`]])`
反回子字符串 *sub* 在 [*start*, *end*] 范围内非重叠出现的次数。可选参数 *start* 与 *end* 会被解读为切片表示法。

`str.`**`decode`**`([`*encoding*`[,` *errors*`]])`
Decodes the string using the codec registered for *encoding*. *encoding* defaults to the default string encoding. *errors* may be given to set a different error handling scheme. The default is `'strict'`, meaning that encoding errors raise *`UnicodeError`*. Other possible values are `'ignore'`, `'replace'` and any other name registered via *`codecs.register_error()`*, see section 编解码器基类.

2.2 新版功能.

在 2.3 版更改: Support for other error handling schemes added.

在 2.7 版更改: 加入了对关键字参数的支持。

str.**encode**([*encoding*[, *errors*]])

Return an encoded version of the string. Default encoding is the current default string encoding. *errors* may be given to set a different error handling scheme. The default for *errors* is `'strict'`, meaning that encoding errors raise a *UnicodeError*. Other possible values are `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` and any other name registered via *codecs.register_error()*, see section 编解码器基类. For a list of possible encodings, see section 标准编码.

2.0 新版功能.

在 2.3 版更改: Support for `'xmlcharrefreplace'` and `'backslashreplace'` and other error handling schemes added.

在 2.7 版更改: 加入了对关键字参数的支持。

str.**endswith**(*suffix*[, *start*[, *end*]])

如果字符串以指定的 *suffix* 结束返回 `True`，否则返回 `False`。*suffix* 也可以为由多个供查找的后缀构成的元组。如果有可选项 *start*，将从所指定位置开始检查。如果有可选项 *end*，将在所指定位置停止比较。

在 2.5 版更改: Accept tuples as *suffix*.

str.**expandtabs**([*tabsize*])

返回字符串的副本，其中所有的制表符会由一个或多个空格替换，具体取决于当前列位置和给定的制表符宽度。每 *tabsize* 个字符设为一个制表位（默认值 8 时设定的制表位在列 0, 8, 16 依次类推）。要展开字符串，当前列将被设为零并逐一检查字符串中的每个字符。如果字符为制表符 (`\t`)，则会在结果中插入一个或多个空格符，直到当前列等于下一个制表位。（制表符本身不会被复制。）如果字符为换行符 (`\n`) 或回车符 (`\r`)，它会被复制并将当前列重设为零。任何其他字符会被不加修改地复制并将当前列加一，不论该字符在被打印时会如何显示。

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
```

str.**find**(*sub*[, *start*[, *end*]])

返回子字符串 *sub* 在 `s[start:end]` 切片内被找到的最小索引。可选参数 *start* 与 *end* 会被解读为切片表示法。如果 *sub* 未被找到则返回 `-1`。

注解: *find()* 方法应该只在你需要知道 *sub* 所在位置时使用。要检查 *sub* 是否为子字符串，请使用 `in` 操作符:

```
>>> 'Py' in 'Python'
True
```

---

str.**format**(**args*, ***kwargs*)

执行字符串格式化操作。调用此方法的字符串可以包含字符串字面值或者以花括号 `{}` 括起来的替换域。每个替换域可以包含一个位置参数的数字索引，或者一个关键字参数的名称。返回的字符串副本中每个替换域都会被替换为对应参数的字符串值。

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

请参阅格式字符串语法 了解有关可以在格式字符串中指定的各种格式选项的说明。

This method of string formatting is the new standard in Python 3, and should be preferred to the `%` formatting described in *String Formatting Operations* in new code.

2.6 新版功能.

str.**index**(*sub*[, *start*[, *end*]])
Like *find()*, but raise *ValueError* when the substring is not found.

str.**isalnum**()
Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

str.**isalpha**()
Return true if all characters in the string are alphabetic and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

str.**isdigit**()
Return true if all characters in the string are digits and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

str.**islower**()
Return true if all cased characters[4] in the string are lowercase and there is at least one cased character, false otherwise.

For 8-bit strings, this method is locale-dependent.

str.**isspace**()
Return true if there are only whitespace characters in the string and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

str.**istitle**()
Return true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return false otherwise.

For 8-bit strings, this method is locale-dependent.

str.**isupper**()
Return true if all cased characters[4] in the string are uppercase and there is at least one cased character, false otherwise.

For 8-bit strings, this method is locale-dependent.

str.**join**(*iterable*)
Return a string which is the concatenation of the strings in *iterable*. If there is any Unicode object in *iterable*, return a Unicode instead. A *TypeError* will be raised if there are any non-string or non Unicode object values in *iterable*. The separator between elements is the string providing this method.

str.**ljust**(*width*[, *fillchar*])
Return the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is a space). The original string is returned if *width* is less than or equal to `len(s)`.

在 2.4 版更改: Support for the *fillchar* argument.

str.**lower**()
返回原字符串的副本，其所有区分大小写的字符[4] 均转换为小写。

For 8-bit strings, this method is locale-dependent.

str.**lstrip**([*chars*])
Return a copy of the string with leading characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped:

[4] 区分大小写的字符是指所属一般类别属性为“Lu”(Letter, uppercase), “Ll”(Letter, lowercase) 或“Lt”(Letter, titlecase) 之一的字符。

```
>>> '   spacious   '.lstrip()
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

在 2.2.2 版更改: Support for the *chars* argument.

str.**partition**(*sep*)
在 *sep* 首次出现的位置拆分字符串，返回一个 3 元组，其中包含分隔符之前的部分、分隔符本身，以及分隔符之后的部分。如果分隔符未找到，则返回的 3 元组中包含字符本身以及两个空字符串。

2.5 新版功能.

str.**replace**(*old*, *new*[, *count*])
返回字符串的副本，其中出现的所有子字符串 *old* 都将被替换为 *new*。如果给出了可选参数 *count*，则只替换前 *count* 次出现。

str.**rfind**(*sub*[, *start*[, *end*]])
返回子字符串 *sub* 在字符串内被找到的最大（最右）索引，这样 *sub* 将包含在 `s[start:end]` 当中。可选参数 *start* 与 *end* 会被解读为切片表示法。如果未找到则返回 `-1`。

str.**rindex**(*sub*[, *start*[, *end*]])
类似于*rfind()*，但在子字符串 *sub* 未找到时会引发*ValueError*。

str.**rjust**(*width*[, *fillchar*])
Return the string right justified in a string of length *width*. Padding is done using the specified *fillchar* (default is a space). The original string is returned if *width* is less than or equal to `len(s)`.

在 2.4 版更改: Support for the *fillchar* argument.

str.**rpartition**(*sep*)
在 *sep* 最后一次出现的位置拆分字符串，返回一个 3 元组，其中包含分隔符之前的部分、分隔符本身，以及分隔符之后的部分。如果分隔符未找到，则返回的 3 元组中包含两个空字符串以及字符串本身。

2.5 新版功能.

str.**rsplit**([*sep*[, *maxsplit*]])
返回一个由字符串内单词组成的列表，使用 *sep* 作为分隔字符串。如果给出了 *maxsplit*，则最多进行 *maxsplit* 次拆分，从 *最右边*开始。如果 *sep* 未指定或为 `None`，任何空白字符串都会被作为分隔符。除了从右边开始拆分，*rsplit()* 的其他行为都类似于下文所述的*split()*。

2.4 新版功能.

str.**rstrip**([*chars*])
Return a copy of the string with trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

在 2.2.2 版更改: Support for the *chars* argument.

str.**split**([*sep*[, *maxsplit*]])
返回一个由字符串内单词组成的列表，使用 *sep* 作为分隔字符串。如果给出了 *maxsplit*，则最多进行 *maxsplit* 次拆分（因此，列表最多会有 `maxsplit+1` 个元素）。如果 *maxsplit* 未指定或为 `-1`，则不限制拆分次数（进行所有可能的拆分）。

如果给出了 *sep*，则连续的分隔符不会被组合在一起而是被视为分隔空字符串 (例如 `'1,,2'.split(',')` 将返回 `['1', '', '2']`)。*sep* 参数可能由多个字符组成 (例如 `'1<>2<>3'.split('<>')` 将返回 `['1', '2', '3']`)。使用指定的分隔符拆分空字符串将返回 `['']`。

如果 *sep* 未指定或为 `None`，则会应用另一种拆分算法：连续的空格会被视为单个分隔符，其结果将不包含开头或末尾的空字符串，如果字符串包含前缀或后缀空格的话。因此，使用 `None` 拆分空字符串或仅包含空格的字符串将返回 `[]`。

For example, `' 1 2 3 '.split()` returns `['1', '2', '3']`, and `' 1 2 3 '.split(None, 1)` returns `['1', '2 3 ']`.

`str.`**`splitlines`**`(`[*keepends*]`)`

Return a list of the lines in the string, breaking at line boundaries. This method uses the *universal newlines* approach to splitting lines. Line breaks are not included in the resulting list unless *keepends* is given and true.

Python recognizes `"\r"`, `"\n"`, and `"\r\n"` as line boundaries for 8-bit strings.

例如

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

不同于*split()*，当给出了分隔字符串 *sep* 时，对于空字符串此方法将返回一个空列表，而末尾的换行不会令结果中增加额外的行:

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

作为比较，`split('\n')` 的结果为:

```
>>> ''.split('\n')
['']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

`unicode.`**`splitlines`**`(`[*keepends*]`)`

Return a list of the lines in the string, like *str.splitlines()*. However, the Unicode method splits on the following line boundaries, which are a superset of the *universal newlines* recognized for 8-bit strings.

表示符	描述
`\n`	换行
`\r`	回车
`\r\n`	回车 + 换行
`\v` 或 `\x0b`	行制表符
`\f` 或 `\x0c`	换表单
`\x1c`	文件分隔符
`\x1d`	组分隔符
`\x1e`	记录分隔符
`\x85`	下一行 (C1 控制码)
`\u2028`	行分隔符
`\u2029`	段分隔符

在 2.7 版更改: `\v` 和 `\f` 被添加到行边界列表

str.**startswith**(*prefix*[, *start*[, *end*]])
如果字符串以指定的 *prefix* 开始则返回 `True`，否则返回 `False`。*prefix* 也可以为由多个供查找的前缀构成的元组。如果有可选项 *start*，将从所指定位置开始检查。如果有可选项 *end*，将在所指定位置停止比较。

在 2.5 版更改: Accept tuples as *prefix*.

str.**strip**([*chars*])
Return a copy of the string with the leading and trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

在 2.2.2 版更改: Support for the *chars* argument.

str.**swapcase**()
Return a copy of the string with uppercase characters converted to lowercase and vice versa.

For 8-bit strings, this method is locale-dependent.

str.**title**()
返回原字符串的标题版本，其中每个单词第一个字母为大写，其余字母为小写。

该算法使用一种简单的与语言无关的定义，将连续的字母组合视为单词。该定义在多数情况下都很有效，但它也意味着代表缩写形式与所有格的撇号也会成为单词边界，这可能导致不希望的结果:

```
>>> "they're bill's friends from the UK".title()
"They'Re Bill'S Friends From The Uk"
```

可以使用正则表达式来构建针对撇号的特别处理:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

For 8-bit strings, this method is locale-dependent.

str.**translate**(*table*[, *deletechars*])
Return a copy of the string where all characters occurring in the optional argument *deletechars* are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length 256.

You can use the *maketrans()* helper function in the *string* module to create a translation table. For string objects, set the *table* argument to `None` for translations that only delete characters:

```
>>> 'read this short text'.translate(None, 'aeiou')
'rd ths shrt txt'
```

2.6 新版功能: Support for a `None` *table* argument.

For Unicode objects, the *translate()* method does not accept the optional *deletechars* argument. Instead, it returns a copy of the *s* where all characters have been mapped through the given translation table which must be a mapping of Unicode ordinals to Unicode ordinals, Unicode strings or `None`. Unmapped characters are left untouched. Characters mapped to `None` are deleted. Note, a more flexible approach is to create a custom character mapping codec using the *codecs* module (see `encodings.cp1251` for an example).

`str.`**`upper`**`()`
返回原字符串的副本，其中所有区分大小写的字符[4] 均转换为大写。请注意如果 `s` 包含不区分大小写的字符或者如果结果字符的 Unicode 类别不是 “Lu” (Letter, uppercase) 而是 “Lt” (Letter, titlecase) 则 `s.upper().isupper()` 有可能为 `False`。

For 8-bit strings, this method is locale-dependent.

`str.`**`zfill`**`(`*width*`)`
Return the numeric string left filled with zeros in a string of length *width*. A sign prefix is handled correctly. The original string is returned if *width* is less than or equal to `len(s)`.

2.2.2 新版功能.

The following methods are present only on unicode objects:

`unicode.`**`isnumeric`**`()`
Return `True` if there are only numeric characters in S, `False` otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH.

`unicode.`**`isdecimal`**`()`
Return `True` if there are only decimal characters in S, `False` otherwise. Decimal characters include digit characters, and all characters that can be used to form decimal-radix numbers, e.g. U+0660, ARABIC-INDIC DIGIT ZERO.

5.6.2 String Formatting Operations

String and Unicode objects have one unique built-in operation: the `%` operator (modulo). This is also known as the string *formatting* or *interpolation* operator. Given `format % values` (where *format* is a string or Unicode object), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. The effect is similar to the using `sprintf()` in the C language. If *format* is a Unicode object, or if any of the objects being converted using the `%s` conversion are Unicode objects, the result will also be a Unicode object.

如果 *format* 要求一个单独参数，则 *values* 可以为一个非元组对象。[5] 否则的话，*values* 必须或者是一个包含项数与格式字符串中指定的转换符项数相同的元组，或者是一个单独映射对象（例如字典）。

转换标记符包含两个或更多字符并具有以下组成，且必须遵循此处规定的顺序：

1. `'%'` 字符，用于标记转换符的起始。
2. 映射键（可选），由加圆括号的字符序列组成 (例如 `(somename)`)。
3. 转换旗标（可选），用于影响某些转换类型的结果。
4. 最小字段宽度（可选）。如果指定为 `'*'` (星号)，则实际宽度会从 *values* 元组的下一元素中读取，要转换的对象则为最小字段宽度和可选的精度之后的元素。
5. Precision (optional), given as a `'.'` (dot) followed by the precision. If specified as `'*'` (an asterisk), the actual width is read from the next element of the tuple in *values*, and the value to convert comes after the precision.
6. 长度修饰符（可选）。
7. 转换类型。

[5] 要格式化单独一个元组，那么你应当提供一个单例元组，其唯一的元素就是要被格式化的元组。

当右边的参数为一个字典（或其他映射类型）时，字符串中的格式 必须包含加圆括号的映射键，对应 '%' 字符之后字典中的每一项。映射键将从映射中选取要格式化的值。例如：

```
>>> print '%(language)s has %(number)03d quote types.' % \
...       {"language": "Python", "number": 2}
Python has 002 quote types.
```

在此情况下格式中不能出现 * 标记符（因其需要一个序列类的参数列表）。

转换旗标为：

标志	含义
'#'	值的转换将使用“替代形式”（具体定义见下文）。
'0'	转换将为数字值填充零字符。
'-'	转换值将靠左对齐（如果同时给出 '0' 转换，则会覆盖后者）。
' '	(空格) 符号位转换产生的正数（或空字符串）前将留出一个空格。
'+'	符号字符 ('+' 或 '-') 将显示于转换结果的开头（会覆盖“空格”旗标）。

可以给出长度修饰符 (h, l 或 L)，但会被忽略，因为对 Python 来说没有必要–所以 %ld 等价于 %d。

转换类型为：

转换符	含义	注释
'd'	有符号十进制整数。	
'i'	有符号十进制整数。	
'o'	有符号八进制数。	(1)
'u'	过时类型–等价于 'd'。	(7)
'x'	有符号十六进制数（小写）。	(2)
'X'	有符号十六进制数（大写）。	(2)
'e'	浮点指数格式（小写）。	(3)
'E'	浮点指数格式（大写）。	(3)
'f'	浮点十进制格式。	(3)
'F'	浮点十进制格式。	(3)
'g'	浮点格式。如果指数小于 -4 或不小于精度则使用小写指数格式，否则使用十进制格式。	(4)
'G'	浮点格式。如果指数小于 -4 或不小于精度则使用大写指数格式，否则使用十进制格式。	(4)
'c'	单个字符（接受整数或单个字符的字符串）。	
'r'	String (converts any Python object using *repr()*).	(5)
's'	字符串（使用*str()* 转换任何 Python 对象）。	(6)
'%'	不转换参数，在结果中输出一个 '%' 字符。	

注释:

(1) The alternate form causes a leading zero ('0') to be inserted between left-hand padding and the formatting of the number if the leading character of the result is not already a zero.

(2) 此替代形式会在第一个数码之前插入 '0x' 或 '0X' 前缀（取决于是使用 'x' 还是 'X' 格式）。

(3) 此替代形式总是会在结果中包含一个小数点，即使其后并没有数码。

小数点后的数码位数由精度决定，默认为 6。

(4) 此替代形式总是会在结果中包含一个小数点，末尾各位的零不会如其他情况下那样被移除。

小数点前后的有效数码位数由精度决定，默认为 6。

(5) The %r conversion was added in Python 2.0.

The precision determines the maximal number of characters used.

(6) If the object or format provided is a *unicode* string, the resulting string will also be *unicode*.

The precision determines the maximal number of characters used.

(7) 参见 **PEP 237**。

由于 Python 字符串显式指明长度，`%s` 转换不会将 `'\0'` 视为字符串的结束。

在 2.7 版更改: 绝对值超过 1e50 的 `%f` 转换不会再被替换为 `%g` 转换。

Additional string operations are defined in standard modules *string* and *re*.

5.6.3 XRange Type

The *xrange* type is an immutable sequence which is commonly used for looping. The advantage of the *xrange* type is that an *xrange* object will always take the same amount of memory, no matter the size of the range it represents. There are no consistent performance advantages.

XRange objects have very little behavior: they only support indexing, iteration, and the *len()* function.

5.6.4 可变序列类型

List and *bytearray* objects support additional operations that allow in-place modification of the object. Other mutable sequence types (when added to the language) should also support these operations. Strings and tuples are immutable sequence types: such objects cannot be modified once created. The following operations are defined on mutable sequence types (where *x* is an arbitrary object):

运算	结果:	注释
`s[i] = x`	将 *s* 的第 *i* 项替换为 *x*	
`s[i:j] = t`	将 *s* 从 *i* 到 *j* 的切片替换为可迭代对象 *t* 的内容	
`del s[i:j]`	等同于 `s[i:j] = []`	
`s[i:j:k] = t`	将 `s[i:j:k]` 的元素替换为 *t* 的元素	(1)
`del s[i:j:k]`	从列表中移除 `s[i:j:k]` 的元素	
`s.append(x)`	same as `s[len(s):len(s)] = [x]`	(2)
`s.extend(t)` 或 `s += t`	for the most part the same as `s[len(s):len(s)] = t`	(3)
`s *= n`	使用 *s* 的内容重复 *n* 次来对其进行更新	(11)
`s.count(x)`	return number of *i*'s for which `s[i] == x`	
`s.index(x[, i[, j]])`	return smallest *k* such that `s[k] == x` and `i <= k < j`	(4)
`s.insert(i, x)`	same as `s[i:i] = [x]`	(5)
`s.pop([i])`	same as `x = s[i]; del s[i]; return x`	(6)
`s.remove(x)`	same as `del s[s.index(x)]`	(4)
`s.reverse()`	就地将列表中的元素逆序。	(7)
`s.sort([cmp[, key[, reverse]]])`	sort the items of *s* in place	(7)(8)(9)(10)

注释:

(1) *t* must have the same length as the slice it is replacing.

(2) The C implementation of Python has historically accepted multiple parameters and implicitly joined them into a tuple; this no longer works in Python 2.0. Use of this misfeature has been deprecated since Python 1.4.

(3) *t* can be any iterable object.

(4) Raises *ValueError* when *x* is not found in *s*. When a negative index is passed as the second or third parameter to the `index()` method, the list length is added, as for slice indices. If it is still negative, it is truncated to zero, as for slice indices.

在 2.3 版更改: Previously, `index()` didn't have arguments for specifying start and stop positions.

(5) When a negative index is passed as the first parameter to the `insert()` method, the list length is added, as for slice indices. If it is still negative, it is truncated to zero, as for slice indices.

在 2.3 版更改: Previously, all negative indices were truncated to zero.

(6) The `pop()` method's optional argument *i* defaults to `-1`, so that by default the last item is removed and returned.

(7) The `sort()` and `reverse()` methods modify the list in place for economy of space when sorting or reversing a large list. To remind you that they operate by side effect, they don't return the sorted or reversed list.

(8) The `sort()` method takes optional arguments for controlling the comparisons.

cmp specifies a custom comparison function of two arguments (list items) which should return a negative, zero or positive number depending on whether the first argument is considered smaller than, equal to, or larger than the second argument: `cmp=lambda x,y: cmp(x.lower(), y.lower())`. The default value is `None`.

key specifies a function of one argument that is used to extract a comparison key from each list element: `key=str.lower`. The default value is `None`.

reverse 为一个布尔值。如果设为 `True`，则每个列表元素将按反向顺序比较进行排序。

In general, the *key* and *reverse* conversion processes are much faster than specifying an equivalent *cmp* function. This is because *cmp* is called multiple times for each list element while *key* and *reverse* touch each element only once. Use *functools.cmp_to_key()* to convert an old-style *cmp* function to a *key* function.

在 2.3 版更改: Support for `None` as an equivalent to omitting *cmp* was added.

在 2.4 版更改: Support for *key* and *reverse* was added.

(9) Starting with Python 2.3, the `sort()` method is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal —this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

(10) **CPython implementation detail:** While a list is being sorted, the effect of attempting to mutate, or even inspect, the list is undefined. The C implementation of Python 2.3 and newer makes the list appear empty for the duration, and raises *ValueError* if it can detect that the list has been mutated during a sort.

(11) The value *n* is an integer, or an object implementing `__index__()`. Zero and negative values of *n* clear the sequence. Items in the sequence are not copied; they are referenced multiple times, as explained for `s * n` under *Sequence Types —str, unicode, list, tuple, bytearray, buffer, xrange*.

5.7 集合类型—set, frozenset

A *set* object is an unordered collection of distinct *hashable* objects. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference. (For other containers see the built in *dict*, `list`, and *tuple* classes, and the *collections* module.)

2.4 新版功能.

与其他多项集一样，集合也支持 `x in set`, `len(set)` 和 `for x in set`。作为一种无序的多项集，集合并不记录元素位置或插入顺序。相应地，集合不支持索引、切片或其他序列类的操作。

目前有两种内置集合类型，*set* 和*frozenset*。*set* 类型是可变的—其内容可以使用 `add()` 和 `remove()` 这样的方法来改变。由于是可变类型，它没有哈希值，且不能被用作字典的键或其他集合的元素。*frozenset*

类型是不可变并且为*hashable* —其内容在被创建后不能再改变；因此它可以被用作字典的键或其他集合的元素。

As of Python 2.7, non-empty sets (not frozensets) can be created by placing a comma-separated list of elements within braces, for example: `{'jack', 'sjoerd'}`, in addition to the *set* constructor.

两个类的构造器具有相同的作用方式：

class set([*iterable*])
class frozenset([*iterable*])
返回一个新的 set 或 frozenset 对象，其元素来自于 *iterable*。集合的元素必须为*hashable*。要表示由集合对象构成的集合，所有的内层集合必须为*frozenset* 对象。如果未指定 *iterable*，则将返回一个新的空集合。

set 和*frozenset* 的实例提供以下操作：

len(s)
返回集合 *s* 中的元素数量（即 *s* 的基数）。

x in s
检测 *x* 是否为 *s* 中的成员。

x not in s
检测 *x* 是否非 *s* 中的成员。

isdisjoint(*other*)
如果集合中没有与 *other* 共有的元素则返回 `True`。当且仅当两个集合的交集为空集合时，两者为不相交集合。

2.6 新版功能.

issubset(*other*)
set <= other
检测是否集合中的每个元素都在 *other* 之中。

set < other
检测集合是否为 *other* 的真子集，即 `set <= other and set != other`。

issuperset(*other*)
set >= other
检测是否 *other* 中的每个元素都在集合之中。

set > other
检测集合是否为 *other* 的真超集，即 `set >= other and set != other`。

union(*others)
set | other | ...
返回一个新集合，其中包含来自原集合以及 others 指定的所有集合中的元素。

在 2.6 版更改: Accepts multiple input iterables.

intersection(*others)
set & other & ...
返回一个新集合，其中包含原集合以及 others 指定的所有集合中共有的元素。

在 2.6 版更改: Accepts multiple input iterables.

difference(*others)
set - other - ...
返回一个新集合，其中包含原集合中在 others 指定的其他集合中不存在的元素。

在 2.6 版更改: Accepts multiple input iterables.

symmetric_difference(*other*)

set ^ other
返回一个新集合，其中的元素或属于原集合或属于 *other* 指定的其他集合，但不能同时属于两者。

copy()
返回原集合的浅拷贝。

请注意，非运算符版本的*union()*，*intersection()*，*difference()*，以及*symmetric_difference()*，*issubset()* 和*issuperset()* 方法会接受任意可迭代对象作为参数。相比之下，它们所对应的运算符版本则要求其参数为集合。这就排除了容易出错的构造形式例如 `set('abc') & 'cbs'`，而推荐可读性更强的 `set('abc').intersection('cbs')`。

set 和*frozenset* 均支持集合与集合的比较。两个集合当且仅当每个集合中的每个元素均包含于另一个集合之内（即各为对方的子集）时则相等。一个集合当且仅当其为另一个集合的真子集（即为后者的子集但两者不相等）时则小于另一个集合。一个集合当且仅当其为另一个集合的真超集（即为后者的超集但两者不相等）时则大于另一个集合。

set 的实例与*frozenset* 的实例之间基于它们的成员进行比较。例如 `set('abc') == frozenset('abc')` 返回 `True`，`set('abc') in set([frozenset('abc')])` 也一样。

The subset and equality comparisons do not generalize to a total ordering function. For example, any two non-empty disjoint sets are not equal and are not subsets of each other, so *all* of the following return `False`: `a<b`, `a==b`, or `a>b`. Accordingly, sets do not implement the `__cmp__()` method.

由于集合仅定义了部分排序（子集关系），因此由集合构成的列表 `list.sort()` 方法的输出并无定义。

集合的元素，与字典的键类似，必须为*hashable*。

混合了*set* 实例与*frozenset* 的二进制位运算将返回与第一个操作数相同的类型。例如: `frozenset('ab') | set('bc')` 将返回*frozenset* 的实例。

下表列出了可用于*set* 而不能用于不可变的*frozenset* 实例的操作:

update(*others*)
set |= other | ...
更新集合，添加来自 others 中的所有元素。

在 2.6 版更改: Accepts multiple input iterables.

intersection_update(*others*)
set &= other & ...
更新集合，只保留其中在所有 others 中也存在的元素。

在 2.6 版更改: Accepts multiple input iterables.

difference_update(*others*)
set -= other | ...
更新集合，移除其中也存在于 others 中的元素。

在 2.6 版更改: Accepts multiple input iterables.

symmetric_difference_update(*other*)
set ^= other
更新集合，只保留存在于集合的一方而非共同存在的元素。

add(*elem*)
将元素 *elem* 添加到集合中。

remove(*elem*)
从集合中移除元素 *elem*。如果 *elem* 不存在于集合中则会引发*KeyError*。

discard(*elem*)
如果元素 *elem* 存在于集合中则将其移除。

pop()
从集合中移除并返回任意一个元素。如果集合为空则会引发*KeyError*。

clear()
从集合中移除所有元素。

请注意，非运算符版本的*update()*，*intersection_update()*，*difference_update()*和*symmetric_difference_update()*方法将接受任意可迭代对象作为参数。

请注意，__contains__(), *remove()* 和*discard()* 方法的 *elem* 参数可能是一个 set。为支持对一个等价的 frozenset 进行搜索，会根据 *elem* 临时创建一个该类型对象。

参见:

Comparison to the built-in set types Differences between the *sets* module and the built-in set types.

5.8 映射类型—dict

A *mapping* object maps *hashable* values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the *dictionary*. (For other containers see the built in `list`, *set*, and *tuple* classes, and the *collections* module.)

字典的键几乎可以是任何值。非*hashable* 的值，即包含列表、字典或其他可变类型的值（此类对象基于值而非对象标识进行比较）不可用作键。数字类型用作键时遵循数字比较的一般规则：如果两个数值相等 (例如 `1` 和 `1.0`) 则两者可以被用来索引同一字典条目。（但是请注意，由于计算机对于浮点数存储的只是近似值，因此将其用作字典键是不明智的。）

字典可以通过将以逗号分隔的 键：值对列表包含于花括号之内来创建，例如：`{'jack': 4098, 'sjoerd': 4127}` 或 `{4098: 'jack', 4127: 'sjoerd'}`，也可以通过*dict* 构造器来创建。

class dict(***kwarg*)
class dict(*mapping*, ***kwarg*)
class dict(*iterable*, ***kwarg*)
返回一个新的字典，基于可选的位置参数和可能为空的关键字参数集来初始化。

如果没有给出位置参数，将创建一个空字典。如果给出一个位置参数并且其属于映射对象，将创建一个具有与映射对象相同键值对的字典。否则的话，位置参数必须为一个*iterable* 对象。该可迭代对象中的每一项本身必须为一个刚好包含两个元素的可迭代对象。每一项中的第一个对象将成为新字典的一个键，第二个对象将成为其对应的值。如果一个键出现一次以上，该键的最后一个值将成为其在新字典中对应的值。

如果给出了关键字参数，则关键字参数及其值会被加入到基于位置参数创建的字典。如果要加入的键已存在，来自关键字参数的值将替代来自位置参数的值。

作为演示，以下示例返回的字典均等于 `{"one": 1, "two": 2, "three": 3}`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

像第一个例子那样提供关键字参数的方式只能使用有效的 Python 标识符作为键。其他方式则可使用任何有效的键。

2.2 新版功能.

在 2.3 版更改: Support for building a dictionary from keyword arguments added.

这些是字典所支持的操作（因而自定义的映射类型也应当支持）:

len(d)
返回字典 *d* 中的项数。

d[key]
返回 *d* 中以 *key* 为键的项。如果映射中不存在 *key* 则会引发*KeyError*。

如果字典的子类定义了方法 __missing__() 并且 *key* 不存在，则 d[key] 操作将调用该方法并附带键 *key* 作为参数。d[key] 随后将返回或引发 __missing__(key) 调用所返回或引发的任何对象或异常。没有其他操作或方法会发起调用 __missing__()。如果未定义 __missing__()，则会引发*KeyError*。__missing__() 必须是一个方法；它不能是一个实例变量:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

上面的例子显示了*collections.Counter* 实现的部分代码。还有另一个不同的 __missing__ 方法是由*collections.defaultdict* 所使用的。

2.5 新版功能: Recognition of __missing__ methods of dict subclasses.

d[key] = value
将 d[key] 设为 *value*。

del d[key]
将 d[key] 从 *d* 中移除。如果映射中不存在 *key* 则会引发*KeyError*。

key in d
如果 *d* 中存在键 *key* 则返回 True，否则返回 False。

2.2 新版功能.

key not in d
等价于 not key in d。

2.2 新版功能.

iter(d)
Return an iterator over the keys of the dictionary. This is a shortcut for *iterkeys()*.

clear()
移除字典中的所有元素。

copy()
返回原字典的浅拷贝。

fromkeys(*seq*[, *value*])
Create a new dictionary with keys from *seq* and values set to *value*.

fromkeys() is a class method that returns a new dictionary. *value* defaults to None.

2.3 新版功能.

get(*key*[, *default*])
如果 *key* 存在于字典中则返回 *key* 的值，否则返回 *default*。如果 *default* 未给出则默认为 None，因而此方法绝不会引发*KeyError*。

has_key(*key*)
Test for the presence of *key* in the dictionary. *has_key()* is deprecated in favor of `key in d`.

items()
Return a copy of the dictionary's list of `(key, value)` pairs.

CPython implementation detail: Keys and values are listed in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions.

If *items()*, *keys()*, *values()*, *iteritems()*, *iterkeys()*, and *itervalues()* are called with no intervening modifications to the dictionary, the lists will directly correspond. This allows the creation of `(value, key)` pairs using *zip()*: `pairs = zip(d.values(), d.keys())`. The same relationship holds for the *iterkeys()* and *itervalues()* methods: `pairs = zip(d.itervalues(), d.iterkeys())` provides the same value for `pairs`. Another way to create the same list is `pairs = [(v, k) for (k, v) in d.iteritems()]`.

iteritems()
Return an iterator over the dictionary's `(key, value)` pairs. See the note for *dict.items()*.

Using *iteritems()* while adding or deleting entries in the dictionary may raise a *RuntimeError* or fail to iterate over all entries.

2.2 新版功能.

iterkeys()
Return an iterator over the dictionary's keys. See the note for *dict.items()*.

Using *iterkeys()* while adding or deleting entries in the dictionary may raise a *RuntimeError* or fail to iterate over all entries.

2.2 新版功能.

itervalues()
Return an iterator over the dictionary's values. See the note for *dict.items()*.

Using *itervalues()* while adding or deleting entries in the dictionary may raise a *RuntimeError* or fail to iterate over all entries.

2.2 新版功能.

keys()
Return a copy of the dictionary's list of keys. See the note for *dict.items()*.

pop(*key*[, *default*])
如果 *key* 存在于字典中则将其移除并返回其值，否则返回 *default*。如果 *default* 未给出且 *key* 不存在于字典中，则会引发*KeyError*。

2.3 新版功能.

popitem()
Remove and return an arbitrary `(key, value)` pair from the dictionary.

popitem() is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling *popitem()* raises a *KeyError*.

setdefault(*key*[, *default*])
如果字典存在键 *key*，返回它的值。如果不存在，插入值为 *default* 的键 *key*，并返回 *default*。*default* 默认为 None。

update([*other*])
使用来自 *other* 的键/值对更新字典，覆盖原有的键。返回 None。

update() accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: `d.update(red=1, blue=2)`.

在 2.4 版更改: Allowed the argument to be an iterable of key/value pairs and allowed keyword arguments.

values()
Return a copy of the dictionary's list of values. See the note for *dict.items()*.

viewitems()
Return a new view of the dictionary's items (`(key, value)` pairs). See below for documentation of view objects.

2.7 新版功能.

viewkeys()
Return a new view of the dictionary's keys. See below for documentation of view objects.

2.7 新版功能.

viewvalues()
Return a new view of the dictionary's values. See below for documentation of view objects.

2.7 新版功能.

Dictionaries compare equal if and only if they have the same `(key, value)` pairs.

5.8.1 字典视图对象

The objects returned by *dict.viewkeys()*, *dict.viewvalues()* and *dict.viewitems()* are *view objects*. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

字典视图可以被迭代以产生与其对应的数据，并支持成员检测：

len(dictview)
返回字典中的条目数。

iter(dictview)
返回字典中的键、值或项（以（键，值）为元素的元组表示）的迭代器。

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond. This allows the creation of `(value, key)` pairs using *zip()*: `pairs = zip(d.values(), d.keys())`. Another way to create the same list is `pairs = [(v, k) for (k, v) in d.items()]`.

在添加或删除字典中的条目期间对视图进行迭代可能引发*RuntimeError* 或者无法完全迭代所有条目。

x in dictview
如果 *x* 是对应字典中存在的键、值或项（在最后一种情况下 *x* 应为一个（键，值）元组）则返回 True。

Keys views are set-like since their entries are unique and hashable. If all values are hashable, so that (key, value) pairs are unique and hashable, then the items view is also set-like. (Values views are not treated as set-like since the entries are generally not unique.) Then these set operations are available ("other" refers either to another view or a set):

dictview & other
Return the intersection of the dictview and the other object as a new set.

dictview | other
Return the union of the dictview and the other object as a new set.

dictview - other
Return the difference between the dictview and the other object (all elements in *dictview* that aren't in *other*) as a new set.

dictview ^ other
Return the symmetric difference (all elements either in *dictview* or *other*, but not in both) of the dictview and the other object as a new set.

一个使用字典视图的示例:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.viewkeys()
>>> values = dishes.viewvalues()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order
>>> list(keys)
['eggs', 'bacon', 'sausage', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['spam', 'bacon']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
```

5.9 File Objects

File objects are implemented using C's `stdio` package and can be created with the built-in *`open()`* function. File objects are also returned by some other built-in functions and methods, such as *`os.popen()`* and *`os.fdopen()`* and the `makefile()` method of socket objects. Temporary files can be created using the *`tempfile`* module, and high-level file operations such as copying, moving, and deleting files and directories can be achieved with the *`shutil`* module.

When a file operation fails for an I/O-related reason, the exception *`IOError`* is raised. This includes situations where the operation is not defined for some reason, like `seek()` on a tty device or writing a file opened for reading.

Files have the following methods:

`file.`**`close`**`()`
Close the file. A closed file cannot be read or written any more. Any operation which requires that the file be open will raise a *`ValueError`* after the file has been closed. Calling *`close()`* more than once is allowed.

As of Python 2.5, you can avoid having to call this method explicitly if you use the `with` statement. For example, the following code will automatically close *f* when the `with` block is exited:

```
from __future__ import with_statement # This isn't required in Python 2.6

with open("hello.txt") as f:
    for line in f:
        print line,
```

In older versions of Python, you would have needed to do this to get the same effect:

```
f = open("hello.txt")
try:
    for line in f:
        print line,
finally:
    f.close()
```

注解： Not all "file-like" types in Python support use as a context manager for the `with` statement. If your code is intended to work with any file-like object, you can use the function *contextlib.closing()* instead of using the object directly.

`file.`**`flush`**`()`
Flush the internal buffer, like `stdio`'s `fflush()`. This may be a no-op on some file-like objects.

注解： *flush()* does not necessarily write the file's data to disk. Use *flush()* followed by *os.fsync()* to ensure this behavior.

`file.`**`fileno`**`()`
Return the integer "file descriptor" that is used by the underlying implementation to request I/O operations from the operating system. This can be useful for other, lower level interfaces that use file descriptors, such as the *fcntl* module or *os.read()* and friends.

注解： File-like objects which do not have a real file descriptor should *not* provide this method!

`file.`**`isatty`**`()`
Return `True` if the file is connected to a tty(-like) device, else `False`.

注解： If a file-like object is not associated with a real file, this method should *not* be implemented.

`file.`**`next`**`()`
A file object is its own iterator, for example `iter(f)` returns *f* (unless *f* is closed). When a file is used as an iterator, typically in a `for` loop (for example, `for line in f: print line.strip()`), the *next()* method is called repeatedly. This method returns the next input line, or raises *StopIteration* when EOF is hit when the file is open for reading (behavior is undefined when the file is open for writing). In order to make a `for` loop the most efficient way of looping over the lines of a file (a very common operation), the *next()* method uses a hidden read-ahead buffer. As a consequence of using a read-ahead buffer, combining *next()* with other file methods (like *readline()*) does not work right. However, using *seek()* to reposition the file to an absolute position will flush the read-ahead buffer.

2.3 新版功能.

file.**read**([*size*])
Read at most *size* bytes from the file (less if the read hits EOF before obtaining *size* bytes). If the *size* argument is negative or omitted, read all data until EOF is reached. The bytes are returned as a string object. An empty string is returned when EOF is encountered immediately. (For certain files, like ttys, it makes sense to continue reading after an EOF is hit.) Note that this method may call the underlying C function `fread()` more than once in an effort to acquire as close to *size* bytes as possible. Also note that when in non-blocking mode, less data than was requested may be returned, even if no *size* parameter was given.

注解: This function is simply a wrapper for the underlying `fread()` C function, and will behave the same in corner cases, such as whether the EOF value is cached.

---

file.**readline**([*size*])
Read one entire line from the file. A trailing newline character is kept in the string (but may be absent when a file ends with an incomplete line).[6] If the *size* argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned. When *size* is not 0, an empty string is returned *only* when EOF is encountered immediately.

注解: Unlike `stdio`'s `fgets()`, the returned string contains null characters (`'\0'`) if they occurred in the input.

---

file.**readlines**([*sizehint*])
Read until EOF using *readline()* and return a list containing the lines thus read. If the optional *sizehint* argument is present, instead of reading up to EOF, whole lines totalling approximately *sizehint* bytes (possibly after rounding up to an internal buffer size) are read. Objects implementing a file-like interface may choose to ignore *sizehint* if it cannot be implemented, or cannot be implemented efficiently.

file.**xreadlines**()
This method returns the same thing as `iter(f)`.

2.1 新版功能.

2.3 版后已移除: Use `for line in file` instead.

file.**seek**(*offset*[, *whence*])
Set the file's current position, like `stdio`'s `fseek()`. The *whence* argument is optional and defaults to `os.SEEK_SET` or `0` (absolute file positioning); other values are `os.SEEK_CUR` or `1` (seek relative to the current position) and `os.SEEK_END` or `2` (seek relative to the file's end). There is no return value.

For example, `f.seek(2, os.SEEK_CUR)` advances the position by two and `f.seek(-3, os.SEEK_END)` sets the position to the third to last.

Note that if the file is opened for appending (mode `'a'` or `'a+'`), any *seek()* operations will be undone at the next write. If the file is only opened for writing in append mode (mode `'a'`), this method is essentially a no-op, but it remains useful for files opened in append mode with reading enabled (mode `'a+'`). If the file is opened in text mode (without `'b'`), only offsets returned by *tell()* are legal. Use of other offsets causes undefined behavior.

Note that not all file objects are seekable.

在 2.6 版更改: Passing float values as offset has been deprecated.

file.**tell**()
Return the file's current position, like `stdio`'s `ftell()`.

[6] The advantage of leaving the newline on is that returning an empty string is then an unambiguous EOF indication. It is also possible (in cases where it might matter, for example, if you want to make an exact copy of a file while scanning its lines) to tell whether the last line of a file ended in a newline or not (yes this happens!).

注解: On Windows, *tell()* can return illegal values (after an `fgets()`) when reading files with Unix-style line-endings. Use binary mode (`'rb'`) to circumvent this problem.

`file.`**`truncate`**`([`*size*`])`
Truncate the file's size. If the optional *size* argument is present, the file is truncated to (at most) that size. The size defaults to the current position. The current file position is not changed. Note that if a specified size exceeds the file's current size, the result is platform-dependent: possibilities include that the file may remain unchanged, increase to the specified size as if zero-filled, or increase to the specified size with undefined new content. Availability: Windows, many Unix variants.

`file.`**`write`**`(`*str*`)`
Write a string to the file. There is no return value. Due to buffering, the string may not actually show up in the file until the *flush()* or *close()* method is called.

`file.`**`writelines`**`(`*sequence*`)`
Write a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value. (The name is intended to match *readlines()*; *writelines()* does not add line separators.)

Files support the iterator protocol. Each iteration returns the same result as *readline()*, and iteration ends when the *readline()* method returns an empty string.

File objects also offer a number of other interesting attributes. These are not required for file-like objects, but should be implemented if they make sense for the particular object.

`file.`**`closed`**
bool indicating the current state of the file object. This is a read-only attribute; the *close()* method changes the value. It may not be available on all file-like objects.

`file.`**`encoding`**
The encoding that this file uses. When Unicode strings are written to a file, they will be converted to byte strings using this encoding. In addition, when the file is connected to a terminal, the attribute gives the encoding that the terminal is likely to use (that information might be incorrect if the user has misconfigured the terminal). The attribute is read-only and may not be present on all file-like objects. It may also be `None`, in which case the file uses the system default encoding for converting Unicode strings.

2.3 新版功能.

`file.`**`errors`**
The Unicode error handler used along with the encoding.

2.6 新版功能.

`file.`**`mode`**
The I/O mode for the file. If the file was created using the *open()* built-in function, this will be the value of the *mode* parameter. This is a read-only attribute and may not be present on all file-like objects.

`file.`**`name`**
If the file object was created using *open()*, the name of the file. Otherwise, some string that indicates the source of the file object, of the form `<...>`. This is a read-only attribute and may not be present on all file-like objects.

`file.`**`newlines`**
If Python was built with *universal newlines* enabled (the default) this read-only attribute exists, and for files opened in universal newline read mode it keeps track of the types of newlines encountered while reading the file. The values it can take are `'\r'`, `'\n'`, `'\r\n'`, `None` (unknown, no newlines read yet) or a tuple containing all the newline types seen, to indicate that multiple newline conventions were encountered. For files not opened in universal newlines read mode the value of this attribute will be `None`.

file.**softspace**
Boolean that indicates whether a space character needs to be printed before another value when using the `print` statement. Classes that are trying to simulate a file object should also have a writable *softspace* attribute, which should be initialized to zero. This will be automatic for most classes implemented in Python (care may be needed for objects that override attribute access); types implemented in C will have to provide a writable *softspace* attribute.

注解: This attribute is not used to control the `print` statement, but to allow the implementation of `print` to keep track of its internal state.

5.10 memoryview type

2.7 新版功能.

memoryview objects allow Python code to access the internal data of an object that supports the buffer protocol without copying. Memory is generally interpreted as simple bytes.

class memoryview(*obj*)
Create a *memoryview* that references *obj*. *obj* must support the buffer protocol. Built-in objects that support the buffer protocol include *str* and *bytearray* (but not *unicode*).

A *memoryview* has the notion of an *element*, which is the atomic memory unit handled by the originating object *obj*. For many simple types such as *str* and *bytearray*, an element is a single byte, but other third-party types may expose larger elements.

`len(view)` returns the total number of elements in the memoryview, *view*. The *itemsize* attribute will give you the number of bytes in a single element.

A *memoryview* supports slicing to expose its data. Taking a single index will return a single element as a *str* object. Full slicing will result in a subview:

```
>>> v = memoryview('abcefg')
>>> v[1]
'b'
>>> v[-1]
'g'
>>> v[1:4]
<memory at 0x77ab28>
>>> v[1:4].tobytes()
'bce'
```

If the object the memoryview is over supports changing its data, the memoryview supports slice assignment:

```
>>> data = bytearray('abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = 'z'
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = '123'
>>> data
bytearray(b'z123fg')
>>> v[2] = 'spam'
```

(下页继续)

(续上页)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot modify size of memoryview object
```

Notice how the size of the memoryview object cannot be changed.

memoryview has two methods:

tobytes()
Return the data in the buffer as a bytestring (an object of class *str*).

```
>>> m = memoryview("abc")
>>> m.tobytes()
'abc'
```

tolist()
Return the data in the buffer as a list of integers.

```
>>> memoryview("abc").tolist()
[97, 98, 99]
```

还存在一些可用的只读属性：

format
A string containing the format (in *struct* module style) for each element in the view. This defaults to `'B'`, a simple bytestring.

itemsize
The size in bytes of each element of the memoryview.

shape
一个整数元组，通过*ndim* 的长度值给出内存所代表的 N 维数组的形状。

ndim
一个整数，表示内存所代表的多维数组具有多少个维度。

strides
一个整数元组，通过*ndim* 的长度给出以字节表示的大小，以便访问数组中每个维度上的每个元素。

readonly
一个表明内存是否只读的布尔值。

5.11 上下文管理器类型

2.5 新版功能.

Python's `with` statement supports the concept of a runtime context defined by a context manager. This is implemented using two separate methods that allow user-defined classes to define a runtime context that is entered before the statement body is executed and exited when the statement ends.

The *context management protocol* consists of a pair of methods that need to be provided for a context manager object to define a runtime context:

contextmanager.**__enter__**()
Enter the runtime context and return either this object or another object related to the runtime context. The value returned by this method is bound to the identifier in the `as` clause of `with` statements using this context manager.

An example of a context manager that returns itself is a file object. File objects return themselves from __enter__() to allow *open()* to be used as the context expression in a `with` statement.

An example of a context manager that returns a related object is the one returned by *decimal.localcontext()*. These managers set the active decimal context to a copy of the original decimal context and then return the copy. This allows changes to be made to the current decimal context in the body of the `with` statement without affecting code outside the `with` statement.

`contextmanager.__exit__`(*exc_type*, *exc_val*, *exc_tb*)
退出运行时上下文并返回一个布尔值旗标来表明所发生的任何异常是否应当被屏蔽。如果在执行 `with` 语句的语句体期间发生了异常，则参数会包含异常的类型、值以及回溯信息。在其他情况下三个参数均为 `None`。

Returning a true value from this method will cause the `with` statement to suppress the exception and continue execution with the statement immediately following the `with` statement. Otherwise the exception continues propagating after this method has finished executing. Exceptions that occur during execution of this method will replace any exception that occurred in the body of the `with` statement.

The exception passed in should never be reraised explicitly - instead, this method should return a false value to indicate that the method completed successfully and does not want to suppress the raised exception. This allows context management code (such as `contextlib.nested`) to easily detect whether or not an *__exit__()* method has actually failed.

Python 定义了一些上下文管理器来支持简易的线程同步、文件或其他对象的快速关闭，以及更方便地操作活动的十进制算术上下文。除了实现上下文管理协议以外，不同类型不会被特殊处理。请参阅*contextlib* 模块查看相关的示例。

Python's *generator*s and the `contextlib.contextmanager` *decorator* provide a convenient way to implement these protocols. If a generator function is decorated with the `contextlib.contextmanager` decorator, it will return a context manager implementing the necessary `__enter__()` and `__exit__()` methods, rather than the iterator produced by an undecorated generator function.

请注意，Python/C API 中 Python 对象的类型结构中并没有针对这些方法的专门槽位。想要定义这些方法的扩展类型必须将它们作为普通的 Python 可访问方法来提供。与设置运行时上下文的开销相比，单个类字典查找的开销可以忽略不计。

5.12 其他内置类型

解释器支持一些其他种类的对象。这些对象大都仅支持一两种操作。

5.12.1 模块

模块唯一的特殊操作是属性访问: `m.name`，这里 *m* 为一个模块而 *name* 访问定义在 *m* 的符号表中的一个名称。模块属性可以被赋值。(请注意 `import` 语句严格来说也是对模块对象的一种操作；`import foo` 不要求存在一个名为 *foo* 的模块对象，而是要求存在一个对于名为 *foo* 的模块的 (永久性) 定义。)

每个模块都有一个特殊属性*__dict__*。这是包含模块的符号表的字典。修改此字典将实际改变模块的符号表，但是无法直接对*__dict__* 赋值 (你可以写 `m.__dict__['a'] = 1`，这会将 `m.a` 定义为 `1`，但是你不能写 `m.__dict__ = {}`)。不建议直接修改*__dict__*。

内置于解释器中的模块会写成这样: `<module 'sys' (built-in)>`。如果是从一个文件加载，则会写成 `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`。

5.12.2 类与类实例

关于这些类型请参阅 objects 和 class。

5.12.3 函数

函数对象是通过函数定义创建的。对函数对象的唯一操作是调用它: `func(argument-list)`。

实际上存在两种不同的函数对象：内置函数和用户自定义函数。两者支持同样的操作（调用函数），但实现方式不同，因此对象类型也不同。

更多信息请参阅 function。

5.12.4 方法

方法是使用属性表示法来调用的函数。存在两种形式：内置方法（例如列表的 `append()` 方法）和类实例方法。内置方法由支持它们的类型来描述。

The implementation adds two special read-only attributes to class instance methods: `m.im_self` is the object on which the method operates, and `m.im_func` is the function implementing the method. Calling `m(arg-1, arg-2, ..., arg-n)` is completely equivalent to calling `m.im_func(m.im_self, arg-1, arg-2, ..., arg-n)`.

Class instance methods are either *bound* or *unbound*, referring to whether the method was accessed through an instance or a class, respectively. When a method is unbound, its `im_self` attribute will be `None` and if called, an explicit `self` object must be passed as the first argument. In this case, `self` must be an instance of the unbound method's class (or a subclass of that class), otherwise a *`TypeError`* is raised.

Like function objects, methods objects support getting arbitrary attributes. However, since method attributes are actually stored on the underlying function object (`meth.im_func`), setting method attributes on either bound or unbound methods is disallowed. Attempting to set an attribute on a method results in an *`AttributeError`* being raised. In order to set a method attribute, you need to explicitly set it on the underlying function object:

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'instancemethod' object has no attribute 'whoami'
>>> c.method.im_func.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

更多信息请参阅 types。

5.12.5 代码对象

Code objects are used by the implementation to represent "pseudo-compiled" executable Python code such as a function body. They differ from function objects because they don't contain a reference to their global execution environment. Code objects are returned by the built-in *compile()* function and can be extracted from function objects through their `func_code` attribute. See also the *code* module.

A code object can be executed or evaluated by passing it (instead of a source string) to the `exec` statement or the built-in *eval()* function.

更多信息请参阅 types。

5.12.6 类型对象

类型对象表示各种对象类型。对象的类型可通过内置函数*type()* 来获取。类型没有特殊的操作。标准库模块*types* 定义了所有标准内置类型的名称。

Types are written like this: `<type 'int'>`.

5.12.7 空对象

This object is returned by functions that don't explicitly return a value. It supports no special operations. There is exactly one null object, named `None` (a built-in name).

该对象的写法为 `None`。

5.12.8 省略符对象

This object is used by extended slice notation (see slicings). It supports no special operations. There is exactly one ellipsis object, named *Ellipsis* (a built-in name).

It is written as `Ellipsis`. When in a subscript, it can also be written as `...`, for example `seq[...]`.

5.12.9 未实现对象

This object is returned from comparisons and binary operations when they are asked to operate on types they don't support. See comparisons for more information.

该对象的写法为 `NotImplemented`。

5.12.10 布尔值

布尔值是两个常量对象 `False` 和 `True`。它们被用来表示逻辑上的真假（不过其他值也可被当作真值或假值）。在数字类的上下文中（例如被用作算术运算符的参数时），它们的行为分别类似于整数 0 和 1。内置函数*bool()* 可被用来将任意值转换为布尔值，只要该值可被解析为一个逻辑值（参见之前的逻辑值检测 部分）。

该对象的写法分别为 `False` 和 `True`。

5.12.11 内部对象

有关此对象的信息请参阅 types。其中描述了栈帧对象、回溯对象以及切片对象等等。

5.13 特殊属性

语言实现为部分对象类型添加了一些特殊的只读属性，它们具有各自的作用。其中一些并不会被*dir()* 内置函数所列出。

object.**__dict__**
一个字典或其他类型的映射对象，用于存储对象的（可写）属性。

object.**__methods__**
2.2 版后已移除: Use the built-in function *dir()* to get a list of an object's attributes. This attribute is no longer available.

object.**__members__**
2.2 版后已移除: Use the built-in function *dir()* to get a list of an object's attributes. This attribute is no longer available.

instance.**__class__**
类实例所属的类。

class.**__bases__**
由类对象的基类所组成的元组。

definition.**__name__**
The name of the class, type, function, method, descriptor, or generator instance.

The following attributes are only supported by *new-style classes*.

class.**__mro__**
此属性是由类组成的元组，在方法解析期间会基于它来查找基类。

class.**mro**()
此方法可被一个元类来重载，以为其实例定制方法解析顺序。它会在类实例化时被调用，其结果存储于*__mro__* 之中。

class.**__subclasses__**()
Each new-style class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive. Example:

```
>>> int.__subclasses__()
[<type 'bool'>]
```

备注

CHAPTER 6

内置异常

Exceptions should be class objects. The exceptions are defined in the module *exceptions*. This module never needs to be imported explicitly: the exceptions are provided in the built-in namespace as well as the *exceptions* module.

For class exceptions, in a `try` statement with an `except` clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which *it* is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an "associated value" indicating the detailed cause of the error. This may be a string or a tuple containing several items of information (e.g., an error code and a string explaining the code). The associated value is the second argument to the `raise` statement. If the exception class is derived from the standard root class *BaseException*, the associated value is present as the exception instance's `args` attribute.

用户代码可以引发内置异常。这可被用于测试异常处理程序或报告错误条件，"就像"在解释器引发了相同异常的情况时一样；但是请注意，没有任何机制能防止用户代码引发不适当的错误。

内置异常类可以被子类化以定义新的异常；鼓励程序员从*Exception* 类或它的某个子类而不是从*BaseException* 来派生新的异常。关于定义异常的更多信息可以在 Python 教程的 tut-userexceptions 部分查看。

The following exceptions are only used as base classes for other exceptions.

exception BaseException

The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that, use *Exception*). If *str()* or *unicode()* is called on an instance of this class, the representation of the argument(s) to the instance are returned, or the empty string when there were no arguments.

2.5 新版功能.

args

The tuple of arguments given to the exception constructor. Some built-in exceptions (like *IOError*) expect a certain number of arguments and assign a special meaning to the elements of this tuple, while others are usually called only with a single string giving an error message.

exception Exception

所有内置的非系统退出类异常都派生自此类。所有用户自定义异常也应当派生自此类。

在 2.5 版更改: Changed to inherit from *BaseException*.

exception StandardError
The base class for all built-in exceptions except *StopIteration*, *GeneratorExit*, *KeyboardInterrupt* and *SystemExit*. *StandardError* itself is derived from *Exception*.

exception ArithmeticError
此基类用于派生针对各种算术类错误而引发的内置异常: *OverflowError*, *ZeroDivisionError*, *FloatingPointError*。

exception BufferError
当与 缓冲区相关的操作无法执行时将被引发。

exception LookupError
此基类用于派生当映射或序列所使用的键或索引无效时引发的异常: *IndexError*, *KeyError*。这可以通过*codecs.lookup()* 来直接引发。

exception EnvironmentError
The base class for exceptions that can occur outside the Python system: *IOError*, *OSError*. When exceptions of this type are created with a 2-tuple, the first item is available on the instance's *errno* attribute (it is assumed to be an error number), and the second item is available on the `strerror` attribute (it is usually the associated error message). The tuple itself is also available on the `args` attribute.

1.5.2 新版功能.

When an *EnvironmentError* exception is instantiated with a 3-tuple, the first two items are available as above, while the third item is available on the `filename` attribute. However, for backwards compatibility, the `args` attribute contains only a 2-tuple of the first two constructor arguments.

The `filename` attribute is `None` when this exception is created with other than 3 arguments. The *errno* and `strerror` attributes are also `None` when the instance was created with other than 2 or 3 arguments. In this last case, `args` contains the verbatim constructor arguments as a tuple.

The following exceptions are the exceptions that are actually raised.

exception AssertionError
当 `assert` 语句失败时将被引发。

exception AttributeError
当属性引用 (参见 attribute-references) 或赋值失败时将被引发。(当一个对象根本不支持属性引用或属性赋值时则将引发*TypeError*。)

exception EOFError
Raised when one of the built-in functions (*input()* or *raw_input()*) hits an end-of-file condition (EOF) without reading any data. (N.B.: the *file.read()* and *file.readline()* methods return an empty string when they hit EOF.)

exception FloatingPointError
Raised when a floating point operation fails. This exception is always defined, but can only be raised when Python is configured with the `--with-fpectl` option, or the `WANT_SIGFPE_HANDLER` symbol is defined in the `pyconfig.h` file.

exception GeneratorExit
Raised when a *generator*'s `close()` method is called. It directly inherits from *BaseException* instead of *StandardError* since it is technically not an error.

2.5 新版功能.

在 2.6 版更改: Changed to inherit from *BaseException*.

exception IOError
Raised when an I/O operation (such as a `print` statement, the built-in *open()* function or a method of a file object) fails for an I/O-related reason, e.g., “file not found” or “disk full”.

This class is derived from *EnvironmentError*. See the discussion above for more information on exception instance attributes.

在 2.6 版更改: Changed *socket.error* to use this as a base class.

exception ImportError
Raised when an `import` statement fails to find the module definition or when a `from ... import` fails to find a name that is to be imported.

exception IndexError
Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not a plain integer, *TypeError* is raised.)

exception KeyError
当在现有键集合中找不到指定的映射（字典）键时将被引发。

exception KeyboardInterrupt
Raised when the user hits the interrupt key (normally `Control-C` or `Delete`). During execution, a check for interrupts is made regularly. Interrupts typed when a built-in function *input()* or *raw_input()* is waiting for input also raise this exception. The exception inherits from *BaseException* so as to not be accidentally caught by code that catches *Exception* and thus prevent the interpreter from exiting.

在 2.5 版更改: Changed to inherit from *BaseException*.

exception MemoryError
当一个操作耗尽内存但情况仍可（通过删除一些对象）进行挽救时将被引发。关联的值是一个字符串，指明是哪种（内部）操作耗尽了内存。请注意由于底层的内存管理架构（C 的 `malloc()` 函数），解释器也许并不总是能够从这种情况下完全恢复；但它毕竟可以引发一个异常，这样就能打印出栈回溯信息，以便找出导致问题的失控程序。

exception NameError
当某个局部或全局名称未找到时将被引发。此异常仅用于非限定名称。关联的值是一条错误信息，其中包含未找到的名称。

exception NotImplementedError
This exception is derived from *RuntimeError*. In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method.

1.5.2 新版功能.

exception OSError
This exception is derived from *EnvironmentError*. It is raised when a function returns a system-related error (not for illegal argument types or other incidental errors). The *errno* attribute is a numeric error code from `errno`, and the `strerror` attribute is the corresponding string, as would be printed by the C function `perror()`. See the module *errno*, which contains names for the error codes defined by the underlying operating system.

For exceptions that involve a file system path (such as `chdir()` or `unlink()`), the exception instance will contain a third attribute, `filename`, which is the file name passed to the function.

1.5.2 新版功能.

exception OverflowError
Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for long integers (which would rather raise *MemoryError* than give up) and for most operations with plain integers, which return a long integer instead. Because of the lack of standardization of floating point exception handling in C, most floating point operations also aren’t checked.

exception ReferenceError
此异常将在使用*weakref.proxy()* 函数所创建的弱引用来访问该引用的某个已被作为垃圾回收的属性时被引发。有关弱引用的更多信息请参阅*weakref* 模块。

2.2 新版功能: Previously known as the *weakref.ReferenceError* exception.

exception RuntimeError
当检测到一个不归属于任何其他类别的错误时将被引发。关联的值是一个指明究竟发生了什么问题的字符串。

exception StopIteration
Raised by an *iterator*'s *next()* method to signal that there are no further values. This is derived from *Exception* rather than *StandardError*, since this is not considered an error in its normal application.

2.2 新版功能.

exception SyntaxError
Raised when the parser encounters a syntax error. This may occur in an `import` statement, in an `exec` statement, in a call to the built-in function *eval()* or *input()*, or when reading the initial script or standard input (also interactively).

该类的实例包含有属性 `filename`, `lineno`, `offset` 和 `text` 用于方便地访问相应的详细信息。异常实例的*str()* 仅返回消息文本。

exception IndentationError
与不正确的缩进相关的语法错误的基类。这是*SyntaxError* 的一个子类。

exception TabError
当缩进包含对制表符和空格符不一致的使用时将被引发。这是*IndentationError* 的一个子类。

exception SystemError
当解释器发现内部错误，但情况看起来尚未严重到要放弃所有希望时将被引发。关联的值是一个指明发生了什么问题的字符串（表示为低层级的符号）。

你应当将此问题报告给你所用 Python 解释器的作者或维护人员。请确认报告 Python 解释器的版本号 (`sys.version`; 它也会在交互式 Python 会话开始时被打印出来)，具体的错误消息（异常所关联的值）以及可能触发该错误的程序源码。

exception SystemExit
This exception is raised by the *sys.exit()* function. When it is not handled, the Python interpreter exits; no stack traceback is printed. If the associated value is a plain integer, it specifies the system exit status (passed to C's `exit()` function); if it is `None`, the exit status is zero; if it has another type (such as a string), the object's value is printed and the exit status is one.

Instances have an attribute `code` which is set to the proposed exit status or error message (defaulting to `None`). Also, this exception derives directly from *BaseException* and not *StandardError*, since it is not technically an error.

对*sys.exit()* 的调用会被转换为一个异常以便能执行清理处理程序 (`try` 语句的 `finally` 子句)，并且使得调试器可以执行一段脚本而不必冒失去控制的风险。如果绝对确实地需要立即退出（例如在调用*os.fork()* 之后的子进程中）则可使用*os._exit()*.

The exception inherits from *BaseException* instead of *StandardError* or *Exception* so that it is not accidentally caught by code that catches *Exception*. This allows the exception to properly propagate up and cause the interpreter to exit.

在 2.5 版更改: Changed to inherit from *BaseException*.

exception TypeError
当一个操作或函数被应用于类型不适当的对象时将被引发。关联的值是一个字符串，给出有关类型不匹配的详情。

exception UnboundLocalError
当在函数或方法中对某个局部变量进行引用，但该变量并未绑定任何值时将被引发。此异常是*NameError* 的一个子类。

2.0 新版功能.

exception UnicodeError
当发生与 Unicode 相关的编码或解码错误时将被引发。此异常是*ValueError* 的一个子类。

UnicodeError 具有一些描述编码或解码错误的属性。例如 `err.object[err.start:err.end]` 会给出导致编解码器失败的特定无效输入。

encoding
引发错误的编码名称。

reason
描述特定编解码器错误的字符串。

object
编解码器试图要编码或解码的对象。

start
object 中无效数据的开始位置索引。

end
object 中无效数据的末尾位置索引（不含）。

2.0 新版功能.

exception UnicodeEncodeError
当在编码过程中发生与 Unicode 相关的错误时将被引发。此异常是*UnicodeError* 的一个子类。

2.3 新版功能.

exception UnicodeDecodeError
当在解码过程中发生与 Unicode 相关的错误时将被引发。此异常是*UnicodeError* 的一个子类。

2.3 新版功能.

exception UnicodeTranslateError
在转写过程中发生与 Unicode 相关的错误时将被引发。此异常是*UnicodeError* 的一个子类。

2.3 新版功能.

exception ValueError
当操作或函数接收到具有正确类型但值不适合的参数，并且情况不能用更精确的异常例如*IndexError* 来描述时将被引发。

exception VMSError
Only available on VMS. Raised when a VMS-specific error occurs.

exception WindowsError
Raised when a Windows-specific error occurs or when the error number does not correspond to an `errno` value. The `winerror` and `strerror` values are created from the return values of the `GetLastError()` and `FormatMessage()` functions from the Windows Platform API. The *errno* value maps the `winerror` value to corresponding `errno.h` values. This is a subclass of *OSError*.

2.0 新版功能.

在 2.5 版更改: Previous versions put the `GetLastError()` codes into *errno*.

exception ZeroDivisionError
当除法或取余运算的第二个参数为零时将被引发。关联的值是一个字符串，指明操作数和运算的类型。

The following exceptions are used as warning categories; see the *warnings* module for more information.

exception Warning
警告类别的基类。

exception UserWarning
用户代码所产生警告的基类。

exception DeprecationWarning
Base class for warnings about deprecated features.

exception PendingDeprecationWarning
Base class for warnings about features which will be deprecated in the future.

exception SyntaxWarning
与模糊的语法相关的警告的基类。

exception RuntimeWarning
与模糊的运行时行为相关的警告的基类。

exception FutureWarning
Base class for warnings about constructs that will change semantically in the future.

exception ImportWarning
与在模块导入中可能的错误相关的警告的基类。

2.5 新版功能.

exception UnicodeWarning
与 Unicode 相关的警告的基类。

2.5 新版功能.

exception BytesWarning
Base class for warnings related to bytes and bytearray.

2.6 新版功能.

6.1 异常层次结构

内置异常的类层级结构如下：

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
```

(下页继续)

(续上页)

```
      |     +-- EOFError
      |     +-- ImportError
      |     +-- LookupError
      |     |     +-- IndexError
      |     |     +-- KeyError
      |     +-- MemoryError
      |     +-- NameError
      |     |     +-- UnboundLocalError
      |     +-- ReferenceError
      |     +-- RuntimeError
      |     |     +-- NotImplementedError
      |     +-- SyntaxError
      |     |     +-- IndentationError
      |     |           +-- TabError
      |     +-- SystemError
      |     +-- TypeError
      |     +-- ValueError
      |           +-- UnicodeError
      |                 +-- UnicodeDecodeError
      |                 +-- UnicodeEncodeError
      |                 +-- UnicodeTranslateError
      +-- Warning
            +-- DeprecationWarning
            +-- PendingDeprecationWarning
            +-- RuntimeWarning
            +-- SyntaxWarning
            +-- UserWarning
            +-- FutureWarning
            +-- ImportWarning
            +-- UnicodeWarning
            +-- BytesWarning
```

CHAPTER 7

String Services

The modules described in this chapter provide a wide range of string manipulation operations.

In addition, Python's built-in string classes support the sequence type methods described in the *Sequence Types —str, unicode, list, tuple, bytearray, buffer, xrange* section, and also the string-specific methods described in the 字符串的方法 section. To output formatted strings use template strings or the % operator described in the *String Formatting Operations* section. Also, see the *re* module for string functions based on regular expressions.

7.1 string —常见的字符串操作

源代码： Lib/string.py

The *string* module contains a number of useful constants and classes, as well as some deprecated legacy functions that are also available as methods on strings. In addition, Python's built-in string classes support the sequence type methods described in the *Sequence Types —str, unicode, list, tuple, bytearray, buffer, xrange* section, and also the string-specific methods described in the 字符串的方法 section. To output formatted strings use template strings or the % operator described in the *String Formatting Operations* section. Also, see the *re* module for string functions based on regular expressions.

7.1.1 字符串常量

此模块中定义的常量为：

string.**ascii_letters**
下文所述*ascii_lowercase*和*ascii_uppercase*常量的拼连。该值不依赖于语言区域。

string.**ascii_lowercase**
小写字母'abcdefghijklmnopqrstuvwxyz'。该值不依赖于语言区域，不会发生改变。

string.**ascii_uppercase**
大写字母'ABCDEFGHIJKLMNOPQRSTUVWXYZ'。该值不依赖于语言区域，不会发生改变。

string.**digits**
字符串 '0123456789'。

string.**hexdigits**
字符串 '0123456789abcdefABCDEF'。

string.**letters**
The concatenation of the strings *lowercase* and *uppercase* described below. The specific value is locale-dependent, and will be updated when *locale.setlocale()* is called.

string.**lowercase**
A string containing all the characters that are considered lowercase letters. On most systems this is the string 'abcdefghijklmnopqrstuvwxyz'. The specific value is locale-dependent, and will be updated when *locale.setlocale()* is called.

string.**octdigits**
字符串 '01234567'。

string.**punctuation**
由在 C 语言区域中被视为标点符号的 ASCII 字符组成的字符串。

string.**printable**
String of characters which are considered printable. This is a combination of *digits*, *letters*, *punctuation*, and *whitespace*.

string.**uppercase**
A string containing all the characters that are considered uppercase letters. On most systems this is the string 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. The specific value is locale-dependent, and will be updated when *locale.setlocale()* is called.

string.**whitespace**
A string containing all characters that are considered whitespace. On most systems this includes the characters space, tab, linefeed, return, formfeed, and vertical tab.

7.1.2 自定义字符串格式化

2.6 新版功能.

The built-in str and unicode classes provide the ability to do complex variable substitutions and value formatting via the *str.format()* method described in **PEP 3101**. The *Formatter* class in the *string* module allows you to create and customize your own string formatting behaviors using the same implementation as the built-in *format()* method.

class string.**Formatter**
Formatter 类包含下列公有方法：

format(*format_string*, **args*, ***kwargs*)
首要的 API 方法。它接受一个格式字符串和任意一组位置和关键字参数。它只是一个调用*vformat()* 的包装器。

vformat(*format_string*, *args*, *kwargs*)
此函数执行实际的格式化操作。它被公开为一个单独的函数，用于需要传入一个预定义字母作为参数，而不是使用 *args 和 **kwargs 语法将字典解包为多个单独参数并重打包的情况。*vformat()* 完成将格式字符串分解为字符数据和替换字段的工作。它会调用下文所述的几种不同方法。

此外，*Formatter* 还定义了一些旨在被子类替换的方法：

parse(*format_string*)
循环遍历 format_string 并返回一个由可迭代对象组成的元组 (*literal_text*, *field_name*, *format_spec*, *conversion*)。它会被*vformat()* 用来将字符串分解为文本字面值或替换字段。

元组中的值在概念上表示一段字面文本加上一个替换字段。如果没有字面文本（如果连续出现两个替换字段就会发生这种情况），则 *literal_text* 将是一个长度为零的字符串。如果没有替换字段，则 *field_name*, *format_spec* 和 *conversion* 的值将为 `None`。

get_field(*field_name*, *args*, *kwargs*)
给定 *field_name* 作为*parse()* (见上文) 的返回值，将其转换为要格式化的对象。返回一个元组 (obj, used_key)。默认版本接受在 **PEP 3101** 所定义形式的字符串，例如 “0[name]” 或 “label.title”。*args* 和 *kwargs* 与传给*vformat()* 的一样。返回值 *used_key* 与*get_value()* 的 *key* 形参具有相同的含义。

get_value(*key*, *args*, *kwargs*)
提取给定的字段值。*key* 参数将为整数或字符串。如果是整数，它表示 *args* 中位置参数的索引；如果是字符串，它表示 *kwargs* 中的关键字参数名。

args 形参会被设为*vformat()* 的位置参数列表，而 *kwargs* 形参会被设为由关键字参数组成的字典。

对于复合字段名称，仅会为字段名称的第一个组件调用这些函数；后续组件会通过普通属性和索引操作来进行处理。

因此举例来说，字段表达式 ‘0.name’ 将导致调用*get_value()* 时附带 *key* 参数值 0。在*get_value()* 通过调用内置的*getattr()* 函数返回后将会查找 `name` 属性。

如果索引或关键字引用了一个不存在的项，则将引发*IndexError* 或*KeyError*。

check_unused_args(*used_args*, *args*, *kwargs*)
在必要时实现对未使用参数进行检测。此函数的参数是是格式字符串中实际引用的所有参数键的集合（整数表示位置参数，字符串表示名称参数），以及被传给 vformat 的 *args* 和 *kwargs* 的引用。未使用参数的集合可以根据这些形参计算出来。如果检测失败则*check_unused_args()* 应会引发一个异常。

format_field(*value*, *format_spec*)
format_field() 会简单地调用内置全局函数*format()*。提供该方法是为了让子类能够重载它。

convert_field(*value*, *conversion*)
使用给定的转换类型（来自*parse()* 方法所返回的元组）来转换（由*get_field()* 所返回的）值。默认版本支持 ‘s’ (str), ‘r’ (repr) 和 ‘a’ (ascii) 等转换类型。

7.1.3 格式字符串语法

The *str.format()* method and the *Formatter* class share the same syntax for format strings (although in the case of *Formatter*, subclasses can define their own format string syntax).

格式字符串包含有以花括号 `{}` 括起来的 “替换字段”。不在花括号之内的内容被视为字面文本，会不加修改地复制到输出中。如果你需要在字面文本中包含花括号字符，可以通过重复来转义: `{{` and `}}`。

替换字段的语法如下:

```
replacement_field  ::=   "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name         ::=   arg_name ("." attribute_name | "[" element_index "]")*
arg_name           ::=   [identifier | integer]
attribute_name     ::=   identifier
element_index      ::=   integer | index_string
index_string       ::=   <any source character except "]"> +
conversion         ::=   "r" | "s"
format_spec        ::=   <described in the next section>
```

用不太正式的术语来描述，替换字段开头可以用一个 *field_name* 指定要对值进行格式化并取代替换字符被插入到输出结果的对象。*field_name* 之后有可选的 *conversion* 字段，它是一个感叹号 `'!'` 加一个 *format_spec*，并以一个冒号 `':'` 打头。这些指明了替换值的非默认格式。

另请参阅格式规格迷你语言 一节。

field_name 本身以一个数字或关键字 *arg_name* 打头。如果为数字，则它指向一个位置参数，而如果为关键字，则它指向一个命名关键字参数。如果格式字符串中的数字 arg_names 为 0, 1, 2, …的序列，它们可以全部省略（而非部分省略），数字 0, 1, 2, …将会按顺序自动插入。由于 *arg_name* 不使用引号分隔，因此无法在格式字符串中指定任意的字典键 (例如字符串 `'10'` 或 `':-]'`)。*arg_name* 之后可以带上任意数量的索引或属性表达式。`'.name'` 形式的表达式会使用 *getattr()* 选择命名属性，而 `'[index]'` 形式的表达式会使用 `__getitem__()` 执行索引查找。

在 2.7 版更改: The positional argument specifiers can be omitted for *str.format()* and `unicode.format()`, so `'{} {}'` is equivalent to `'{0} {1}'`, `u'{} {}'` is equivalent to `u'{0} {1}'`.

一些简单的格式字符串示例

```
"First, thou shalt count to {0}"  # References first positional argument
"Bring me a {}"                   # Implicitly references the first positional␣
↪argument
"From {} to {}"                   # Same as "From {0} to {1}"
"My quest is {name}"              # References keyword argument 'name'
"Weight in tons {0.weight}"       # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"   # First element of keyword argument 'players'.
```

使用 *conversion* 字段在格式化之前进行类型强制转换。通常，格式化值的工作由值本身的 `__format__()` 方法来完成。但是，在某些情况下最好强制将类型格式化为一个字符串，覆盖其本身的格式化定义。通过在调用 `__format__()` 之前将值转换为字符串，可以绕过正常的格式化逻辑。

Two conversion flags are currently supported: `'!s'` which calls *str()* on the value, and `'!r'` which calls *repr()*.

几个例子:

```
"Harold's a clever {0!s}"         # Calls str() on the argument first
"Bring out the holy {name!r}"     # Calls repr() on the argument first
```

format_spec 字段包含值应如何呈现的规格描述，例如字段宽度、对齐、填充、小数精度等细节信息。每种值类型可以定义自己的“格式化迷你语言”或对 *format_spec* 的解读方式。

大多数内置类型都支持同样的格式化迷你语言，具体描述见下一节。

format_spec 字段还可以在其内部包含嵌套的替换字段。这些嵌套的替换字段可能包括字段名称、转换旗标和格式规格描述，但是不再允许更深层的嵌套。format_spec 内部的替换字段会在解读 *format_spec* 字符串之前先被解读。这将允许动态地指定特定值的格式。

请参阅格式示例 一节查看相关示例。

格式规格迷你语言

“Format specifications” are used within replacement fields contained within a format string to define how individual values are presented (see 格式字符串语法). They can also be passed directly to the built-in *format()* function. Each formattable type may define how the format specification is to be interpreted.

大多数内置类型都为格式规格实现了下列选项，不过某些格式化选项只被数值类型所支持。

A general convention is that an empty format string (`""`) produces the same result as if you had called *str()* on the value. A non-empty format string typically modifies the result.

标准格式说明符的一般形式如下:

```
format_spec  ::=  [[fill]align][sign][#][0][width][,][.precision][type]
fill         ::=  <any character>
align        ::=  "<" | ">" | "=" | "^"
sign         ::=  "+" | "-" | " "
width        ::=  integer
precision    ::=  integer
type         ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" |
```

If a valid *align* value is specified, it can be preceded by a *fill* character that can be any character and defaults to a space if omitted. It is not possible to use a literal curly brace ("`{`" or "`}`") as the *fill* character when using the *str.format()* method. However, it is possible to insert a curly brace with a nested replacement field. This limitation doesn't affect the *format()* function.

各种对齐选项的含义如下:

选项	含义
'<'	强制字段在可用空间内左对齐（这是大多数对象的默认值）。
'>'	强制字段在可用空间内右对齐（这是数字的默认值）。
'='	强制将填充放置在符号（如果有）之后但在数字之前。这用于以 "+000000120" 形式打印字段。此对齐选项仅对数字类型有效。当'0'紧接在字段宽度之前时，它成为默认值。
'^'	强制字段在可用空间内居中。

请注意，除非定义了最小字段宽度，否则字段宽度将始终与填充它的数据大小相同，因此在这种情况下，对齐选项没有意义。

sign 选项仅对数字类型有效，可以是以下之一:

选项	含义
'+'	表示标志应该用于正数和负数。
'-'	表示标志应仅用于负数（这是默认行为）。
space	表示应在正数上使用前导空格，在负数上使用减号。

The '#' option is only valid for integers, and only for binary, octal, or hexadecimal output. If present, it specifies that the output will be prefixed by '0b', '0o', or '0x', respectively.

',' 选项表示使用逗号作为千位分隔符。对于感应区域设置的分隔符，请改用 'n' 整数表示类型。

在 2.7 版更改: 添加了 ',' 选项 (另请参阅 **PEP 378**)。

width is a decimal integer defining the minimum field width. If not specified, then the field width will be determined by the content.

当未显式给出对齐方式时，在 *width* 字段前加一个零 ('0') 字段将为数字类型启用感知正负号的零填充。这相当于设置 *fill* 字符为 '0' 且 *alignment* 类型为 '='。

precision 是一个十进制数字，表示对于以 'f' and 'F' 格式化的浮点数值要在小数点后显示多少个数位，或者对于以 'g' 或 'G' 格式化的浮点数值要在小数点前后共显示多少个数位。对于非数字类型，该字段表示最大字段大小——换句话说就是要使用多少个来自字段内容的字符。对于整数值则不允许使用 *precision*。

最后，*type* 确定了数据应如何呈现。

可用的字符串表示类型是:

类型	含义
's'	字符串格式。这是字符串的默认类型，可以省略。
None	和 's' 一样。

可用的整数表示类型是：

类型	含义
'b'	二进制格式。输出以 2 为基数的数字。
'c'	字符。在打印之前将整数转换为相应的 unicode 字符。
'd'	十进制整数。输出以 10 为基数的数字。
'o'	八进制格式。输出以 8 为基数的数字。
'x'	Hex format. Outputs the number in base 16, using lower- case letters for the digits above 9.
'X'	Hex format. Outputs the number in base 16, using upper- case letters for the digits above 9.
'n'	数字。这与 'd' 相似，不同之处在于它会使用当前区域设置来插入适当的数字分隔字符。
None	和 'd' 相同。

在上述的表示类型之外，整数还可以通过下列的浮点表示类型来格式化 (除了 'n' 和 None)。当这样做时，会在格式化之前使用 *float()* 将整数转换为浮点数。

浮点数和小数值可用的表示类型有：

类型	含义
'e'	指数表示。以使用字母‘e’来标示指数的科学计数法打印数字。默认的精度为 6。
'E'	指数表示。与 'e' 相似，不同之处在于它使用大写字母‘E’作为分隔字符。
'f'	定点表示。将数字显示为一个定点数。默认的精确度为 6。
'F'	Fixed point notation. Same as 'f'.
'g'	常规格式。对于给定的精度 p >= 1，这会将数值舍入到 p 位有效数字，再将结果以定点格式或科学计数法进行格式化，具体取决于其值的大小。 The precise rules are as follows: suppose that the result formatted with presentation type 'e' and precision p-1 would have exponent exp. Then if -4 <= exp < p, the number is formatted with presentation type 'f' and precision p-1-exp. Otherwise, the number is formatted with presentation type 'e' and precision p-1. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it. 正负无穷，正负零和 nan 会分别被格式化为 inf, -inf, 0, -0 和 nan，无论精度如何设定。 精度 0 会被视为等同于精度 1。默认精度为 6。
'G'	常规格式。类似于 'g'，不同之处在于当数值非常大时会切换为 'E'。无穷与 NaN 也会表示为大写形式。
'n'	数字。这与 'g' 相似，不同之处在于它会使用当前区域设置来插入适当的数字分隔字符。
'%'	百分比。将数字乘以 100 并显示为定点 ('f') 格式，后面带一个百分号。
None	The same as 'g'.

格式示例

本节包含*str.format()* 语法的示例以及与旧式 `%` 格式化的比较。

该语法在大多数情况下与旧式的 `%` 格式化类似，只是增加了 `{}` 和 `:` 来取代 `%`。例如，, `'%03.2f'` 可以被改写为 `'{:03.2f}'`。

新的格式语法还支持新增的不同选项，将在以下示例中说明。

按位置访问参数:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 2.7+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # arguments' indices can be repeated
'abracadabra'
```

按名称访问参数:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.
↪81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

访问参数的属性:

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.
↪0.'
>>> class Point(object):
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

访问参数的项:

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

替代 `%s` 和 `%r`:

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

对齐文本以及指定宽度:

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'***********centered***********'
```

替代 %+f, %-f 和 % f 以及指定正负号:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -- same as '{:f}; {:f}'
'3.140000; -3.140000'
```

替代 %x 和 %o 以及转换基于不同进位制的值:

```
>>> # format also supports binary numbers
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
```

使用逗号作为千位分隔符:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

表示为百分数:

```
>>> points = 19.5
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 88.64%'
```

使用特定类型的专属格式化:

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

嵌套参数以及更复杂的示例:

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
```

(下页继续)

(续上页)

```
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print '{0:{width}{base}}'.format(num, base=base, width=width),
...     print
...
    5     5     5   101
    6     6     6   110
    7     7     7   111
    8     8    10  1000
    9     9    11  1001
   10     A    12  1010
   11     B    13  1011
```

7.1.4 模板字符串

2.4 新版功能.

Templates provide simpler string substitutions as described in **PEP 292**. Instead of the normal `%`-based substitutions, Templates support `$`-based substitutions, using the following rules:

- `$$` 为转义符号；它会被替换为单个的 `$`。
- `$identifier` names a substitution placeholder matching a mapping key of `"identifier"`. By default, `"identifier"` must spell a Python identifier. The first non-identifier character after the `$` character terminates this placeholder specification.
- `${identifier}` 等价于 `$identifier`。当占位符之后紧跟着有效的但又不是占位符一部分的标识符字符时需要使用，例如 `"${noun}ification"`。

在字符串的其他位置出现 `$` 将导致引发*ValueError*。

string 模块提供了实现这些规则的*Template* 类。*Template* 有下列方法：

class string.**Template**(*template*)

该构造器接受一个参数作为模板字符串。

substitute(*mapping*[, ***kws*])

Performs the template substitution, returning a new string. *mapping* is any dictionary-like object with keys that match the placeholders in the template. Alternatively, you can provide keyword arguments, where the keywords are the placeholders. When both *mapping* and *kws* are given and there are duplicates, the placeholders from *kws* take precedence.

safe_substitute(*mapping*[, ***kws*])

Like *substitute()*, except that if placeholders are missing from *mapping* and *kws*, instead of raising a *KeyError* exception, the original placeholder will appear in the resulting string intact. Also, unlike with *substitute()*, any other appearances of the `$` will simply return `$` instead of raising *ValueError*.

此方法被认为“安全”，因为虽然仍有可能发生其他异常，但它总是尝试返回可用的字符串而不是引发一个异常。从另一方面来说，*safe_substitute()* 也可能根本算不上安全，因为它将静默地忽略错误格式的模板，例如包含多余的分隔符、不成对的花括号或不是合法 Python 标识符的占位符等等。

Template 的实例还提供一个公有数据属性：

template
这是作为构造器的 *template* 参数被传入的对象。一般来说，你不应该修改它，但并不强制要求只读访问。

以下是一个如何使用模版的示例：

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

进阶用法：你可以派生 *Template* 的子类来自定义占位符语法、分隔符，或用于解析模板字符串的整个正则表达式。为此目的，你可以重载这些类属性：

- *delimiter* –This is the literal string describing a placeholder introducing delimiter. The default value is `$`. Note that this should *not* be a regular expression, as the implementation will call *re.escape()* on this string as needed.
- *idpattern* –This is the regular expression describing the pattern for non-braced placeholders (the braces will be added automatically as appropriate). The default value is the regular expression `[_a-z][_a-z0-9]*`.

作为另一种选项，你可以通过重载类属性 *pattern* 来提供整个正则表达式模式。如果你这样做，该值必须为一个具有四个命名捕获组的正则表达式对象。这些捕获组对应于上面已经给出的规则，以及无效占位符的规则：

- *escaped* –这个组匹配转义序列，在默认模式中即 `$$`。
- *named* –这个组匹配不带花括号的占位符名称；它不应当包含捕获组中的分隔符。
- *braced* –这个组匹配带有花括号的占位符名称；它不应当包含捕获组中的分隔符或者花括号。
- *invalid* –这个组匹配任何其他分隔符模式（通常为单个分隔符），并且它应当出现在正则表达式的末尾。

7.1.5 String functions

The following functions are available to operate on string and Unicode objects. They are not available as string methods.

`string.`**capwords**(*s*[, *sep*])
使用*str.split()* 将参数拆分为单词，使用*str.capitalize()* 将单词转为大写形式，使用*str.join()* 将大写的单词进行拼接。如果可选的第二个参数 *sep* 被省略或为 `None`，则连续的空白字符会被替换为单个空格符并且开头和末尾的空白字符会被移除，否则 *sep* 会被用来拆分和拼接单词。

`string.`**maketrans**(*from*, *to*)
Return a translation table suitable for passing to *translate()*, that will map each character in *from* into the character at the same position in *to*; *from* and *to* must have the same length.

注解： Don't use strings derived from *lowercase* and *uppercase* as arguments; in some locales, these don't have the same length. For case conversions, always use *str.lower()* and *str.upper()*.

7.1.6 Deprecated string functions

The following list of functions are also defined as methods of string and Unicode objects; see section 字符串的方法 for more information on those. You should consider these functions as deprecated, although they will not be removed until Python 3. The functions defined in this module are:

`string.`**`atof`**(*s*)
2.0 版后已移除: Use the `float()` built-in function.

Convert a string to a floating point number. The string must have the standard syntax for a floating point literal in Python, optionally preceded by a sign (+ or −). Note that this behaves identical to the built-in function `float()` when passed a string.

注解: When passing in a string, values for NaN and Infinity may be returned, depending on the underlying C library. The specific set of strings accepted which cause these values to be returned depends entirely on the C library and is known to vary.

`string.`**`atoi`**(*s*[, *base*])
2.0 版后已移除: Use the `int()` built-in function.

Convert string *s* to an integer in the given *base*. The string must consist of one or more digits, optionally preceded by a sign (+ or −). The *base* defaults to 10. If it is 0, a default base is chosen depending on the leading characters of the string (after stripping the sign): `0x` or `0X` means 16, `0` means 8, anything else means 10. If *base* is 16, a leading `0x` or `0X` is always accepted, though not required. This behaves identically to the built-in function `int()` when passed a string. (Also note: for a more flexible interpretation of numeric literals, use the built-in function `eval()`.)

`string.`**`atol`**(*s*[, *base*])
2.0 版后已移除: Use the `long()` built-in function.

Convert string *s* to a long integer in the given *base*. The string must consist of one or more digits, optionally preceded by a sign (+ or −). The *base* argument has the same meaning as for `atoi()`. A trailing `l` or `L` is not allowed, except if the base is 0. Note that when invoked without *base* or with *base* set to 10, this behaves identical to the built-in function `long()` when passed a string.

`string.`**`capitalize`**(*word*)
Return a copy of *word* with only its first character capitalized.

`string.`**`expandtabs`**(*s*[, *tabsize*])
Expand tabs in a string replacing them by one or more spaces, depending on the current column and the given tab size. The column number is reset to zero after each newline occurring in the string. This doesn't understand other non-printing characters or escape sequences. The tab size defaults to 8.

`string.`**`find`**(*s*, *sub*[, *start*[, *end*]])
Return the lowest index in *s* where the substring *sub* is found such that *sub* is wholly contained in `s[start:end]`. Return `-1` on failure. Defaults for *start* and *end* and interpretation of negative values is the same as for slices.

`string.`**`rfind`**(*s*, *sub*[, *start*[, *end*]])
Like `find()` but find the highest index.

`string.`**`index`**(*s*, *sub*[, *start*[, *end*]])
Like `find()` but raise `ValueError` when the substring is not found.

`string.`**`rindex`**(*s*, *sub*[, *start*[, *end*]])
Like `rfind()` but raise `ValueError` when the substring is not found.

`string.`**`count`**(*s*, *sub*[, *start*[, *end*]])
Return the number of (non-overlapping) occurrences of substring *sub* in string `s[start:end]`. Defaults for *start* and *end* and interpretation of negative values are the same as for slices.

string.**lower**(*s*)
Return a copy of *s*, but with upper case letters converted to lower case.

string.**split**(*s*[, *sep*[, *maxsplit*]])
Return a list of the words of the string *s*. If the optional second argument *sep* is absent or `None`, the words are separated by arbitrary strings of whitespace characters (space, tab, newline, return, formfeed). If the second argument *sep* is present and not `None`, it specifies a string to be used as the word separator. The returned list will then have one more item than the number of non-overlapping occurrences of the separator in the string. If *maxsplit* is given, at most *maxsplit* number of splits occur, and the remainder of the string is returned as the final element of the list (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

The behavior of split on an empty string depends on the value of *sep*. If *sep* is not specified, or specified as `None`, the result will be an empty list. If *sep* is specified as any string, the result will be a list containing one element which is an empty string.

string.**rsplit**(*s*[, *sep*[, *maxsplit*]])
Return a list of the words of the string *s*, scanning *s* from the end. To all intents and purposes, the resulting list of words is the same as returned by *split()*, except when the optional third argument *maxsplit* is explicitly specified and nonzero. If *maxsplit* is given, at most *maxsplit* number of splits –the *rightmost* ones –occur, and the remainder of the string is returned as the first element of the list (thus, the list will have at most `maxsplit+1` elements).

2.4 新版功能.

string.**splitfields**(*s*[, *sep*[, *maxsplit*]])
This function behaves identically to *split()*. (In the past, *split()* was only used with one argument, while *splitfields()* was only used with two arguments.)

string.**join**(*words*[, *sep*])
Concatenate a list or tuple of words with intervening occurrences of *sep*. The default value for *sep* is a single space character. It is always true that `string.join(string.split(s, sep), sep)` equals *s*.

string.**joinfields**(*words*[, *sep*])
This function behaves identically to *join()*. (In the past, *join()* was only used with one argument, while *joinfields()* was only used with two arguments.) Note that there is no *joinfields()* method on string objects; use the *join()* method instead.

string.**lstrip**(*s*[, *chars*])
Return a copy of the string with leading characters removed. If *chars* is omitted or `None`, whitespace characters are removed. If given and not `None`, *chars* must be a string; the characters in the string will be stripped from the beginning of the string this method is called on.

在 2.2.3 版更改: The *chars* parameter was added. The *chars* parameter cannot be passed in earlier 2.2 versions.

string.**rstrip**(*s*[, *chars*])
Return a copy of the string with trailing characters removed. If *chars* is omitted or `None`, whitespace characters are removed. If given and not `None`, *chars* must be a string; the characters in the string will be stripped from the end of the string this method is called on.

在 2.2.3 版更改: The *chars* parameter was added. The *chars* parameter cannot be passed in earlier 2.2 versions.

string.**strip**(*s*[, *chars*])
Return a copy of the string with leading and trailing characters removed. If *chars* is omitted or `None`, whitespace characters are removed. If given and not `None`, *chars* must be a string; the characters in the string will be stripped from the both ends of the string this method is called on.

在 2.2.3 版更改: The *chars* parameter was added. The *chars* parameter cannot be passed in earlier 2.2 versions.

string.**swapcase**(*s*)
Return a copy of *s*, but with lower case letters converted to upper case and vice versa.

string.**translate**(*s*, *table*[, *deletechars*])
Delete all characters from *s* that are in *deletechars* (if present), and then translate the characters using *table*, which must be a 256-character string giving the translation for each character value, indexed by its ordinal. If *table* is None, then only the character deletion step is performed.

string.**upper**(*s*)
Return a copy of *s*, but with lower case letters converted to upper case.

string.**ljust**(*s*, *width*[, *fillchar*])
string.**rjust**(*s*, *width*[, *fillchar*])
string.**center**(*s*, *width*[, *fillchar*])
These functions respectively left-justify, right-justify and center a string in a field of given width. They return a string that is at least *width* characters wide, created by padding the string *s* with the character *fillchar* (default is a space) until the given width on the right, left or both sides. The string is never truncated.

string.**zfill**(*s*, *width*)
Pad a numeric string *s* on the left with zero digits until the given *width* is reached. Strings starting with a sign are handled correctly.

string.**replace**(*s*, *old*, *new*[, *maxreplace*])
Return a copy of string *s* with all occurrences of substring *old* replaced by *new*. If the optional argument *maxreplace* is given, the first *maxreplace* occurrences are replaced.

7.2 re —正则表达式操作

This module provides regular expression matching operations similar to those found in Perl. Both patterns and strings to be searched can be Unicode strings as well as 8-bit strings.

正则表达式使用反斜杠（'\'）来表示特殊形式，或者把特殊字符转义成普通字符。而反斜杠在普通的 Python 字符串里也有相同的作用，所以就产生了冲突。比如说，要匹配一个字面上的反斜杠，正则表达式模式不得不写成 '\\\\'，因为正则表达式里匹配一个反斜杠必须是 \\，而每个反斜杠在普通的 Python 字符串里都要写成 \\。

解决办法是对于正则表达式样式使用 Python 的原始字符串表示法；在带有 'r' 前缀的字符串字面值中，反斜杠不必做任何特殊处理。因此 r"\n" 表示包含 '\' 和 'n' 两个字符的字符串，而 "\n" 则表示只包含一个换行符的字符串。样式在 Python 代码中通常都会使用这种原始字符串表示法来表示。

It is important to note that most regular expression operations are available as module-level functions and *RegexObject* methods. The functions are shortcuts that don't require you to compile a regex object first, but miss some fine-tuning parameters.

参见:

第三方模块 regex，提供了与标准库 *re* 模块兼容的 API 接口，同时还提供了额外的功能和更全面的 Unicode 支持。

7.2.1 正则表达式语法

一个正则表达式（或 RE）指定了一集与之匹配的字符串；模块内的函数可以让你检查某个字符串是否跟给定的正则表达式匹配（或者一个正则表达式是否匹配到一个字符串，这两种说法含义相同）。

Regular expressions can be concatenated to form new regular expressions; if *A* and *B* are both regular expressions, then *AB* is also a regular expression. In general, if a string *p* matches *A* and another string *q* matches *B*, the string *pq* will match AB. This holds unless *A* or *B* contain low precedence operations; boundary conditions between *A* and *B*; or have numbered group references. Thus, complex expressions can easily be constructed from simpler primitive expressions like

the ones described here. For details of the theory and implementation of regular expressions, consult the Friedl book referenced above, or almost any textbook about compiler construction.

以下是正则表达式格式的简要说明。更详细的信息和演示，参考 regex-howto。

正则表达式可以包含普通或者特殊字符。绝大部分普通字符，比如 'A', 'a', 或者 '0'，都是最简单的正则表达式。它们就匹配自身。你可以拼接普通字符，所以 last 匹配字符串 'last'. （在这一节的其他部分，我们将用 this special style 这种方式表示正则表达式，通常不带引号，要匹配的字符串用 'in single quotes'，单引号形式。）

Some characters, like '|' or '(', are special. Special characters either stand for classes of ordinary characters, or affect how the regular expressions around them are interpreted. Regular expression pattern strings may not contain null bytes, but can specify the null byte using the \number notation, e.g., '\x00'.

重复修饰符 (*, +, ?, {m,n}, 等) 不能直接嵌套。这样避免了非贪婪后缀 ? 修饰符，和其他实现中的修饰符产生的多义性。要应用一个内层重复嵌套，可以使用括号。比如，表达式 (?:a{6})* 匹配 6 个 'a' 字符重复任意次数。

特殊字符是：

'.' (点) 在默认模式，匹配除了换行的任意字符。如果指定了标签*DOTALL*，它将匹配包括换行符的任意字符。

'^' (插入符号) 匹配字符串的开头，并且在*MULTILINE* 模式也匹配换行后的首个符号。

'$' 匹配字符串尾或者换行符的前一个字符，在*MULTILINE* 模式匹配换行符的前一个字符。foo 匹配 'foo' 和 'foobar'，但正则 foo$ 只匹配 'foo'。更有趣的是，在 'foo1\nfoo2\n' 搜索 foo.$，通常匹配 'foo2'，但在*MULTILINE* 模式，可以匹配到 'foo1'；在 'foo\n' 搜索 $ 会找到两个空串：一个在换行前，一个在字符串最后。

'*' 对它前面的正则式匹配 0 到任意次重复，尽量多的匹配字符串。ab* 会匹配 'a', 'ab', 或者 'a'``后面跟随任意个 ``'b'。

'+' 对它前面的正则式匹配 1 到任意次重复。ab+ 会匹配 'a' 后面跟随 1 个以上到任意个 'b'，它不会匹配 'a'。

'?' 对它前面的正则式匹配 0 到 1 次重复。ab? 会匹配 'a' 或者 'ab'。

?, +?, ?? The '', '+', and '?' qualifiers are all *greedy*; they match as much text as possible. Sometimes this behaviour isn't desired; if the RE <.*> is matched against <a> b <c>, it will match the entire string, and not just <a>. Adding ? after the qualifier makes it perform the match in *non-greedy* or *minimal* fashion; as *few* characters as possible will be matched. Using the RE <.*?> will match only <a>.

"{m}" 对其之前的正则式指定匹配 *m* 个重复；少于 *m* 的话就会导致匹配失败。比如，a{6} 将匹配 6 个 'a'，但是不能是 5 个。

"{m, n}" Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, a{3,5} will match from 3 to 5 'a' characters. Omitting *m* specifies a lower bound of zero, and omitting *n* specifies an infinite upper bound. As an example, a{4,}b will match aaaab or a thousand 'a' characters followed by a b, but not aaab. The comma may not be omitted or the modifier would be confused with the previously described form.

{m,n}? 前一个修饰符的非贪婪模式，只匹配尽量少的字符次数。比如，对于 'aaaaaa'，a{3,5} 匹配 5 个 'a'，而 a{3,5}? 只匹配 3 个 'a'。

'\' 转义特殊字符（允许你匹配 '*', '?', 或者此类其他），或者表示一个特殊序列；特殊序列之后进行讨论。

如果你没有使用原始字符串（r'raw'）来表达样式，要牢记 Python 也使用反斜杠作为转义序列；如果转义序列不被 Python 的分析器识别，反斜杠和字符才能出现在字符串中。如果 Python 可以识别这个序列，那么反斜杠就应该重复两次。这将导致理解障碍，所以高度推荐，就算是最简单的表达式，也要使用原始字符串。

[] 用于表示一个字符集合。在一个集合中：

- 字符可以单独列出，比如 `[amk]` 匹配 `'a'`，`'m'`，或者 `'k'`。
- Ranges of characters can be indicated by giving two characters and separating them by a `'-'`, for example `[a-z]` will match any lowercase ASCII letter, `[0-5][0-9]` will match all the two-digits numbers from `00` to `59`, and `[0-9A-Fa-f]` will match any hexadecimal digit. If `-` is escaped (e.g. `[a\-z]`) or if it's placed as the first or last character (e.g. `[a-]`), it will match a literal `'-'`.
- 特殊字符在集合中，失去它的特殊含义。比如 `[(+*)]` 只会匹配这几个文法字符 `'('`, `'+'`, `'*'`, or `')'`。
- Character classes such as `\w` or `\S` (defined below) are also accepted inside a set, although the characters they match depends on whether *LOCALE* or *UNICODE* mode is in force.
- 不在集合范围内的字符可以通过 取反来进行匹配。如果集合首字符是 `'^'`，所有 不在集合内的字符将会被匹配，比如 `[^5]` 将匹配所有字符，除了 `'5'`，`[^^]` 将匹配所有字符，除了 `'^'`. `^` 如果不在集合首位，就没有特殊含义。
- 在集合内要匹配一个字符 `']'`，有两种方法，要么就在它之前加上反斜杠，要么就把它放到集合首位。比如，`[()[\]{}]` 和 `[]()[{}]` 都可以匹配括号。

'|' `A|B`, where A and B can be arbitrary REs, creates a regular expression that will match either A or B. An arbitrary number of REs can be separated by the `'|'` in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by `'|'` are tried from left to right. When one pattern completely matches, that branch is accepted. This means that once `A` matches, `B` will not be tested further, even if it would produce a longer overall match. In other words, the `'|'` operator is never greedy. To match a literal `'|'`, use `\|`, or enclose it inside a character class, as in `[|]`.

(...) Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the `\number` special sequence, described below. To match the literals `'('` or `')'`, use `\(` or `\)`, or enclose them inside a character class: `[(] [)]`.

(?…) 这是个扩展标记法（一个 `'?'` 跟随 `'('` 并无含义）。`'?'` 后面的第一个字符决定了这个构建采用什么样的语法。这种扩展通常并不创建新的组合；`(?P<name>...)` 是唯一的例外。以下是目前支持的扩展。

(?iLmsux) (One or more letters from the set `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'`.) The group matches the empty string; the letters set the corresponding flags: *re.I* (ignore case), *re.L* (locale dependent), *re.M* (multi-line), *re.S* (dot matches all), *re.U* (Unicode dependent), and *re.X* (verbose), for the entire regular expression. (The flags are described in 模块内容.) This is useful if you wish to include the flags as part of the regular expression, instead of passing a *flag* argument to the *re.compile()* function.

Note that the `(?x)` flag changes how the expression is parsed. It should be used first in the expression string, or after one or more whitespace characters. If there are non-whitespace characters before the flag, the results are undefined.

(?:…) 正则括号的非捕获版本。匹配在括号内的任何正则表达式，但该分组所匹配的子字符串 不能在执行匹配后被获取或是之后在模式中被引用。

(?P<name>…)（命名组合）类似正则组合，但是匹配到的子串组在外部是通过定义的 *name* 来获取的。组合名必须是有效的 Python 标识符，并且每个组合名只能用一个正则表达式定义，只能定义一次。一个符号组合同样是一个数字组合，就像这个组合没有被命名一样。

命名组合可以在三种上下文中引用。如果样式是 `(?P<quote>['"]).*?(?P=quote)` （也就是说，匹配单引号或者双引号括起来的字符串）：

引用组合“quote”的上下文	引用方法
在正则式自身内	• `(?P=quote)` (如示) • `\1`
when processing match object `m`	• `m.group('quote')` • `m.end('quote')` (等)
in a string passed to the `repl` argument of `re.sub()`	• `\g<quote>` • `\g<1>` • `\1`

(?P=name) 反向引用一个命名组合；它匹配前面那个叫 *name* 的命名组中匹配到的串同样的字串。

(?#…) 注释；里面的内容会被忽略。

(?=…) Matches if `...` matches next, but doesn't consume any of the string. This is called a lookahead assertion. For example, `Isaac (?=Asimov)` will match `'Isaac '` only if it's followed by `'Asimov'`.

(?!…) Matches if `...` doesn't match next. This is a negative lookahead assertion. For example, `Isaac (?!Asimov)` will match `'Isaac '` only if it's *not* followed by `'Asimov'`.

(?<=…) Matches if the current position in the string is preceded by a match for `...` that ends at the current position. This is called a *positive lookbehind assertion*. `(?<=abc)def` will find a match in `abcdef`, since the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, meaning that `abc` or `a|b` are allowed, but `a*` and `a{3,4}` are not. Group references are not supported even if they match strings of some fixed length. Note that patterns which start with positive lookbehind assertions will not match at the beginning of the string being searched; you will most likely want to use the *`search()`* function rather than the *`match()`* function:

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

这个例子搜索一个跟随在连字符后的单词:

```
>>> m = re.search('(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

(?<!…) Matches if the current position in the string is not preceded by a match for `...`. This is called a *negative lookbehind assertion*. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length and shouldn't contain group references. Patterns which start with negative lookbehind assertions may match at the beginning of the string being searched.

(?(id/name)yes-pattern|no-pattern) Will try to match with `yes-pattern` if the group with given *id* or *name* exists, and with `no-pattern` if it doesn't. `no-pattern` is optional and can be omitted. For example, `(<)?(\w+@\w+(?:\.\w+)+)(?(1)>)` is a poor email matching pattern, which will match with `'<user@host.com>'` as well as `'user@host.com'`, but not with `'<user@host.com'`.

2.4 新版功能.

The special sequences consist of `'\'` and a character from the list below. If the ordinary character is not on the list, then the resulting RE will match the second character. For example, `\$` matches the character `'$'`.

\number 匹配数字代表的组合。每个括号是一个组合，组合从 1 开始编号。比如 `(.+) \1` 匹配 `'the the'` 或者 `'55 55'`，但不会匹配 `'thethe'` (注意组合后面的空格)。这个特殊序列只能用于匹配前面 99 个组合。如果 *number* 的第一个数位是 0，或者 *number* 是三个八进制数，它将不会被看作是一个组合，而是八进制的数字值。在 `'['` 和 `']'` 字符集合内，任何数字转义都被看作是字符。

\A 只匹配字符串开始。

\b Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of alphanumeric or underscore characters, so the end of a word is indicated by whitespace or a non-alphanumeric, non-underscore character. Note that formally, `\b` is defined as the boundary between a `\w` and a `\W` character (or vice versa), or between `\w` and the beginning/end of the string, so the precise set of characters deemed to be alphanumeric depends on the values of the `UNICODE` and `LOCALE` flags. For example, `r'\bfoo\b'` matches `'foo'`, `'foo.'`, `'(foo)'`, `'bar foo baz'` but not `'foobar'` or `'foo3'`. Inside a character range, `\b` represents the backspace character, for compatibility with Python's string literals.

\B Matches the empty string, but only when it is *not* at the beginning or end of a word. This means that `r'py\B'` matches `'python'`, `'py3'`, `'py2'`, but not `'py'`, `'py.'`, or `'py!'`. `\B` is just the opposite of `\b`, so is also subject to the settings of `LOCALE` and `UNICODE`.

\d When the *UNICODE* flag is not specified, matches any decimal digit; this is equivalent to the set `[0-9]`. With *UNICODE*, it will match whatever is classified as a decimal digit in the Unicode character properties database.

\D When the *UNICODE* flag is not specified, matches any non-digit character; this is equivalent to the set `[^0-9]`. With *UNICODE*, it will match anything other than character marked as digits in the Unicode character properties database.

\s When the *UNICODE* flag is not specified, it matches any whitespace character, this is equivalent to the set `[ \t\n\r\f\v]`. The *LOCALE* flag has no extra effect on matching of the space. If *UNICODE* is set, this will match the characters `[ \t\n\r\f\v]` plus whatever is classified as space in the Unicode character properties database.

\S When the *UNICODE* flag is not specified, matches any non-whitespace character; this is equivalent to the set `[^ \t\n\r\f\v]` The *LOCALE* flag has no extra effect on non-whitespace match. If *UNICODE* is set, then any character not marked as space in the Unicode character properties database is matched.

\w When the *LOCALE* and *UNICODE* flags are not specified, matches any alphanumeric character and the underscore; this is equivalent to the set `[a-zA-Z0-9_]`. With *LOCALE*, it will match the set `[0-9_]` plus whatever characters are defined as alphanumeric for the current locale. If *UNICODE* is set, this will match the characters `[0-9_]` plus whatever is classified as alphanumeric in the Unicode character properties database.

\W When the *LOCALE* and *UNICODE* flags are not specified, matches any non-alphanumeric character; this is equivalent to the set `[^a-zA-Z0-9_]`. With *LOCALE*, it will match any character not in the set `[0-9_]`, and not defined as alphanumeric for the current locale. If *UNICODE* is set, this will match anything other than `[0-9_]` plus characters classified as not alphanumeric in the Unicode character properties database.

\Z 只匹配字符串尾。

If both *LOCALE* and *UNICODE* flags are included for a particular sequence, then *LOCALE* flag takes effect first followed by the *UNICODE*.

绝大部分 Python 的标准转义字符也被正则表达式分析器支持。:

```
\a      \b      \f      \n
\r      \t      \v      \x
\\
```

(注意 `\b` 被用于表示词语的边界，它只在字符集合内表示退格，比如 `[\b]` 。)

Octal escapes are included in a limited form: If the first digit is a 0, or if there are three octal digits, it is considered an octal escape. Otherwise, it is a group reference. As for string literals, octal escapes are always at most three digits in length.

参见:

Mastering Regular Expressions Book on regular expressions by Jeffrey Friedl, published by O'Reilly. The second edition of the book no longer covers Python at all, but the first edition covered writing good regular expression patterns in great detail.

7.2.2 模块内容

模块定义了几个函数，常量，和一个例外。有些函数是编译后的正则表达式方法的简化版本（少了一些特性）。绝大部分重要的应用，总是会先将正则表达式编译，之后在进行操作。

re.**compile**(*pattern*, *flags=0*)

Compile a regular expression pattern into a regular expression object, which can be used for matching using its *match()* and *search()* methods, described below.

这个表达式的行为可以通过指定 标记的值来改变。值可以是以下任意变量，可以通过位的 OR 操作来结合（| 操作符）。

序列

```
prog = re.compile(pattern)
result = prog.match(string)
```

等价于

```
result = re.match(pattern, string)
```

如果需要多次使用这个正则表达式的话，使用*re.compile()* 和保存这个正则对象以便复用，可以让程序更加高效。

注解: The compiled versions of the most recent patterns passed to *re.match()*, *re.search()* or *re.compile()* are cached, so programs that use only a few regular expressions at a time needn't worry about compiling regular expressions.

---

re.**DEBUG**

Display debug information about compiled expression.

re.**I**

re.**IGNORECASE**

Perform case-insensitive matching; expressions like `[A-Z]` will match lowercase letters, too. This is not affected by the current locale. To get this effect on non-ASCII Unicode characters such as `ü` and `Ü`, add the *UNICODE* flag.

re.**L**

re.**LOCALE**

Make `\w`, `\W`, `\b`, `\B`, `\s` and `\S` dependent on the current locale.

re.**M**

re.**MULTILINE**

When specified, the pattern character `'^'` matches at the beginning of the string and at the beginning of each line (immediately following each newline); and the pattern character `'$'` matches at the end of the string and at the end of each line (immediately preceding each newline). By default, `'^'` matches only at the beginning of the string, and `'$'` only at the end of the string and immediately before the newline (if any) at the end of the string.

re.**S**

re.**DOTALL**

Make the `'.'` special character match any character at all, including a newline; without this flag, `'.'` will match anything *except* a newline.

re.**U**
re.**UNICODE**
Make the \w, \W, \b, \B, \d, \D, \s and \S sequences dependent on the Unicode character properties database. Also enables non-ASCII matching for *IGNORECASE*.

2.0 新版功能.

re.**X**
re.**VERBOSE**
这个标记允许你编写更具可读性更友好的正则表达式。通过分段和添加注释。空白符号会被忽略，除非在一个字符集合当中或者由反斜杠转义，或者在 *?, (?: or (?P<…> 分组之内。当一个行内有 # 不在字符集和转义序列，那么它之后的所有字符都是注释。

意思就是下面两个正则表达式等价地匹配一个十进制数字：

```
a = re.compile(r"""\d +  # the integral part
                   \.    # the decimal point
                   \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

re.**search**(*pattern*, *string*, *flags=0*)
Scan through *string* looking for the first location where the regular expression *pattern* produces a match, and return a corresponding *MatchObject* instance. Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

re.**match**(*pattern*, *string*, *flags=0*)
If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding *MatchObject* instance. Return None if the string does not match the pattern; note that this is different from a zero-length match.

注意即便是*MULTILINE* 多行模式，*re.match()* 也只匹配字符串的开始位置，而不匹配每行开始。

如果你想定位 *string* 的任何位置，使用*search()* 来替代（也可参考*search() vs. match()* ）

re.**split**(*pattern*, *string*, *maxsplit=0*, *flags=0*)
Split *string* by the occurrences of *pattern*. If capturing parentheses are used in *pattern*, then the text of all groups in the pattern are also returned as part of the resulting list. If *maxsplit* is nonzero, at most *maxsplit* splits occur, and the remainder of the string is returned as the final element of the list. (Incompatibility note: in the original Python 1.5 release, *maxsplit* was ignored. This has been fixed in later releases.)

```
>>> re.split('\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split('(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split('\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

If there are capturing groups in the separator and it matches at the start of the string, the result will start with an empty string. The same holds for the end of the string:

```
>>> re.split('(\W+)', '...words, words...')
['', '...', 'words', ', ', 'words', '...', '']
```

That way, separator components are always found at the same relative indices within the result list (e.g., if there's one capturing group in the separator, the 0th, the 2nd and so forth).

Note that *split* will never split a string on an empty pattern match. For example:

```
>>> re.split('x*', 'foo')
['foo']
>>> re.split("(?m)^$", "foo\n\nbar\n")
['foo\n\nbar\n']
```

在 2.7 版更改: 增加了可选标记参数。

re.**findall**(*pattern*, *string*, *flags=0*)

对 *string* 返回一个不重复的 *pattern* 的匹配列表，*string* 从左到右进行扫描，匹配按找到的顺序返回。如果样式里存在一到多个组，就返回一个组合列表；就是一个元组的列表（如果样式里有超过一个组合的话）。空匹配也会包含在结果里。

注解： Due to the limitation of the current implementation the character following an empty match is not included in a next match, so `findall(r'^|\w+', 'two words')` returns `['', 'wo', 'words']` (note missed "t"). This is changed in Python 3.7.

1.5.2 新版功能.

在 2.4 版更改: 增加了可选标记参数。

re.**finditer**(*pattern*, *string*, *flags=0*)

Return an *iterator* yielding *MatchObject* instances over all non-overlapping matches for the RE *pattern* in *string*. The *string* is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result. See also the note about *findall()*.

2.2 新版功能.

在 2.4 版更改: 增加了可选标记参数。

re.**sub**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in *string* by the replacement *repl*. If the pattern isn't found, *string* is returned unchanged. *repl* can be a string or a function; if it is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted to a carriage return, and so forth. Unknown escapes such as `\j` are left alone. Backreferences, such as `\6`, are replaced with the substring matched by group 6 in the pattern. For example:

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...        r'static PyObject*\npy_\1(void)\n{',
...        'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

If *repl* is a function, it is called for every non-overlapping occurrence of *pattern*. The function takes a single match object argument, and returns the replacement string. For example:

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

The pattern may be a string or an RE object.

The optional argument *count* is the maximum number of pattern occurrences to be replaced; *count* must be a non-negative integer. If omitted or zero, all occurrences will be replaced. Empty matches for the pattern are replaced only when not adjacent to a previous match, so `sub('x*', '-', 'abc')` returns `'-a-b-c-'`.

在字符串类型的 *repl* 参数里，如上所述的转义和向后引用中，`\g<name>` 会使用命名组合 `name`，（在 `(?P<name>…)` 语法中定义）`\g<number>` 会使用数字组；`\g<2>` 就是 `\2`，但它避免了二义性，如 `\g<2>0`。`\20` 就会被解释为组 20，而不是组 2 后面跟随一个字符 `'0'`。向后引用 `\g<0>` 把 *pattern* 作为一整个组进行引用。

在 2.7 版更改: 增加了可选标记参数。

`re.`**`subn`**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)
行为与*sub()* 相同，但是返回一个元组（字符串， 替换次数）.

在 2.7 版更改: 增加了可选标记参数。

`re.`**`escape`**(*pattern*)
Escape all the characters in *pattern* except ASCII letters and numbers. This is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it. For example:

```
>>> print re.escape('python.exe')
python\.exe

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print '[%s]+' % re.escape(legal_chars)
[abcdefghijklmnopqrstuvwxyz0123456789\!\#\$\%\&\'\*\+\-\.\^\_\`\|\~\:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print '|'.join(map(re.escape, sorted(operators, reverse=True)))
\/|\-|\+|\*\*|\*
```

`re.`**`purge`**()
清除正则表达式缓存。

`exception` `re.`**`error`**
Exception raised when a string passed to one of the functions here is not a valid regular expression (for example, it might contain unmatched parentheses) or when some other error occurs during compilation or matching. It is never an error if a string contains no match for a pattern.

7.2.3 正则表达式对象（正则对象）

`class` `re.`**`RegexObject`**
The *RegexObject* class supports the following methods and attributes:

`search`(*string*[, *pos*[, *endpos*]])
Scan through *string* looking for a location where this regular expression produces a match, and return a corresponding *MatchObject* instance. Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

可选的第二个参数 *pos* 给出了字符串中开始搜索的位置索引；默认为 `0`，它不完全等价于字符串切片；`'^'` 样式字符匹配字符串真正的开头，和换行符后面的第一个字符，但不会匹配索引规定开始的位置。

The optional parameter *endpos* limits how far the string will be searched; it will be as if the string is *endpos* characters long, so only the characters from *pos* to `endpos - 1` will be searched for a match. If *endpos* is less than *pos*, no match will be found, otherwise, if *rx* is a compiled regular expression object, `rx.search(string, 0, 50)` is equivalent to `rx.search(string[:50], 0)`.

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Match at index 0
<_sre.SRE_Match object at ...>
>>> pattern.search("dog", 1)  # No match; search doesn't include the "d"
```

match(*string*[, *pos*[, *endpos*]])
If zero or more characters at the *beginning* of *string* match this regular expression, return a corresponding `MatchObject` instance. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

The optional *pos* and *endpos* parameters have the same meaning as for the `search()` method.

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)   # Match as "o" is the 2nd character of "dog".
<_sre.SRE_Match object at ...>
```

If you want to locate a match anywhere in *string*, use `search()` instead (see also *search() vs. match()*).

split(*string*, *maxsplit=0*)
等价于`split()` 函数，使用了编译后的样式。

findall(*string*[, *pos*[, *endpos*]])
Similar to the `findall()` function, using the compiled pattern, but also accepts optional *pos* and *endpos* parameters that limit the search region like for `match()`.

finditer(*string*[, *pos*[, *endpos*]])
Similar to the `finditer()` function, using the compiled pattern, but also accepts optional *pos* and *endpos* parameters that limit the search region like for `match()`.

sub(*repl*, *string*, *count=0*)
等价于`sub()` 函数，使用了编译后的样式。

subn(*repl*, *string*, *count=0*)
等价于`subn()` 函数，使用了编译后的样式。

flags
The regex matching flags. This is a combination of the flags given to `compile()` and any `(?...)` inline flags in the pattern.

groups
捕获组合的数量。

groupindex
映射由 `(?P<id>)` 定义的命名符号组合和数字组合的字典。如果没有符号组，那字典就是空的。

pattern
The pattern string from which the RE object was compiled.

7.2.4 匹配对象

class re.**MatchObject**
Match objects always have a boolean value of `True`. Since `match()` and `search()` return `None` when there is no match, you can test whether there was a match with a simple `if` statement:

```
match = re.search(pattern, string)
if match:
    process(match)
```

匹配对象支持以下方法和属性：

expand(*template*)
Return the string obtained by doing backslash substitution on the template string *template*, as done by the `sub()` method. Escapes such as `\n` are converted to the appropriate characters, and numeric backreferences

(\1, \2) and named backreferences (\g<1>, \g<name>) are replaced by the contents of the corresponding group.

group([*group1*, ...])

Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, *group1* defaults to zero (the whole match is returned). If a *groupN* argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group. If a group number is negative or larger than the number of groups defined in the pattern, an *IndexError* exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is None. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # The entire match
'Isaac Newton'
>>> m.group(1)       # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)       # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)    # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

如果正则表达式使用了 (?P<name>…) 语法，*groupN* 参数就也可能是命名组合的名字。如果一个字符串参数在样式中未定义为组合名，一个*IndexError* 就 raise。

A moderately complicated example:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds
↪")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

Named groups can also be referred to by their index:

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

If a group matches multiple times, only the last match is accessible:

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the last match.
'c3'
```

groups([*default*])

Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The *default* argument is used for groups that did not participate in the match; it defaults to None. (Incompatibility note: in the original Python 1.5 release, if the tuple was one element long, a string would be returned instead. In later versions (from 1.5.1 on), a singleton tuple is returned in such cases.)

For example:

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

If we make the decimal place and everything after it optional, not all groups might participate in the match. These groups will default to `None` unless the *default* argument is given:

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # Second group defaults to None.
('24', None)
>>> m.groups('0')   # Now, the second group defaults to '0'.
('24', '0')
```

groupdict([*default*])
Return a dictionary containing all the *named* subgroups of the match, keyed by the subgroup name. The *default* argument is used for groups that did not participate in the match; it defaults to `None`. For example:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds
↪")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

start([*group*])
end([*group*])
返回 *group* 匹配到的字串的开始和结束标号。*group* 默认为 0（意思是整个匹配的子串）。如果 *group* 存在，但未产生匹配，就返回 `-1` 。对于一个匹配对象 *m*，和一个未参与匹配的组 *g* ，组 *g* (等价于 `m.group(g)`) 产生的匹配是

```
m.string[m.start(g):m.end(g)]
```

注意 `m.start(group)` 将会等于 `m.end(group)` ，如果 *group* 匹配一个空字符串的话。比如，在 `m = re.search('b(c?)', 'cba')` 之后，`m.start(0)` 为 1, `m.end(0)` 为 2, `m.start(1)` 和 `m.end(1)` 都是 2, `m.start(2)` raise 一个 *IndexError* 例外。

An example that will remove *remove_this* from email addresses:

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

span([*group*])
For *MatchObject* *m*, return the 2-tuple `(m.start(group), m.end(group))`. Note that if *group* did not contribute to the match, this is `(-1, -1)`. *group* defaults to zero, the entire match.

pos
The value of *pos* which was passed to the *search()* or *match()* method of the *RegexObject*. This is the index into the string at which the RE engine started looking for a match.

endpos
The value of *endpos* which was passed to the *search()* or *match()* method of the *RegexObject*. This is the index into the string beyond which the RE engine will not go.

lastindex
捕获组的最后一个匹配的整数索引值，或者 `None` 如果没有匹配产生的话。比如，对于字符串 `'ab'`，表达式 `(a)b`, `((a)(b))`, 和 `((ab))` 将得到 `lastindex == 1` ，而 `(a)(b)` 会得到 `lastindex == 2` 。

lastgroup
最后一个匹配的命名组名字，或者 None 如果没有产生匹配的话。

re
The regular expression object whose *match()* or *search()* method produced this *MatchObject* instance.

string
The string passed to *match()* or *search()*.

7.2.5 Examples

Checking For a Pair

在这个例子里，我们使用以下辅助函数来更好的显示匹配对象：

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

假设你在写一个扑克程序，一个玩家的一手牌为五个字符的串，每个字符表示一张牌，”a” 就是 A，“k” K，“q” Q，“j” J，“t” 为 10，“2” 到 “9” 表示 2 到 9。

To see if a given string is a valid hand, one could do the following:

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q"))  # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e"))  # Invalid.
>>> displaymatch(valid.match("akt"))    # Invalid.
>>> displaymatch(valid.match("727ak"))  # Valid.
"<Match: '727ak', groups=()>"
```

That last hand, `"727ak"`, contained a pair, or two of the same valued cards. To match this with a regular expression, one could use backreferences as such:

```
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match("717ak"))     # Pair of 7s.
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match("718ak"))     # No pairs.
>>> displaymatch(pair.match("354aa"))     # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

To find out what card the pair consists of, one could use the *group()* method of *MatchObject* in the following manner:

```
>>> pair.match("717ak").group(1)
'7'

# Error because re.match() returns None, which doesn't have a group() method:
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'
```

(下页继续)

(续上页)

```
>>> pair.match("354aa").group(1)
'a'
```

模拟 scanf()

Python 目前没有一个类似 c 函数 `scanf()` 的替代品。正则表达式通常比 `scanf()` 格式字符串要更强大一些，但也带来更多复杂性。下面的表格提供了 `scanf()` 格式符和正则表达式大致相同的映射。

scanf() 格式符	正则表达式
%c	.
%5c	.{5}
%d	[-+]?\d+
%e, %E, %f, %g	[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?
%i	[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)
%o	[-+]?[0-7]+
%s	\S+
%u	\d+
%x, %X	[-+]?(0[xX])?[\dA-Fa-f]+

从文件名和数字提取字符串

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

你可以使用 `scanf()` 格式化

```
%s - %d errors, %d warnings
```

等价的正则表达式是：

```
(\S+) - (\d+) errors, (\d+) warnings
```

search() vs. match()

Python 提供了两种不同的操作：基于*re.match()* 检查字符串开头，或者*re.search()* 检查字符串的任意位置（默认 Perl 中的行为）。

例如

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("c", "abcdef")   # Match
<_sre.SRE_Match object at ...>
```

在*search()* 中，可以用 `'^'` 作为开始来限制匹配到字符串的首位

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("^c", "abcdef")  # No match
>>> re.search("^a", "abcdef")  # Match
<_sre.SRE_Match object at ...>
```

Note however that in *MULTILINE* mode *match()* only matches at the beginning of the string, whereas using *search()* with a regular expression beginning with `'^'` will match at the beginning of each line.

```
>>> re.match('X', 'A\nB\nX', re.MULTILINE)  # No match
>>> re.search('^X', 'A\nB\nX', re.MULTILINE)  # Match
<_sre.SRE_Match object at ...>
```

建立一个电话本

split() 将字符串用参数传递的样式分隔开。这个方法对于转换文本数据到易读而且容易修改的数据结构，是很有用的，如下面的例子证明。

First, here is the input. Normally it may come from a file, here we are using triple-quoted string syntax:

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

条目用一个或者多个换行符分开。现在我们将字符串转换为一个列表，每个非空行都有一个条目:

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
'Frank Burger: 925.541.7625 662 South Dogwood Way',
'Heather Albrecht: 548.326.4584 919 Park Place']
```

最终，将每个条目分割为一个由名字、姓氏、电话号码和地址组成的列表。我们为*split()* 使用了 maxsplit 形参，因为地址中包含有被我们作为分割模式的空格符:

```
>>> [re.split(":? ", entry, 3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

`:?` 样式匹配姓后面的冒号，因此它不出现在结果列表中。如果 maxsplit 设置为 `4` ，我们还可以从地址中获取到房间号:

```
>>> [re.split(":? ", entry, 4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

文字整理

sub() 替换字符串中出现的样式的每一个实例。这个例子证明了使用*sub()* 来整理文字，或者随机化每个字符的位置，除了首位和末尾字符

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

找到所有副词

findall() matches *all* occurrences of a pattern, not just the first one as *search()* does. For example, if a writer wanted to find all of the adverbs in some text, they might use *findall()* in the following manner:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']
```

找到所有副词和位置

If one wants more information about all matches of a pattern than the matched text, *finditer()* is useful as it provides instances of *MatchObject* instead of strings. Continuing with the previous example, if a writer wanted to find all of the adverbs *and their positions* in some text, they would use *finditer()* in the following manner:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly", text):
...     print '%02d-%02d: %s' % (m.start(), m.end(), m.group(0))
07-16: carefully
40-47: quickly
```

原始字符记法

Raw string notation (`r"text"`) keeps regular expressions sane. Without it, every backslash (`'\'`) in a regular expression would have to be prefixed with another one to escape it. For example, the two following lines of code are functionally identical:

```
>>> re.match(r"\W(.)\1\W", " ff ")
<_sre.SRE_Match object at ...>
>>> re.match("\\W(.)\\1\\W", " ff ")
<_sre.SRE_Match object at ...>
```

When one wants to match a literal backslash, it must be escaped in the regular expression. With raw string notation, this means `r"\\"`. Without raw string notation, one must use `"\\\\"`, making the following lines of code functionally identical:

```
>>> re.match(r"\\", r"\\")
<_sre.SRE_Match object at ...>
>>> re.match("\\\\", r"\\")
<_sre.SRE_Match object at ...>
```

7.3 struct —Interpret strings as packed binary data

This module performs conversions between Python values and C structs represented as Python strings. This can be used in handling binary data stored in files or from network connections, among other sources. It uses 格式字符串 as compact descriptions of the layout of the C structs and the intended conversion to/from Python values.

注解： 默认情况下，打包给定 C 结构的结果会包含填充字节以使得所涉及的 C 类型保持正确的对齐；类似地，对齐在解包时也会被纳入考虑。选择此种行为的目的是使得被打包结构的字节能与相应 C 结构在内存中的布局完全一致。要处理平台独立的数据格式或省略隐式的填充字节，请使用 `standard` 大小和对齐而不是 `native` 大小和对齐：详情参见字节顺序，大小和对齐方式。

7.3.1 函数和异常

此模块定义了下列异常和函数：

exception `struct.`**error**
会在多种场合下被引发的异常；其参数为一个描述错误信息的字符串。

`struct.`**pack**(*fmt*, *v1*, *v2*, ...)
Return a string containing the values `v1, v2, ...` packed according to the given format. The arguments must match the values required by the format exactly.

`struct.`**pack_into**(*fmt*, *buffer*, *offset*, *v1*, *v2*, ...)
Pack the values `v1, v2, ...` according to the given format, write the packed bytes into the writable *buffer* starting at *offset*. Note that the offset is a required argument.

2.5 新版功能.

`struct.`**unpack**(*fmt*, *string*)
Unpack the string (presumably packed by `pack(fmt, ...)`) according to the given format. The result is a tuple even if it contains exactly one item. The string must contain exactly the amount of data required by the format (`len(string)` must equal `calcsize(fmt)`).

`struct.`**unpack_from**(*fmt*, *buffer*[, *offset=0*])
Unpack the *buffer* according to the given format. The result is a tuple even if it contains exactly one item. The *buffer* must contain at least the amount of data required by the format (`len(buffer[offset:])` must be at least `calcsize(fmt)`).

2.5 新版功能.

`struct.`**calcsize**(*fmt*)
Return the size of the struct (and hence of the string) corresponding to the given format.

7.3.2 格式字符串

格式字符串是用来在打包和解包数据时指定预期布局的机制。它们使用指定被打包/解包数据类型的格式字符 进行构建。此外，还有一些特殊字符用来控制字节顺序，大小和对齐方式。

字节顺序，大小和对齐方式

默认情况下，C 类型以机器的本机格式和字节顺序表示，并在必要时通过跳过填充字节进行正确对齐（根据 C 编译器使用的规则）。

或者，根据下表，格式字符串的第一个字符可用于指示打包数据的字节顺序，大小和对齐方式：

字符	字节顺序	大小	对齐方式
@	按原字节	按原字节	按原字节
=	按原字节	标准	无
<	小端	标准	无
>	大端	标准	无
!	网络（= 大端）	标准	无

如果第一个字符不是其中之一，则假定为 '@'。

本机字节顺序可能为大端或是小端，取决于主机系统的不同。例如，Intel x86 和 AMD64 (x86-64) 是小端的；Motorola 68000 和 PowerPC G5 是大端的；ARM 和 Intel Itanium 具有可切换的字节顺序（双端）。请使用 sys.byteorder 来检查你的系统字节顺序。

本机大小和对齐方式是使用 C 编译器的 sizeof 表达式来确定的。这总是会与本机字节顺序相绑定。

标准大小仅取决于格式字符；请参阅格式字符 部分中的表格。

请注意 '@' 和 '=' 之间的区别：两个都使用本机字节顺序，但后者的大小和对齐方式是标准化的。

格式 '!' 适合给那些宣称他们记不得网络字节顺序是大端还是小端的可怜人使用。

没有什么方式能指定非本机字节顺序（强制字节对调）；请正确选择使用 '<' 或 '>'。

注释:

(1) 填充只会在连续结构成员之间自动添加。填充不会添加到已编码结构的开头和末尾。

(2) 当使用非本机大小和对齐方式即‘<’，‘>’，‘=’, and ‘!’时不会添加任何填充。

(3) 要将结构的末尾对齐到符合特定类型的对齐要求，请以该类型代码加重复计数的零作为格式结束。参见例子。

格式字符

格式字符具有以下含义；C 和 Python 值之间的按其指定类型的转换应当是相当明显的。‘标准大小’列是指当使用标准大小时以字节表示的已打包值大小；也就是当格式字符串以 '<', '>', '!' 或 '=' 之一开头的情况。当使用本机大小时，已打包值的大小取决于具体的平台。

格式	C 类型	Python 类型	标准大小	注释
x	填充字节	无		
c	char	string of length 1	1	
b	signed char	整数	1	(3)
B	unsigned char	整数	1	(3)
?	_Bool	bool	1	(1)
h	short	整数	2	(3)
H	unsigned short	整数	2	(3)
i	int	整数	4	(3)
I	unsigned int	整数	4	(3)
l	long	整数	4	(3)
L	unsigned long	整数	4	(3)
q	long long	整数	8	(2), (3)
Q	unsigned long long	整数	8	(2), (3)
f	float	浮点数	4	(4)
d	double	浮点数	8	(4)
s	char[]	string		
p	char[]	string		
P	void *	整数		(5), (3)

注释:

(1) '?' 转换码对应于 C99 定义的 _Bool 类型。如果此类型不可用，则使用 char 来模拟。在标准模式下，它总是以一个字节表示。

2.6 新版功能.

(2) The 'q' and 'Q' conversion codes are available in native mode only if the platform C compiler supports C long long, or, on Windows, __int64. They are always available in standard modes.

2.2 新版功能.

(3) When attempting to pack a non-integer using any of the integer conversion codes, if the non-integer has a __index__() method then that method is called to convert the argument to an integer before packing. If no __index__() method exists, or the call to __index__() raises *TypeError*, then the __int__() method is tried. However, the use of __int__() is deprecated, and will raise *DeprecationWarning*.

在 2.7 版更改: Use of the __index__() method for non-integers is new in 2.7.

在 2.7 版更改: Prior to version 2.7, not all integer conversion codes would use the __int__() method to convert, and *DeprecationWarning* was raised only for float arguments.

(4) For the 'f' and 'd' conversion codes, the packed representation uses the IEEE 754 binary32 (for 'f') or binary64 (for 'd') format, regardless of the floating-point format used by the platform.

(5) 'P' 格式字符仅对本机字节顺序可用（选择为默认或使用 '@' 字节顺序字符）。字节顺序字符 '=' 选择使用基于主机系统的小端或大端排序。struct 模块不会将其解读为本机排序，因此 'P' 格式将不可用。

格式字符之前可以带有整数重复计数。例如，格式字符串 '4h' 的含义与 'hhhh' 完全相同。

格式之间的空白字符会被忽略；但是计数及其格式字符中不可有空白字符。

For the 's' format character, the count is interpreted as the size of the string, not a repeat count like for the other format characters; for example, '10s' means a single 10-byte string, while '10c' means 10 characters. If a count is not given, it defaults to 1. For packing, the string is truncated or padded with null bytes as appropriate to make it fit. For unpacking, the resulting string always has exactly the specified number of bytes. As a special case, '0s' means a single, empty string (while '0c' means 0 characters).

The 'p' format character encodes a "Pascal string", meaning a short variable-length string stored in a *fixed number of bytes*, given by the count. The first byte stored is the length of the string, or 255, whichever is smaller. The bytes of the string follow. If the string passed in to *pack()* is too long (longer than the count minus 1), only the leading count-1 bytes of the string are stored. If the string is shorter than count-1, it is padded with null bytes so that exactly count bytes in all are used. Note that for *unpack()*, the 'p' format character consumes count bytes, but that the string returned can never contain more than 255 characters.

For the 'P' format character, the return value is a Python integer or long integer, depending on the size needed to hold a pointer when it has been cast to an integer type. A *NULL* pointer will always be returned as the Python integer 0. When packing pointer-sized values, Python integer or long integer objects may be used. For example, the Alpha and Merced processors use 64-bit pointer values, meaning a Python long integer will be used to hold the pointer; other platforms use 32-bit pointers and will use a Python integer.

对于 '?' 格式字符，返回值为*True*或*False*。在打包时将会使用参数对象的逻辑值。以本机或标准 bool 类型表示的 0 或 1 将被打包，任何非零值在解包时将为 True。

例子

注解: 所有示例都假定使用一台大端机器的本机字节顺序、大小和对齐方式。

打包/解包三个整数的基础示例:

```
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', '\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('hhl')
8
```

解包的字段可通过将它们赋值给变量或将结果包装为一个具名元组来命名:

```
>>> record = 'raymond   \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name='raymond   ', serialnum=4658, school=264, gradelevel=8)
```

格式字符的顺序可能对大小产生影响，因为满足对齐要求所需的填充是不同的:

```
>>> pack('ci', '*', 0x12131415)
'*\x00\x00\x00\x12\x13\x14\x15'
>>> pack('ic', 0x12131415, '*')
'\x12\x13\x14\x15*'
>>> calcsize('ci')
8
>>> calcsize('ic')
5
```

以下格式 'llh0l' 指定在末尾有两个填充字节，假定 long 类型按 4 个字节的边界对齐:

```
>>> pack('llh0l', 1, 2, 3)
'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

这仅当本机大小和对齐方式生效时才会起作用；标准大小和对齐方式并不会强制进行任何对齐。

参见:

模块*array* 被打包为二进制存储的同质数据。

模块*xdrlib* 打包和解包 XDR 数据。

7.3.3 类

struct 模块还定义了以下类型:

class struct.**Struct**(*format*)
返回一个新的 Struct 对象，它会根据格式字符串 *format* 来写入和读取二进制数据。一次性地创建 Struct 对象并调用其方法相比使用同样的格式调用*struct* 函数更为高效，因为这样格式字符串只需被编译一次。

2.5 新版功能.

已编译的 Struct 对象支持以下方法和属性:

pack(*v1*, *v2*, ...)
Identical to the *pack()* function, using the compiled format. (`len(result)` will equal `self.size`.)

pack_into(*buffer*, *offset*, *v1*, *v2*, ...)
等价于*pack_into()* 函数，使用了已编译的格式。

unpack(*string*)
Identical to the *unpack()* function, using the compiled format. (`len(string)` must equal `self.size`).

unpack_from(*buffer*, *offset=0*)
Identical to the *unpack_from()* function, using the compiled format. (`len(buffer[offset:])` must be at least `self.size`).

format
用于构造此 Struct 对象的格式字符串。

size
The calculated size of the struct (and hence of the string) corresponding to *format*.

7.4 difflib —计算差异的辅助工具

2.1 新版功能.

此模块提供用于比较序列的类和函数。例如，它可以用于比较文件，并可以产生各种格式的不同信息，包括 HTML 和上下文以及统一格式的差异点。有关目录和文件的比较，请参见*filecmp* 模块。

class difflib.**SequenceMatcher**
This is a flexible class for comparing pairs of sequences of any type, so long as the sequence elements are *hashable*. The basic algorithm predates, and is a little fancier than, an algorithm published in the late 1980's by Ratcliff and Obershelp under the hyperbolic name "gestalt pattern matching." The idea is to find the longest contiguous matching subsequence that contains no "junk" elements (the Ratcliff and Obershelp algorithm doesn't address junk). The same idea is then applied recursively to the pieces of the sequences to the left and to the right of the matching subsequence. This does not yield minimal edit sequences, but does tend to yield matches that "look right" to people.

耗时: 基本 Ratcliff-Obershelp 算法在最坏情况下为立方时间而在一般情况下为平方时间。*SequenceMatcher* 在最坏情况下为平方时间而在一般情况下的行为受到序列中有多少相同元素这一因素的微妙影响；在最佳情况下则为线性时间。

自动垃圾启发式计算: *SequenceMatcher* 支持使用启发式计算来自动将特定序列项视为垃圾。这种启发式计算会统计每个单独项在序列中出现的次数。如果某一项（在第一项之后）的重复次数超过序列长度的 1% 并且序列长度至少有 200 项，该项会被标记为“热门”并被视为序列匹配中的垃圾。这种启发式计算可以通过在创建*SequenceMatcher* 时将 `autojunk` 参数设为 `False` 来关闭。

2.7.1 新版功能: *autojunk* 形参。

class difflib.**Differ**

这个类的作用是比较由文本行组成的序列，并产生可供人阅读的差异或增量信息。Differ 统一使用*SequenceMatcher* 来完成行序列的比较以及相似（接近匹配）行内部字符序列的比较。

Differ 增量的每一行均以双字母代码打头：

双字母代码	含义
`'- '`	行为序列 1 所独有
`'+ '`	行为序列 2 所独有
`'  '`	行在两序列中相同
`'? '`	行不存在于任一输入序列

以‘`?`’打头的行尝试将视线引至行以外而不存在于任一输入序列的差异。如果序列包含制表符则这些行可能会令人感到迷惑。

class difflib.**HtmlDiff**

这个类可用于创建 HTML 表格（或包含表格的完整 HTML 文件）以并排地逐行显示文本比较，行间与行外的更改将突出显示。此表格可以基于完全或上下文差异模式来生成。

这个类的构造函数：

__init__(*tabsize=8*, *wrapcolumn=None*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

初始化*HtmlDiff* 的实例。

tabsize 是一个可选关键字参数，指定制表位的间隔，默认值为 `8`。

wrapcolumn 是一个可选关键字参数，指定行文本自动打断并换行的列位置，默认值为 `None` 表示不自动换行。

linejunk 和 *charjunk* 均是可选关键字参数，会传入*ndiff()* (被*HtmlDiff* 用来生成并排显示的 HTML 差异)。请参阅*ndiff()* 文档了解参数默认值及其说明。

下列是公开的方法

make_file(*fromlines*, *tolines [, fromdesc][, todesc][, context][, numlines]*)

比较 *fromlines* 和 *tolines* (字符串列表) 并返回一个字符串，表示一个完整 HTML 文件，其中包含各行差异的表格，行间与行外的更改将突出显示。

fromdesc 和 *todesc* 均是可选关键字参数，指定来源/目标文件的列标题字符串（默认均为空白字符串）。

context 和 *numlines* 均是可选关键字参数。当只要显示上下文差异时就将 *context* 设为 `True`，否则默认值 `False` 为显示完整文件。*numlines* 默认为 `5`。当 *context* 为 `True` 时 *numlines* 将控制围绕突出显示差异部分的上下文行数。当 *context* 为 `False` 时 *numlines* 将控制在使用“next”超链接时突出显示差异部分之前所显示的行数（设为零则会导致“next”超链接将下一个突出显示差异部分放在浏览器顶端，不添加任何前导上下文）。

make_table(*fromlines*, *tolines [, fromdesc][, todesc][, context][, numlines]*)

比较 *fromlines* 和 *tolines* (字符串列表) 并返回一个字符串，表示一个包含各行差异的完整 HTML 表格，行间与行外的更改将突出显示。

此方法的参数与*make_file()* 方法的相同。

Tools/scripts/diff.py 是这个类的命令行前端，其中包含一个很好的使用示例。

2.4 新版功能.

difflib.**context_diff**(*a, b[, fromfile][, tofile][, fromfiledate][, tofiledate][, n][, lineterm]*)

比较 *a* 和 *b* (字符串列表)；返回上下文差异格式的增量信息 (一个产生增量行的*generator*)。

所谓上下文差异是一种只显示有更改的行再加几个上下文行的紧凑形式。更改被显示为之前/之后的样式。上下文行数由 *n* 设定，默认为三行。

By default, the diff control lines (those with `***` or `---`) are created with a trailing newline. This is helpful so that inputs created from *file.readlines()* result in diffs that are suitable for use with *file.writelines()* since both the inputs and outputs have trailing newlines.

对于没有末尾换行符的输入，应将 *lineterm* 参数设为 `""`，这样输出内容将统一不带换行符。

上下文差异格式通常带有一个记录文件名和修改时间的标头。这些信息的部分或全部可以使用字符串 *fromfile*, *tofile*, *fromfiledate* 和 *tofiledate* 来指定。修改时间通常以 ISO 8601 格式表示。如果未指定，这些字符串默认为空。

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> for line in context_diff(s1, s2, fromfile='before.py', tofile='after.py'):
...     sys.stdout.write(line)
*** before.py
--- after.py
***************
*** 1,4 ****
! bacon
! eggs
! ham
  guido
--- 1,4 ----
! python
! eggy
! hamster
  guido
```

请参阅*difflib 的命令行接口* 获取更详细的示例。

2.3 新版功能.

difflib.**get_close_matches**(*word, possibilities[, n][, cutoff]*)

返回由最佳“近似”匹配构成的列表。*word* 为一个指定目标近似匹配的序列（通常为字符串），*possibilities* 为一个由用于匹配 *word* 的序列构成的列表（通常为字符串列表）。

可选参数 *n* (默认为 3) 指定最多返回多少个近似匹配；*n* 必须大于 0.

可选参数 *cutoff* (默认为 0.6) 是一个 [0, 1] 范围内的浮点数。与 *word* 相似度得分未达到该值的候选匹配将被忽略。

候选匹配中（不超过 *n* 个）的最佳匹配将以列表形式返回，按相似度得分排序，最相似的排在最前面。

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('apple', keyword.kwlist)
```

(下页继续)

(续上页)

```
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

difflib.**ndiff**(*a, b[, linejunk][, charjunk]*)

比较 *a* 和 *b* (字符串列表)；返回*Differ* 形式的增量信息 (一个产生增量行的*generator*)。

可选关键字形参 *linejunk* 和 *charjunk* 均为过滤函数 (或为 None)：

linejunk: A function that accepts a single string argument, and returns true if the string is junk, or false if not. The default is (None), starting with Python 2.3. Before then, the default was the module-level function *IS_LINE_JUNK()*, which filters out lines without visible characters, except for at most one pound character ('#'). As of Python 2.3, the underlying *SequenceMatcher* class does a dynamic analysis of which lines are so frequent as to constitute noise, and this usually works better than the pre-2.3 default.

charjunk: A function that accepts a character (a string of length 1), and returns if the character is junk, or false if not. The default is module-level function *IS_CHARACTER_JUNK()*, which filters out whitespace characters (a blank or tab; note: bad idea to include newline in this!).

Tools/scripts/ndiff.py 是这个函数的命令行前端。

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(1),
...              'ore\ntree\nemu\n'.splitlines(1))
>>> print ''.join(diff),
- one
?  ^
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

difflib.**restore**(*sequence*, *which*)

返回两个序列中产生增量的那一个。

给出一个由*Differ.compare()* 或*ndiff()* 产生的 序列，提取出来自文件 1 或 2 (*which* 形参) 的行，去除行前缀。

示例:

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(1),
...              'ore\ntree\nemu\n'.splitlines(1))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print ''.join(restore(diff, 1)),
one
two
three
>>> print ''.join(restore(diff, 2)),
ore
tree
emu
```

difflib.**unified_diff**(*a, b[, fromfile][, tofile][, fromfiledate][, tofiledate][, n][, lineterm]*)

比较 *a* 和 *b* (字符串列表)；返回统一差异格式的增量信息 (一个产生增量行的*generator*)。

所以统一差异是一种只显示有更改的行再加几个上下文行的紧凑形式。更改被显示为内联的样式（而不是分开的之前/之后文本块)。上下文行数由 *n* 设定，默认为三行。

By default, the diff control lines (those with `---`, `+++`, or `@@`) are created with a trailing newline. This is helpful so that inputs created from *file.readlines()* result in diffs that are suitable for use with *file.writelines()* since both the inputs and outputs have trailing newlines.

对于没有末尾换行符的输入，应将 *lineterm* 参数设为 `""`，这样输出内容将统一不带换行符。

上下文差异格式通常带有一个记录文件名和修改时间的标头。这些信息的部分或全部可以使用字符串 *fromfile*, *tofile*, *fromfiledate* 和 *tofiledate* 来指定。修改时间通常以 ISO 8601 格式表示。如果未指定，这些字符串默认为空。

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> for line in unified_diff(s1, s2, fromfile='before.py', tofile='after.py'):
...     sys.stdout.write(line)
--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
 guido
```

请参阅*difflib 的命令行接口* 获取更详细的示例。

2.3 新版功能.

`difflib.`**`IS_LINE_JUNK`**(*line*)

Return true for ignorable lines. The line *line* is ignorable if *line* is blank or contains a single `'#'`, otherwise it is not ignorable. Used as a default for parameter *linejunk* in *ndiff()* before Python 2.3.

`difflib.`**`IS_CHARACTER_JUNK`**(*ch*)

Return true for ignorable characters. The character *ch* is ignorable if *ch* is a space or tab, otherwise it is not ignorable. Used as a default for parameter *charjunk* in *ndiff()*.

参见:

模式匹配：格式塔方法 John W. Ratcliff 和 D. E. Metzener 对于一种类似算法的讨论。此文于 1988 年 7 月发表于 Dr. Dobb's Journal。

7.4.1 SequenceMatcher 对象

SequenceMatcher 类具有这样的构造器：

`class` `difflib.`**`SequenceMatcher`**(*isjunk=None*, *a=''*, *b=''*, *autojunk=True*)

Optional argument *isjunk* must be `None` (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is "junk" and should be ignored. Passing `None` for *isjunk* is equivalent to passing `lambda x: 0`; in other words, no elements are ignored. For example, pass:

```
lambda x: x in " \t"
```

如果你以字符序列的形式对行进行比较，并且不希望区分空格符或硬制表符。

可选参数 *a* 和 *b* 为要比较的序列；两者默认为空字符串。两个序列的元素都必须为*hashable*。

可选参数 *autojunk* 可用于启用自动垃圾启发式计算。

2.7.1 新版功能: *autojunk* 形参。

SequenceMatcher 对象具有以下方法：

set_seqs(*a*, *b*)
设置要比较的两个序列。

SequenceMatcher 计算并缓存有关第二个序列的详细信息，这样如果你想要将一个序列与多个序列进行比较，可使用*set_seq2()* 一次性地设置该常用序列并重复地对每个其他序列各调用一次*set_seq1()*。

set_seq1(*a*)
设置要比较的第一个序列。要比较的第二个序列不会改变。

set_seq2(*b*)
设置要比较的第二个序列。要比较的第一个序列不会改变。

find_longest_match(*alo*, *ahi*, *blo*, *bhi*)
找出 `a[alo:ahi]` 和 `b[blo:bhi]` 中的最长匹配块。

如果 *isjunk* 被省略或为 `None`，*find_longest_match()* 将返回 `(i, j, k)` 使得 `a[i:i+k]` 等于 `b[j:j+k]`，其中 `alo <= i <= i+k <= ahi` 并且 `blo <= j <= j+k <= bhi`。对于所有满足这些条件的 `(i', j', k')`，如果 `i == i', j <= j'` 也被满足，则附加条件 `k >= k', i <= i'`。换句话说，对于所有最长匹配块，返回在 *a* 当中最先出现的一个，而对于在 *a* 当中最先出现的所有最长匹配块，则返回在 *b* 当中最先出现的一个。

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

如果提供了 *isjunk*，将按上述规则确定第一个最长匹配块，但额外附加不允许块内出现垃圾元素的限制。然后将通过（仅）匹配两边的垃圾元素来尽可能地扩展该块。这样结果块绝对不会匹配垃圾元素，除非同样的垃圾元素正好与有意义的匹配相邻。

这是与之前相同的例子，但是将空格符视为垃圾。这将防止 `' abcd'` 直接与第二个序列末尾的 `' abcd'` 相匹配。而只可以匹配 `'abcd'`，并且是匹配第二个序列最左边的 `'abcd'`：

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

如果未找到匹配块，此方法将返回 `(alo, blo, 0)`。

在 2.6 版更改: 此方法将返回一个*named tuple* `Match(a, b, size)`。

get_matching_blocks()
返回描述非重叠匹配子序列的三元组列表。每个三元组的形式为 `(i, j, n)`，其含义为 `a[i:i+n] == b[j:j+n]`。这些三元组按 *i* 和 *j* 单调递增排列。

最后一个三元组用于占位，其值为 `(len(a), len(b), 0)`。它是唯一 `n == 0` 的三元组。如果 `(i, j, n)` 和 `(i', j', n')` 是在列表中相邻的三元组，且后者不是列表中的最后一个三元组，则 `i+n < i'` 或 `j+n < j'`；换句话说，相邻的三元组总是描述非相邻的相等块。

在 2.5 版更改: The guarantee that adjacent triples always describe non-adjacent blocks was implemented.

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

get_opcodes()
返回描述如何将 *a* 变为 *b* 的 5 元组列表，每个元组的形式为 `(tag, i1, i2, j1, j2)`。在第一个元组中 `i1 == j1 == 0`，而在其余的元组中 *i1* 等于前一个元组的 *i2*，并且 *j1* 也等于前一个元组的 *j2*。

tag 值为字符串，其含义如下：

值	含义
'replace'	a[i1:i2] 应由 b[j1:j2] 替换。
'delete'	a[i1:i2] 应被删除。请注意在此情况下 j1 == j2。
'insert'	b[j1:j2] 应插入到 a[i1:i1]。请注意在此情况下 i1 == i2。
'equal'	a[i1:i2] == b[j1:j2] (两个子序列相同)。

For example:

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...    print ("%7s a[%d:%d] (%s) b[%d:%d] (%s)" %
...           (tag, i1, i2, a[i1:i2], j1, j2, b[j1:j2]))
 delete a[0:1] (q) b[0:0] ()
  equal a[1:3] (ab) b[0:2] (ab)
replace a[3:4] (x) b[2:3] (y)
  equal a[4:6] (cd) b[3:5] (cd)
 insert a[6:6] () b[5:6] (f)
```

get_grouped_opcodes([*n*])
返回一个带有最多 *n* 行上下文的分组的*generator*。

从*get_opcodes()* 所返回的组开始，此方法会拆分出较小的更改簇并消除没有更改的间隔区域。

这些分组以与*get_opcodes()* 相同的格式返回。

2.3 新版功能.

ratio()
返回一个取值范围 [0, 1] 的浮点数作为序列相似性度量。

其中 T 是两个序列中元素的总数量，M 是匹配的数量，即 2.0*M / T。请注意如果两个序列完全相同则该值为 1.0，如果两者完全不同则为 0.0。

如果*get_matching_blocks()* 或*get_opcodes()* 尚未被调用则此方法运算消耗较大，在此情况下你可能需要先调用*quick_ratio()* 或*real_quick_ratio()* 来获取一个上界。

quick_ratio()
相对快速地返回一个*ratio()* 的上界。

real_quick_ratio()
非常快速地返回一个*ratio()* 的上界。

这三个返回匹配部分占字符总数的比率的方法可能由于不同的近似级别而给出不一样的结果，但是 quick_ratio() 和 real_quick_ratio() 总是会至少与 ratio() 一样大：

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

7.4.2 SequenceMatcher 的示例

This example compares two strings, considering blanks to be "junk:"

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                     "private Thread currentThread;",
...                     "private volatile Thread currentThread;")
```

ratio() 返回一个 [0, 1] 范围内的整数作为两个序列相似性的度量。根据经验，ratio() 值超过 0.6 就意味着两个序列是近似匹配的：

```
>>> print round(s.ratio(), 3)
0.866
```

如果你只对两个序列相匹配的位置感兴趣，则 get_matching_blocks() 就很方便：

```
>>> for block in s.get_matching_blocks():
...     print "a[%d] and b[%d] match for %d elements" % block
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

请注意 get_matching_blocks() 返回的最后一个元组总是只用于占位的 (len(a), len(b), 0)，这也是元组末尾元素（匹配的元素数量）为 0 的唯一情况。

如果你想要知道如何将第一个序列转成第二个序列，可以使用 get_opcodes()：

```
>>> for opcode in s.get_opcodes():
...     print "%6s a[%d:%d] b[%d:%d]" % opcode
 equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
 equal a[8:29] b[17:38]
```

参见：

- 此模块中的*get_close_matches()* 函数显示了如何基于*SequenceMatcher* 构建简单的代码来执行有用的功能。
- 使用*SequenceMatcher* 构建小型应用的 简易版本控制方案。

7.4.3 Differ 对象

请注意*Differ* 所生成的增量并不保证是 **最小**差异。相反，最小差异往往是违反直觉的，因为它们会同步任何可能的地方，有时甚至意外产生相距 100 页的匹配。将同步点限制为连续匹配保留了一些局部性概念，这偶尔会带来产生更长差异的代价。

Differ 类具有这样的构造器：

class difflib.**Differ**([*linejunk*[, *charjunk*]])

可选关键字形参 *linejunk* 和 *charjunk* 均为过滤函数 (或为 None)：

linejunk: 接受单个字符串作为参数的函数，如果其为垃圾字符串则返回真值。默认值为 None，意味着没有任何行会被视为垃圾行。

charjunk: 接受单个字符（长度为 1 的字符串）作为参数的函数，如果其为垃圾字符则返回真值。默认值为 None，意味着没有任何字符会被视为垃圾字符。

Differ 对象是通过一个单独方法来使用（生成增量）的：

compare(*a*, *b*)
比较两个由行组成的序列，并生成增量（一个由行组成的序列）。

Each sequence must contain individual single-line strings ending with newlines. Such sequences can be obtained from the *readlines()* method of file-like objects. The delta generated also consists of newline-terminated strings, ready to be printed as-is via the *writelines()* method of a file-like object.

7.4.4 Differ 示例

This example compares two texts. First we set up the texts, sequences of individual single-line strings ending with newlines (such sequences can also be obtained from the *readlines()* method of file-like objects):

```
>>> text1 = '''  1. Beautiful is better than ugly.
...   2. Explicit is better than implicit.
...   3. Simple is better than complex.
...   4. Complex is better than complicated.
... '''.splitlines(1)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = '''  1. Beautiful is better than ugly.
...   3.   Simple is better than complex.
...   4. Complicated is better than complex.
...   5. Flat is better than nested.
... '''.splitlines(1)
```

接下来我们实例化一个 Differ 对象：

```
>>> d = Differ()
```

请注意在实例化*Differ*对象时我们可以传入函数来过滤掉“垃圾”行和字符。详情参见*Differ()*构造器说明。

最后，我们比较两个序列：

```
>>> result = list(d.compare(text1, text2))
```

result 是一个字符串列表，让我们将其美化打印出来：

```
>>> from pprint import pprint
>>> pprint(result)
['    1. Beautiful is better than ugly.\n',
 '-   2. Explicit is better than implicit.\n',
 '-   3. Simple is better than complex.\n',
 '+   3.   Simple is better than complex.\n',
 '?     ++\n',
 '-   4. Complex is better than complicated.\n',
 '?            ^                     ---- ^\n',
 '+   4. Complicated is better than complex.\n',
 '?           ++++ ^                     ^\n',
 '+   5. Flat is better than nested.\n']
```

作为单独的多行字符串显示出来则是这样：

```
>>> import sys
>>> sys.stdout.writelines(result)
```

(下页继续)

(续上页)

```
    1. Beautiful is better than ugly.
-   2. Explicit is better than implicit.
-   3. Simple is better than complex.
+   3.   Simple is better than complex.
?     ++
-   4. Complex is better than complicated.
?            ^                     ---- ^
+   4. Complicated is better than complex.
?          ++++ ^                      ^
+   5. Flat is better than nested.
```

7.4.5 difflib 的命令行接口

这个实例演示了如何使用 difflib 来创建一个类似于 `diff` 的工具。它同样包含在 Python 源码发布包中，文件名为 `Tools/scripts/diff.py`。

```
""" Command line interface to difflib.py providing diffs in four formats:

* ndiff:    lists every line and highlights interline changes.
* context:  highlights clusters of changes in a before/after format.
* unified:  highlights clusters of changes in an inline format.
* html:     generates side by side comparison with change highlights.

"""

import sys, os, time, difflib, optparse

def main():
     # Configure the option parser
    usage = "usage: %prog [options] fromfile tofile"
    parser = optparse.OptionParser(usage)
    parser.add_option("-c", action="store_true", default=False,
                      help='Produce a context format diff (default)')
    parser.add_option("-u", action="store_true", default=False,
                      help='Produce a unified format diff')
    hlp = 'Produce HTML side by side diff (can use -c and -l in conjunction)'
    parser.add_option("-m", action="store_true", default=False, help=hlp)
    parser.add_option("-n", action="store_true", default=False,
                      help='Produce a ndiff format diff')
    parser.add_option("-l", "--lines", type="int", default=3,
                      help='Set number of context lines (default 3)')
    (options, args) = parser.parse_args()

    if len(args) == 0:
        parser.print_help()
        sys.exit(1)
    if len(args) != 2:
        parser.error("need to specify both a fromfile and tofile")

    n = options.lines
    fromfile, tofile = args # as specified in the usage string

    # we're passing these as arguments to the diff function
    fromdate = time.ctime(os.stat(fromfile).st_mtime)
    todate = time.ctime(os.stat(tofile).st_mtime)
```

(下页继续)

(续上页)

```
    with open(fromfile, 'U') as f:
        fromlines = f.readlines()
    with open(tofile, 'U') as f:
        tolines = f.readlines()

    if options.u:
        diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile,
                                    fromdate, todate, n=n)
    elif options.n:
        diff = difflib.ndiff(fromlines, tolines)
    elif options.m:
        diff = difflib.HtmlDiff().make_file(fromlines, tolines, fromfile,
                                            tofile, context=options.c,
                                            numlines=n)
    else:
        diff = difflib.context_diff(fromlines, tolines, fromfile, tofile,
                                    fromdate, todate, n=n)

    # we're using writelines because diff is a generator
    sys.stdout.writelines(diff)

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

7.5 StringIO —Read and write strings as files

This module implements a file-like class, *StringIO*, that reads and writes a string buffer (also known as *memory files*). See the description of file objects for operations (section *File Objects*). (For standard strings, see *str* and *unicode*.)

class StringIO.**StringIO**([*buffer*])

When a *StringIO* object is created, it can be initialized to an existing string by passing the string to the constructor. If no string is given, the *StringIO* will start empty. In both cases, the initial file position starts at zero.

The *StringIO* object can accept either Unicode or 8-bit strings, but mixing the two may take some care. If both are used, 8-bit strings that cannot be interpreted as 7-bit ASCII (that use the 8th bit) will cause a *UnicodeError* to be raised when *getvalue()* is called.

The following methods of *StringIO* objects require special mention:

StringIO.**getvalue**()

Retrieve the entire contents of the "file" at any time before the *StringIO* object's *close()* method is called. See the note above for information about mixing Unicode and 8-bit strings; such mixing can cause this method to raise *UnicodeError*.

StringIO.**close**()

Free the memory buffer. Attempting to do further operations with a closed *StringIO* object will raise a *ValueError*.

Example usage:

```
import StringIO

output = StringIO.StringIO()
output.write('First line.\n')
```

(下页继续)

(续上页)

```
print >>output, 'Second line.'

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

7.6 cStringIO —Faster version of StringIO

The module *cStringIO* provides an interface similar to that of the *StringIO* module. Heavy use of *StringIO.StringIO* objects can be made more efficient by using the function *StringIO()* from this module instead.

cStringIO.**StringIO**([*s*])
Return a StringIO-like stream for reading or writing.

Since this is a factory function which returns objects of built-in types, there's no way to build your own version using subclassing. It's not possible to set attributes on it. Use the original *StringIO* module in those cases.

Unlike the *StringIO* module, this module is not able to accept Unicode strings that cannot be encoded as plain ASCII strings.

Another difference from the *StringIO* module is that calling *StringIO()* with a string parameter creates a read-only object. Unlike an object created without a string parameter, it does not have write methods. These objects are not generally visible. They turn up in tracebacks as StringI and StringO.

The following data objects are provided as well:

cStringIO.**InputType**
The type object of the objects created by calling *StringIO()* with a string parameter.

cStringIO.**OutputType**
The type object of the objects returned by calling *StringIO()* with no parameters.

There is a C API to the module as well; refer to the module source for more information.

Example usage:

```
import cStringIO

output = cStringIO.StringIO()
output.write('First line.\n')
print >>output, 'Second line.'

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

7.7 textwrap —文本自动换行与填充

2.3 新版功能.

源代码: Lib/textwrap.py

The *textwrap* module provides two convenience functions, *wrap()* and *fill()*, as well as *TextWrapper*, the class that does all the work, and a utility function *dedent()*. If you're just wrapping or filling one or two text strings, the convenience functions should be good enough; otherwise, you should use an instance of *TextWrapper* for efficiency.

textwrap.**wrap**(*text*[, *width*[, ...]])
对 *text* (字符串) 中的单独段落自动换行以使每行长度最多为 *width* 个字符。返回由输出行组成的列表，行尾不带换行符。

可选的关键字参数对应于*TextWrapper* 的实例属性，具体文档见下。*width* 默认为 70。

请参阅*TextWrapper.wrap()* 方法了解有关*wrap()* 行为的详细信息。

textwrap.**fill**(*text*[, *width*[, ...]])
对 *text* 中的单独段落自动换行，并返回一个包含被自动换行段落的单独字符串。*fill()* 是以下语句的快捷方式

```
"\n".join(wrap(text, ...))
```

特别要说明的是，*fill()* 接受与*wrap()* 完全相同的关键字参数。

Both *wrap()* and *fill()* work by creating a *TextWrapper* instance and calling a single method on it. That instance is not reused, so for applications that wrap/fill many text strings, it will be more efficient for you to create your own *TextWrapper* object.

文本最好在空白符位置自动换行，包括带连字符单词的连字符之后；长单词仅在必要时会被拆分，除非*TextWrapper.break_long_words* 被设为假值。

An additional utility function, *dedent()*, is provided to remove indentation from strings that have unwanted whitespace to the left of the text.

textwrap.**dedent**(*text*)
移除 *text* 中每一行的任何相同前缀空白符。

这可以用来清除三重引号字符串行左侧空格，而仍然在源码中显示为缩进格式。

Note that tabs and spaces are both treated as whitespace, but they are not equal: the lines `"  hello"` and `"\thello"` are considered to have no common leading whitespace. (This behaviour is new in Python 2.5; older versions of this module incorrectly expanded tabs before searching for common leading whitespace.)

只包含空白符的行会在输入时被忽略并在输出时被标准化为单个换行符。

例如

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
    hello
      world
    '''
    print repr(s)           # prints '    hello\n      world\n    '
    print repr(dedent(s))   # prints 'hello\n  world\n'
```

class textwrap.**TextWrapper**(...)

The *TextWrapper* constructor accepts a number of optional keyword arguments. Each argument corresponds to one instance attribute, so for example

```
wrapper = TextWrapper(initial_indent="* ")
```

就相当于

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

你可以多次重用相同的*TextWrapper* 对象，并且你也可以在使用期间通过直接向实例属性赋值来修改它的任何选项。

TextWrapper 的实例属性（以及构造器的关键字参数）如下所示：

width

(默认: 70) 自动换行的最大行长度。只要输入文本中没有长于*width* 的单个单词，*TextWrapper* 就能保证没有长于*width* 个字符的输出行。

expand_tabs

(默认: True) 如果为真值，则 *text* 中所有的制表符将使用 *text* 的 expandtabs() 方法扩展为空格符。

replace_whitespace

(default: True) 如果为真值，在制表符扩展之后、自动换行之前，*wrap()* 方法将把每个空白字符都替换为单个空格。会被替换的空白字符如下：制表，换行，垂直制表，进纸和回车 ('\t\n\v\f\r')。

注解： 如果*expand_tabs* 为假值且*replace_whitespace* 为真值，每个制表符将被替换为单个空格，这与制表符扩展是 不一样的。

注解： 如果*replace_whitespace* 为假值，在一行的中间有可能出现换行符并导致怪异的输出。因此，文本应当（使用*str.splitlines()* 或类似方法）拆分为段落并分别进行自动换行。

---

drop_whitespace

(默认: True) 如果为真值，每一行开头和末尾的空白字符（在包装之后、缩进之前）会被丢弃。但是段落开头的空白字符如果后面不带任何非空白字符则不会被丢弃。如果被丢弃的空白字符占据了一个整行，则该整行将被丢弃。

2.6 新版功能: Whitespace was always dropped in earlier versions.

initial_indent

(默认: '') 将被添加到被自动换行输出内容的第一行的字符串。其长度会被计入第一行的长度。空字符串不会被缩进。

subsequent_indent

(default: '') 将被添加到被自动换行输出内容除第一行外的所有行的字符串。其长度会被计入除行一行外的所有行的长度。

fix_sentence_endings

(默认: False) 如果为真值，*TextWrapper* 将尝试检测句子结尾并确保句子间总是以恰好两个空格符分隔。对于使用等宽字体的文本来说通常都需要这样。但是，句子检测算法并不完美：它假定句子结尾是一个小写字母加字符 '.', '!' 或 '?' 中的一个，并可能带有字符 '"' 或 "'"，最后以一个空格结束。此算法的问题之一是它无法区分以下文本中的 “Dr.”

```
[...] Dr. Frankenstein's monster [...]
```

和以下文本中的 “Spot.”

```
[...] See Spot. See Spot run [...]
```

fix_sentence_endings 默认为假值。

Since the sentence detection algorithm relies on `string.lowercase` for the definition of “lowercase letter,” and a convention of using two spaces after a period to separate sentences on the same line, it is specific to English-language texts.

break_long_words
(默认: `True`) 如果为真值，则长度超过*width* 的单词将被分开以保证行的长度不会超过*width*。如果为假值，超长单词不会被分开，因而某些行的长度可能会超过*width*。(超长单词将被单独作为一行，以尽量减少超出*width* 的情况。)

break_on_hyphens
(默认: `True`) 如果为真值，将根据英语的惯例首选在空白符和复合词的连字符之后自动换行。如果为假值，则只有空白符会被视为合适的潜在断行位置，但如果你确实不希望出现分开的单词则你必须将*break_long_words* 设为假值。之前版本的默认行为总是允许分开带有连字符的单词。

2.6 新版功能.

TextWrapper also provides two public methods, analogous to the module-level convenience functions:

wrap(*text*)
对 *text* (字符串) 中的单独段落自动换行以使每行长度最多为*width* 个字符。所有自动换行选项均获取自*TextWrapper* 实例的实例属性。返回由输出行组成的列表，行尾不带换行符。如果自动换行输出结果没有任何内容，则返回空列表。

fill(*text*)
对 *text* 中的单独段落自动换行并返回包含被自动换行段落的单独字符串。

7.8 codecs —编解码器注册和相关基类

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry which manages the codec and error handling lookup process.

It defines the following functions:

`codecs.`**encode**(*obj*[, *encoding*[, *errors*]])
Encodes *obj* using the codec registered for *encoding*. The default encoding is `'ascii'`.

可以给定 *Errors* 以设置所需要的错误处理方案。默认的错误处理方案 `'strict'` 表示编码错误将引发*ValueError* (或更特定编解码器相关的子类，例如*UnicodeEncodeError*)。请参阅编解码器基类了解有关编解码器错误处理的更多信息。

2.4 新版功能.

`codecs.`**decode**(*obj*[, *encoding*[, *errors*]])
Decodes *obj* using the codec registered for *encoding*. The default encoding is `'ascii'`.

可以给定 *Errors* 以设置所需要的错误处理方案。默认的错误处理方案 `'strict'` 表示编码错误将引发*ValueError* (或更特定编解码器相关的子类，例如*UnicodeDecodeError*)。请参阅编解码器基类了解有关编解码器错误处理的更多信息。

2.4 新版功能.

codecs.**register**(*search_function*)

Register a codec search function. Search functions are expected to take one argument, the encoding name in all lower case letters, and return a CodecInfo object having the following attributes:

- name The name of the encoding;
- encode The stateless encoding function;
- decode The stateless decoding function;
- incrementalencoder An incremental encoder class or factory function;
- incrementaldecoder An incremental decoder class or factory function;
- streamwriter A stream writer class or factory function;
- streamreader A stream reader class or factory function.

The various functions or classes take the following arguments:

encode and *decode*: These must be functions or methods which have the same interface as the *encode()*/*decode()* methods of Codec instances (see *Codec Interface*). The functions/methods are expected to work in a stateless mode.

incrementalencoder and *incrementaldecoder*: These have to be factory functions providing the following interface:

```
factory(errors='strict')
```

The factory functions must return objects providing the interfaces defined by the base classes *IncrementalEncoder* and *IncrementalDecoder*, respectively. Incremental codecs can maintain state.

streamreader and *streamwriter*: These have to be factory functions providing the following interface:

```
factory(stream, errors='strict')
```

The factory functions must return objects providing the interfaces defined by the base classes *StreamReader* and *StreamWriter*, respectively. Stream codecs can maintain state.

Possible values for errors are

- 'strict': raise an exception in case of an encoding error
- 'replace': replace malformed data with a suitable replacement marker, such as '?' or '\ufffd'
- 'ignore': ignore malformed data and continue without further notice
- 'xmlcharrefreplace': replace with the appropriate XML character reference (for encoding only)
- 'backslashreplace': replace with backslashed escape sequences (for encoding only)

as well as any other error handling name defined via *register_error()*.

In case a search function cannot find a given encoding, it should return None.

codecs.**lookup**(*encoding*)

Looks up the codec info in the Python codec registry and returns a CodecInfo object as defined above.

首先将会在注册表缓存中查找编码，如果未找到，则会扫描注册的搜索函数列表。如果没有找到 CodecInfo 对象，则将引发*LookupError*。否则，CodecInfo 对象将被存入缓存并返回给调用者。

To simplify access to the various codecs, the module provides these additional functions which use *lookup()* for the codec lookup:

codecs.**getencoder**(*encoding*)

查找给定编码的编解码器并返回其编码器函数。

在编码无法找到时将引发*LookupError*。

codecs.**getdecoder**(*encoding*)
查找给定编码的编解码器并返回其解码器函数。

在编码无法找到时将引发*LookupError*。

codecs.**getincrementalencoder**(*encoding*)
查找给定编码的编解码器并返回其增量式编码器类或工厂函数。

在编码无法找到或编解码器不支持增量式编码器时将引发*LookupError*。

2.5 新版功能.

codecs.**getincrementaldecoder**(*encoding*)
查找给定编码的编解码器并返回其增量式解码器类或工厂函数。

在编码无法找到或编解码器不支持增量式解码器时将引发*LookupError*。

2.5 新版功能.

codecs.**getreader**(*encoding*)
Look up the codec for the given encoding and return its StreamReader class or factory function.

在编码无法找到时将引发*LookupError*。

codecs.**getwriter**(*encoding*)
Look up the codec for the given encoding and return its StreamWriter class or factory function.

在编码无法找到时将引发*LookupError*。

codecs.**register_error**(*name*, *error_handler*)
Register the error handling function *error_handler* under the name *name*. *error_handler* will be called during encoding and decoding in case of an error, when *name* is specified as the errors parameter.

For encoding *error_handler* will be called with a *UnicodeEncodeError* instance, which contains information about the location of the error. The error handler must either raise this or a different exception or return a tuple with a replacement for the unencodable part of the input and a position where encoding should continue. The encoder will encode the replacement and continue encoding the original input at the specified position. Negative position values will be treated as being relative to the end of the input string. If the resulting position is out of bound an *IndexError* will be raised.

Decoding and translating works similar, except *UnicodeDecodeError* or *UnicodeTranslateError* will be passed to the handler and that the replacement from the error handler will be put into the output directly.

codecs.**lookup_error**(*name*)
返回之前在名称 *name* 之下注册的错误处理方案。

在处理方案无法找到时将引发*LookupError*。

codecs.**strict_errors**(*exception*)
Implements the `strict` error handling: each encoding or decoding error raises a *UnicodeError*.

codecs.**replace_errors**(*exception*)
Implements the `replace` error handling: malformed data is replaced with a suitable replacement character such as `'?'` in bytestrings and `'\ufffd'` in Unicode strings.

codecs.**ignore_errors**(*exception*)
Implements the `ignore` error handling: malformed data is ignored and encoding or decoding is continued without further notice.

codecs.**xmlcharrefreplace_errors**(*exception*)
Implements the `xmlcharrefreplace` error handling (for encoding only): the unencodable character is replaced by an appropriate XML character reference.

codecs.**backslashreplace_errors**(*exception*)
Implements the backslashreplace error handling (for encoding only): the unencodable character is replaced by a backslashed escape sequence.

To simplify working with encoded files or stream, the module also defines these utility functions:

codecs.**open**(*filename*, *mode*[, *encoding*[, *errors*[, *buffering*]]])
Open an encoded file using the given *mode* and return a wrapped version providing transparent encoding/decoding. The default file mode is 'r' meaning to open the file in read mode.

注解: The wrapped version will only accept the object format defined by the codecs, i.e. Unicode objects for most built-in codecs. Output is also codec-dependent and will usually be Unicode as well.

注解: Files are always opened in binary mode, even if no binary mode was specified. This is done to avoid data loss due to encodings using 8-bit values. This means that no automatic conversion of '\n' is done on reading and writing.

---

encoding specifies the encoding which is to be used for the file.

可以指定 *errors* 来定义错误处理方案。默认值 'strict' 表示在出现编码错误时引发*ValueError*。

buffering 的含义与内置*open()* 函数中的相同。默认为行缓冲。

codecs.**EncodedFile**(*file*, *input*[, *output*[, *errors*]])
Return a wrapped version of file which provides transparent encoding translation.

Strings written to the wrapped file are interpreted according to the given *input* encoding and then written to the original file as strings using the *output* encoding. The intermediate encoding will usually be Unicode but depends on the specified codecs.

If *output* is not given, it defaults to *input*.

可以指定 *errors* 来定义错误处理方案。默认值 'strict' 表示在出现编码错误时引发*ValueError*。

codecs.**iterencode**(*iterable*, *encoding*[, *errors*])
Uses an incremental encoder to iteratively encode the input provided by *iterable*. This function is a *generator*. *errors* (as well as any other keyword argument) is passed through to the incremental encoder.

2.5 新版功能.

codecs.**iterdecode**(*iterable*, *encoding*[, *errors*])
Uses an incremental decoder to iteratively decode the input provided by *iterable*. This function is a *generator*. *errors* (as well as any other keyword argument) is passed through to the incremental decoder.

2.5 新版功能.

本模块还提供了以下常量，适用于读取和写入依赖于平台的文件：

codecs.**BOM**
codecs.**BOM_BE**
codecs.**BOM_LE**
codecs.**BOM_UTF8**
codecs.**BOM_UTF16**
codecs.**BOM_UTF16_BE**
codecs.**BOM_UTF16_LE**
codecs.**BOM_UTF32**
codecs.**BOM_UTF32_BE**

codecs.**BOM_UTF32_LE**

These constants define various encodings of the Unicode byte order mark (BOM) used in UTF-16 and UTF-32 data streams to indicate the byte order used in the stream or file and in UTF-8 as a Unicode signature. *BOM_UTF16* is either *BOM_UTF16_BE* or *BOM_UTF16_LE* depending on the platform's native byte order, *BOM* is an alias for *BOM_UTF16*, *BOM_LE* for *BOM_UTF16_LE* and *BOM_BE* for *BOM_UTF16_BE*. The others represent the BOM in UTF-8 and UTF-32 encodings.

7.8.1 编解码器基类

The *codecs* module defines a set of base classes which define the interface and can also be used to easily write your own codecs for use in Python.

Each codec has to define four interfaces to make it usable as codec in Python: stateless encoder, stateless decoder, stream reader and stream writer. The stream reader and writers typically reuse the stateless encoder/decoder to implement the file protocols.

The Codec class defines the interface for stateless encoders/decoders.

To simplify and standardize error handling, the *encode()* and *decode()* methods may implement different error handling schemes by providing the *errors* string argument. The following string values are defined and implemented by all standard Python codecs:

值	含义
'strict'	Raise *UnicodeError* (or a subclass); this is the default.
'ignore'	Ignore the character and continue with the next.
'replace'	Replace with a suitable replacement character; Python will use the official U+FFFD REPLACEMENT CHARACTER for the built-in Unicode codecs on decoding and '?' on encoding.
'xmlcharrefreplace'	Replace with the appropriate XML character reference (only for encoding).
'backslashreplace'	Replace with backslashed escape sequences (only for encoding).

The set of allowed values can be extended via *register_error()*.

Codec Objects

The Codec class defines these methods which also define the function interfaces of the stateless encoder and decoder:

Codec.**encode**(*input*[, *errors*])

Encodes the object *input* and returns a tuple (output object, length consumed). While codecs are not restricted to use with Unicode, in a Unicode context, encoding converts a Unicode object to a plain string using a particular character set encoding (e.g., cp1252 or iso-8859-1).

errors defines the error handling to apply. It defaults to 'strict' handling.

此方法不一定会在 Codec 实例中保存状态。可使用必须保存状态的*StreamWriter* 作为编解码器以便高效地进行编码。

编码器必须能够处理零长度的输入并在此情况下返回输出对象类型的空对象。

Codec.**decode**(*input*[, *errors*])

Decodes the object *input* and returns a tuple (output object, length consumed). In a Unicode context, decoding converts a plain string encoded using a particular character set encoding to a Unicode object.

input must be an object which provides the bf_getreadbuf buffer slot. Python strings, buffer objects and memory mapped files are examples of objects providing this slot.

errors defines the error handling to apply. It defaults to 'strict' handling.

此方法不一定会在 Codec 实例中保存状态。可使用必须保存状态的*StreamReader* 作为编解码器以便高效地进行解码。

解码器必须能够处理零长度的输入并在此情况下返回输出对象类型的空对象。

IncrementalEncoder 和*IncrementalDecoder* 类提供了增量式编码和解码的基本接口。对输入的编码/解码不是通过对无状态编码器/解码器的一次调用，而是通过对增量式编码器/解码器的*encode()*/*decode()* 方法的多次调用。增量式编码器/解码器会在方法调用期间跟踪编码/解码过程。

调用*encode()*/*decode()* 方法后的全部输出相当于将所有通过无状态编码器/解码器进行编码/解码的单个输入连接在一起所得到的输出。

IncrementalEncoder 对象

2.5 新版功能.

IncrementalEncoder 类用来对一个输入进行分步编码。它定义了以下方法，每个增量式编码器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**IncrementalEncoder**([*errors*])

IncrementalEncoder 实例的构造器。

所有增量式编码器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

The *IncrementalEncoder* may implement different error handling schemes by providing the *errors* keyword argument. These parameters are predefined:

- `'strict'` Raise *ValueError* (or a subclass); this is the default.
- `'ignore'` Ignore the character and continue with the next.
- `'replace'` Replace with a suitable replacement character
- `'xmlcharrefreplace'` Replace with the appropriate XML character reference
- `'backslashreplace'` Replace with backslashed escape sequences.

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在*IncrementalEncoder* 对象的生命期内在不同的错误处理策略之间进行切换。

errors 参数所允许的值集合可以使用*register_error()* 来扩展。

encode(*object*[, *final*])

编码 *object* (会将编码器的当前状态纳入考虑) 并返回已编码的结果对象。如果这是对*encode()* 的最终调用则 *final* 必须为真值（默认为假值）。

reset()

Reset the encoder to the initial state.

IncrementalDecoder 对象

IncrementalDecoder 类用来对一个输入进行分步解码。它定义了以下方法，每个增量式解码器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**IncrementalDecoder**([*errors*])

IncrementalDecoder 实例的构造器。

所有增量式解码器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

The *IncrementalDecoder* may implement different error handling schemes by providing the *errors* keyword argument. These parameters are predefined:

- `'strict'` Raise *ValueError* (or a subclass); this is the default.
- `'ignore'` Ignore the character and continue with the next.
- `'replace'` Replace with a suitable replacement character.

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在*IncrementalDecoder* 对象的生命期内在不同的错误处理策略之间进行切换。

errors 参数所允许的值集合可以使用*register_error()* 来扩展。

decode(*object*[, *final*])
解码 *object* (会将解码器的当前状态纳入考虑) 并返回已解码的结果对象。如果这是对*decode()* 的最终调用则 *final* 必须为真值（默认为假值）。如果 *final* 为真值则解码器必须对输入进行完全解码并且必须刷新所有缓冲区。如果这无法做到（例如由于在输入结束时字节串序列不完整）则它必须像在无状态的情况下那样初始化错误处理（这可能引发一个异常）。

reset()
将解码器重置为初始状态。

StreamWriter 和*StreamReader* 类提供了一些泛用工作接口，可被用来非常方便地实现新的编码格式子模块。请参阅 `encodings.utf_8` 中的示例了解如何做到这一点。

StreamWriter 对象

StreamWriter 类是 `Codec` 的子类，它定义了以下方法，每个流式写入器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**StreamWriter**(*stream*[, *errors*])
StreamWriter 实例的构造器。

所有流式写入器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

stream must be a file-like object open for writing binary data.

The *StreamWriter* may implement different error handling schemes by providing the *errors* keyword argument. These parameters are predefined:

- `'strict'` Raise *ValueError* (or a subclass); this is the default.
- `'ignore'` Ignore the character and continue with the next.
- `'replace'` Replace with a suitable replacement character
- `'xmlcharrefreplace'` Replace with the appropriate XML character reference
- `'backslashreplace'` Replace with backslashed escape sequences.

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在*StreamWriter* 对象的生命期内在不同的错误处理策略之间进行切换。

errors 参数所允许的值集合可以使用*register_error()* 来扩展。

write(*object*)
将编码后的对象内容写入到流。

writelines(*list*)
Writes the concatenated list of strings to the stream (possibly by reusing the *write()* method).

reset()
刷新并重置用于保持状态的编解码器缓冲区。

调用此方法应当确保在干净的状态下放入输出数据，以允许直接添加新的干净数据而无须重新扫描整个流来恢复状态。

除了上述的方法，*StreamWriter* 还必须继承来自下层流的所有其他方法和属性。

StreamReader 对象

StreamReader 类是 Codec 的子类，它定义了以下方法，每个流式读取器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**StreamReader**(*stream*[, *errors*])
StreamReader 实例的构造器。

所有流式读取器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

stream must be a file-like object open for reading (binary) data.

The *StreamReader* may implement different error handling schemes by providing the *errors* keyword argument. These parameters are defined:

- 'strict' Raise *ValueError* (or a subclass); this is the default.
- 'ignore' Ignore the character and continue with the next.
- 'replace' Replace with a suitable replacement character.

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在*StreamReader* 对象的生命期内在不同的错误处理策略之间进行切换。

errors 参数所允许的值集合可以使用*register_error()* 来扩展。

read([*size*[, *chars*[, *firstline*]]])
解码来自流的数据并返回结果对象。

chars indicates the number of characters to read from the stream. *read()* will never return more than *chars* characters, but it might return less, if there are not enough characters available.

size indicates the approximate maximum number of bytes to read from the stream for decoding purposes. The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. *size* is intended to prevent having to decode huge files in one step.

firstline indicates that it would be sufficient to only return the first line, if there are decoding errors on later lines.

此方法应当使用“贪婪”读取策略，这意味着它应当在编码格式定义和给定大小所允许的情况下尽可能多地读取数据，例如，如果在流上存在可选的编码结束或状态标记，这些内容也应当被读取。

在 2.4 版更改: *chars* argument added.

在 2.4.2 版更改: *firstline* argument added.

readline([*size*[, *keepends*]])
从输入流读取一行并返回解码后的数据。

如果给定了 *size*，则将其作为 size 参数传递给流的*read()* 方法。

如果 *keepends* 为假值，则行结束符将从返回的行中去除。

在 2.4 版更改: *keepends* argument added.

readlines([*sizehint*[, *keepends*]])
从输入流读取所有行并将其作为一个行列表返回。

Line-endings are implemented using the codec's decoder method and are included in the list entries if *keepends* is true.

如果给定了 *sizehint*，则将其作为 *size* 参数传递给流的`read()` 方法。

reset()
重置用于保持状态的编解码器缓冲区。

Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.

除了上述的方法，`StreamReader` 还必须继承来自下层流的所有其他方法和属性。

The next two base classes are included for convenience. They are not needed by the codec registry, but may provide useful in practice.

StreamReaderWriter 对象

The `StreamReaderWriter` allows wrapping streams which work in both read and write modes.

其设计使得开发者可以使用`lookup()` 函数所返回的工厂函数来构造实例。

class codecs.**StreamReaderWriter**(*stream*, *Reader*, *Writer*, *errors*)
创建一个`StreamReaderWriter` 实例。*stream* 必须为一个文件类对象。*Reader* 和 *Writer* 必须为分别提供了`StreamReader` 和`StreamWriter` 接口的工厂函数或类。错误处理通过与流式读取器和写入器所定义的相同方式来完成。

`StreamReaderWriter` 实例定义了`StreamReader` 和`StreamWriter` 类的组合接口。它们还继承了来自下层流的所有其他方法和属性。

StreamRecoder 对象

The `StreamRecoder` provide a frontend - backend view of encoding data which is sometimes useful when dealing with different encoding environments.

其设计使得开发者可以使用`lookup()` 函数所返回的工厂函数来构造实例。

class codecs.**StreamRecoder**(*stream*, *encode*, *decode*, *Reader*, *Writer*, *errors*)
Creates a `StreamRecoder` instance which implements a two-way conversion: *encode* and *decode* work on the frontend (the input to `read()` and output of `write()`) while *Reader* and *Writer* work on the backend (reading and writing to the stream).

You can use these objects to do transparent direct recodings from e.g. Latin-1 to UTF-8 and back.

stream must be a file-like object.

encode, *decode* must adhere to the `Codec` interface. *Reader*, *Writer* must be factory functions or classes providing objects of the `StreamReader` and `StreamWriter` interface respectively.

encode and *decode* are needed for the frontend translation, *Reader* and *Writer* for the backend translation. The intermediate format used is determined by the two sets of codecs, e.g. the Unicode codecs will use Unicode as the intermediate encoding.

错误处理通过与流式读取器和写入器所定义的相同方式来完成。

`StreamRecoder` 实例定义了`StreamReader` 和`StreamWriter` 类的组合接口。它们还继承了来自下层流的所有其他方法和属性。

7.8.2 编码格式与 Unicode

Unicode strings are stored internally as sequences of code points (to be precise as `Py_UNICODE` arrays). Depending on the way Python is compiled (either via `--enable-unicode=ucs2` or `--enable-unicode=ucs4`, with the former being the default) `Py_UNICODE` is either a 16-bit or 32-bit data type. Once a Unicode object is used outside of CPU and memory, CPU endianness and how these arrays are stored as bytes become an issue. Transforming a unicode object into a sequence of bytes is called encoding and recreating the unicode object from the sequence of bytes is known as decoding. There are many different methods for how this transformation can be done (these methods are also called encodings). The simplest method is to map the code points 0–255 to the bytes `0x0–0xff`. This means that a unicode object that contains code points above `U+00FF` can't be encoded with this method (which is called `'latin-1'` or `'iso-8859-1'`). `unicode.encode()` will raise a *UnicodeEncodeError* that looks like this: `UnicodeEncodeError: 'latin-1' codec can't encode character u'\u1234' in position 3: ordinal not in range(256)`.

There's another group of encodings (the so called charmap encodings) that choose a different subset of all unicode code points and how these code points are mapped to the bytes `0x0–0xff`. To see how this is done simply open e.g. `encodings/cp1252.py` (which is an encoding that is used primarily on Windows). There's a string constant with 256 characters that shows you which character is mapped to which byte value.

All of these encodings can only encode 256 of the 1114112 code points defined in unicode. A simple and straightforward way that can store each Unicode code point, is to store each code point as four consecutive bytes. There are two possibilities: store the bytes in big endian or in little endian order. These two encodings are called `UTF-32-BE` and `UTF-32-LE` respectively. Their disadvantage is that if e.g. you use `UTF-32-BE` on a little endian machine you will always have to swap bytes on encoding and decoding. `UTF-32` avoids this problem: bytes will always be in natural endianness. When these bytes are read by a CPU with a different endianness, then bytes have to be swapped though. To be able to detect the endianness of a `UTF-16` or `UTF-32` byte sequence, there's the so called BOM ("Byte Order Mark"). This is the Unicode character `U+FEFF`. This character can be prepended to every `UTF-16` or `UTF-32` byte sequence. The byte swapped version of this character (`0xFFFE`) is an illegal character that may not appear in a Unicode text. So when the first character in an `UTF-16` or `UTF-32` byte sequence appears to be a `U+FFFE` the bytes have to be swapped on decoding. Unfortunately the character `U+FEFF` had a second purpose as a `ZERO WIDTH NO-BREAK SPACE`: a character that has no width and doesn't allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using `U+FEFF` as a `ZERO WIDTH NO-BREAK SPACE` has been deprecated (with `U+2060` (`WORD JOINER`) assuming this role). Nevertheless Unicode software still must be able to handle `U+FEFF` in both roles: as a BOM it's a device to determine the storage layout of the encoded bytes, and vanishes once the byte sequence has been decoded into a Unicode string; as a `ZERO WIDTH NO-BREAK SPACE` it's a normal character that will be decoded like any other.

还有另一种编码格式能够对所有的 Unicode 字符进行编码：UTF-8。UTF-8 是一种 8 位编码，这意味着在 UTF-8 中没有字节顺序问题。UTF-8 字节序列中的每个字节由两部分组成：标志位（最重要的位）和内容位。标志位是由零至四个值为 `1` 的二进制位加一个值为 `0` 的二进制位构成的序列。Unicode 字符会按以下形式进行编码（其中 x 为内容位，当拼接为一体时将给出对应的 Unicode 字符）：

范围	编码
`U-00000000` …`U-0000007F`	0xxxxxxx
`U-00000080` …`U-000007FF`	110xxxxx 10xxxxxx
`U-00000800` …`U-0000FFFF`	1110xxxx 10xxxxxx 10xxxxxx
`U-00010000` …`U-0010FFFF`	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Unicode 字符最不重要的一个位就是最右侧的二进制位 x。

As UTF-8 is an 8-bit encoding no BOM is required and any `U+FEFF` character in the decoded Unicode string (even if it's the first character) is treated as a `ZERO WIDTH NO-BREAK SPACE`.

Without external information it's impossible to reliably determine which encoding was used for encoding a Unicode string. Each charmap encoding can decode any random byte sequence. However that's not possible with UTF-8, as UTF-8 byte sequences have a structure that doesn't allow arbitrary byte sequences. To increase the reliability with which

a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python 2.5 calls `"utf-8-sig"`) for its Notepad program: Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence: `0xef`, `0xbb`, `0xbf`) is written. As it's rather improbable that any charmap encoded file starts with these byte values (which would e.g. map to

LATIN SMALL LETTER I WITH DIAERESIS
RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
INVERTED QUESTION MARK

in iso-8859-1), this increases the probability that a `utf-8-sig` encoding can be correctly guessed from the byte sequence. So here the BOM is not used to be able to determine the byte order used for generating the byte sequence, but as a signature that helps in guessing the encoding. On encoding the utf-8-sig codec will write `0xef`, `0xbb`, `0xbf` as the first three bytes to the file. On decoding `utf-8-sig` will skip those three bytes if they appear as the first three bytes in the file. In UTF-8, the use of the BOM is discouraged and should generally be avoided.

7.8.3 标准编码

Python 自带了许多内置的编解码器，它们的实现或者是通过 C 函数，或者是通过映射表。以下表格是按名称排序的编解码器列表，并提供了一些常见别名以及编码格式通常针对的语言。别名和语言列表都不是详尽无遗的。请注意仅有大小写区别或使用连字符替代下划线的拼写形式也都是有效的别名；因此，`'utf-8'` 是 `'utf_8'` 编解码器的有效别名。

许多字符集都支持相同的语言。它们在个别字符（例如是否支持 EURO SIGN 等）以及给字符所分配的码位方面存在差异。特别是对于欧洲语言来说，通常存在以下几种变体：

- 某个 ISO 8859 编码集
- 某个 Microsoft Windows 编码页，通常是派生自某个 8859 编码集，但会用附加的图形字符来替换控制字符。
- 某个 IBM EBCDIC 编码页
- 某个 IBM PC 编码页，通常会兼容 ASCII

编码	别名	语言
ascii	646, us-ascii	英语
big5	big5-tw, csbig5	繁体中文
big5hkscs	big5-hkscs, hkscs	繁体中文
cp037	IBM037, IBM039	英语
cp424	EBCDIC-CP-HE, IBM424	希伯来语
cp437	437, IBM437	英语
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	西欧
cp720		阿拉伯语
cp737		希腊语
cp775	IBM775	波罗的海语言
cp850	850, IBM850	西欧
cp852	852, IBM852	中欧和东欧
cp855	855, IBM855	保加利亚语，白俄罗斯语，马其顿语，俄语，塞尔维亚语
cp856		希伯来语
cp857	857, IBM857	土耳其语
cp858	858, IBM858	西欧
cp860	860, IBM860	葡萄牙语

下页继续

表 3 – 续上页

编码	别名	语言
cp861	861, CP-IS, IBM861	冰岛语
cp862	862, IBM862	希伯来语
cp863	863, IBM863	加拿大语
cp864	IBM864	阿拉伯语
cp865	865, IBM865	丹麦语/挪威语
cp866	866, IBM866	俄语
cp869	869, CP-GR, IBM869	希腊语
cp874		泰语
cp875		希腊语
cp932	932, ms932, mskanji, ms-kanji	日语
cp949	949, ms949, uhc	韩语
cp950	950, ms950	繁体中文
cp1006		乌尔都语
cp1026	ibm1026	土耳其语
cp1140	ibm1140	西欧
cp1250	windows-1250	中欧和东欧
cp1251	windows-1251	保加利亚语，白俄罗斯语，马其顿语，俄语，塞尔维亚语
cp1252	windows-1252	西欧
cp1253	windows-1253	希腊语
cp1254	windows-1254	土耳其语
cp1255	windows-1255	希伯来语
cp1256	windows-1256	阿拉伯语
cp1257	windows-1257	波罗的海语言
cp1258	windows-1258	越南语
euc_jp	eucjp, ujis, u-jis	日语
euc_jis_2004	jisx0213, eucjis2004	日语
euc_jisx0213	eucjisx0213	日语
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	韩语
gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso- ir-58	简体中文
gbk	936, cp936, ms936	统一汉语
gb18030	gb18030-2000	统一汉语
hz	hzgb, hz-gb, hz-gb-2312	简体中文
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	日语
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	日语
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	日语，韩语，简体中文，西欧，希腊语
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	日语
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	日语
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	日语
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	韩语
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	West Europe
iso8859_2	iso-8859-2, latin2, L2	中欧和东欧

下页继续

表 3 – 续上页

编码	别名	语言
iso8859_3	iso-8859-3, latin3, L3	世界语，马耳他语
iso8859_4	iso-8859-4, latin4, L4	波罗的海语言
iso8859_5	iso-8859-5, cyrillic	保加利亚语，白俄罗斯语，马其顿语，俄语，塞尔维亚语
iso8859_6	iso-8859-6, arabic	阿拉伯语
iso8859_7	iso-8859-7, greek, greek8	希腊语
iso8859_8	iso-8859-8, hebrew	希伯来语
iso8859_9	iso-8859-9, latin5, L5	土耳其语
iso8859_10	iso-8859-10, latin6, L6	北欧语言
iso8859_11	iso-8859-11, thai	泰语
iso8859_13	iso-8859-13, latin7, L7	波罗的海语言
iso8859_14	iso-8859-14, latin8, L8	凯尔特语
iso8859_15	iso-8859-15, latin9, L9	西欧
iso8859_16	iso-8859-16, latin10, L10	东南欧
johab	cp1361, ms1361	韩语
koi8_r		俄语
koi8_u		乌克兰语
mac_cyrillic	maccyrillic	保加利亚语，白俄罗斯语，马其顿语，俄语，塞尔维亚语
mac_greek	macgreek	希腊语
mac_iceland	maciceland	冰岛语
mac_latin2	maclatin2, maccentraleurope	中欧和东欧
mac_roman	macroman	西欧
mac_turkish	macturkish	土耳其语
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	哈萨克语
shift_jis	csshiftjis, shiftjis, sjis, s_jis	日语
shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	日语
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	日语
utf_32	U32, utf32	所有语言
utf_32_be	UTF-32BE	所有语言
utf_32_le	UTF-32LE	所有语言
utf_16	U16, utf16	所有语言
utf_16_be	UTF-16BE	all languages (BMP only)
utf_16_le	UTF-16LE	all languages (BMP only)
utf_7	U7, unicode-1-1-utf-7	所有语言
utf_8	U8, UTF, utf8	所有语言
utf_8_sig		所有语言

7.8.4 Python 专属的编码格式

A number of predefined codecs are specific to Python, so their codec names have no meaning outside Python. These are listed in the tables below based on the expected input and output types (note that while text encodings are the most common use case for codecs, the underlying codec infrastructure supports arbitrary data transforms rather than just text encodings). For asymmetric codecs, the stated purpose describes the encoding direction.

The following codecs provide unicode-to-str encoding[1] and str-to-unicode decoding[2], similar to the Unicode text encod-

[1] str objects are also accepted as input in place of unicode objects. They are implicitly converted to unicode by decoding them using the default encoding. If this conversion fails, it may lead to encoding operations raising *UnicodeDecodeError*.

[2] unicode objects are also accepted as input in place of str objects. They are implicitly converted to str by encoding them using the default encoding.

ings.

编码	别名	Purpose
idna		Implements **RFC 3490**, see also *encodings.idna*
mbcs	dbcs	Windows only: Encode operand according to the ANSI codepage (CP_ACP)
palmos		Encoding of PalmOS 3.5
punycode		Implements **RFC 3492**
raw_unicode_escape		Produce a string that is suitable as raw Unicode literal in Python source code
rot_13	rot13	Returns the Caesar-cypher encryption of the operand
undefined		Raise an exception for all conversions. Can be used as the system encoding if no automatic *coercion* between byte and Unicode strings is desired.
unicode_escape		Produce a string that is suitable as Unicode literal in Python source code
unicode_internal		Return the internal representation of the operand

2.3 新版功能: The `idna` and `punycode` encodings.

The following codecs provide str-to-str encoding and decoding[2].

编码	别名	Purpose	Encoder/decoder
base64_codec	base64, base-64	Convert operand to multiline MIME base64 (the result always includes a trailing `'\n'`)	*base64.encodestring()*, *base64.decodestring()*
bz2_codec	bz2	Compress the operand using bz2	*bz2.compress()*, *bz2.decompress()*
hex_codec	hex	Convert operand to hexadecimal representation, with two digits per byte	*binascii.b2a_hex()*, *binascii.a2b_hex()*
quopri_codec	quopri, quoted-printable, quotedprintable	Convert operand to MIME quoted printable	*quopri.encode()* with `quotetabs=True`, *quopri.decode()*
string_escape		Produce a string that is suitable as string literal in Python source code	
uu_codec	uu	Convert the operand using uuencode	*uu.encode()*, *uu.decode()*
zlib_codec	zip, zlib	Compress the operand using gzip	*zlib.compress()*, *zlib.decompress()*

7.8.5 `encodings.idna` —应用程序中的国际化域名

2.3 新版功能.

此模块实现了 **RFC 3490** (应用程序中的国际化域名) 和 **RFC 3492** (Nameprep: 用于国际化域名 (IDN) 的 Stringprep 配置文件)。它是在 `punycode` 编码格式和*stringprep* 的基础上构建的。

这些 RFC 共同定义了一个在域名中支持非 ASCII 字符的协议。一个包含非 ASCII 字符的域名 (例如 `www.Alliancefrançaise.nu`) 会被转换为兼容 ASCII 的编码格式 (简称 ACE，例如 `www.xn--alliancefranaise-npb.nu`)。随后此域名的 ACE 形式可以用于所有由于特定协议而不允许使用任意字符的场合，例如 DNS 查询，HTTP *Host* 字段等等。此转换是在应用中进行的；如有可能将对用户可见：应用应当透明地将 Unicode 域名标签转换为线上的 IDNA，并在 ACE 标签被呈现给用户之前将其转换回 Unicode。

If this conversion fails, it may lead to decoding operations raising *UnicodeEncodeError*.

Python supports this conversion in several ways: the `idna` codec performs conversion between Unicode and ACE, separating an input string into labels based on the separator characters defined in section 3.1 (1) of **RFC 3490** and converting each label to ACE as required, and conversely separating an input byte string into labels based on the `.` separator and converting any ACE labels found into unicode. Furthermore, the *socket* module transparently converts Unicode host names to ACE, so that applications need not be concerned about converting host names themselves when they pass them to the socket module. On top of that, modules that have host names as function parameters, such as *httplib* and *ftplib*, accept Unicode host names (*httplib* then also transparently sends an IDNA hostname in the *Host* field if it sends that field at all).

When receiving host names from the wire (such as in reverse name lookup), no automatic conversion to Unicode is performed: Applications wishing to present such host names to the user should decode them to Unicode.

encodings.idna 模块还实现了 nameprep 过程，该过程会对主机名执行特定的规范化操作，以实现国际域名的大小写不敏感特性与合并相似的字符。如果有需要可以直接使用 nameprep 函数。

`encodings.idna.`**`nameprep`**(*label*)
返回 *label* 经过名称处理操作的版本。该实现目前基于查询字符串，因此 `AllowUnassigned` 为真值。

`encodings.idna.`**`ToASCII`**(*label*)
将标签转换为 ASCII，规则定义见 **RFC 3490**。`UseSTD3ASCIIRules` 预设为假值。

`encodings.idna.`**`ToUnicode`**(*label*)
将标签转换为 Unicode，规则定义见 **RFC 3490**。

7.8.6 `encodings.utf_8_sig` —带 BOM 签名的 UTF-8 编解码器

2.5 新版功能.

This module implements a variant of the UTF-8 codec: On encoding a UTF-8 encoded BOM will be prepended to the UTF-8 encoded bytes. For the stateful encoder this is only done once (on the first write to the byte stream). For decoding an optional UTF-8 encoded BOM at the start of the data will be skipped.

7.9 `unicodedata` —Unicode 数据库

This module provides access to the Unicode Character Database which defines character properties for all Unicode characters. The data in this database is based on the `UnicodeData.txt` file version 5.2.0 which is publicly available from ftp://ftp.unicode.org/.

The module uses the same names and symbols as defined by the UnicodeData File Format 5.2.0 (see https://www.unicode.org/reports/tr44/). It defines the following functions:

`unicodedata.`**`lookup`**(*name*)
Look up character by name. If a character with the given name is found, return the corresponding Unicode character. If not found, *KeyError* is raised.

`unicodedata.`**`name`**(*unichr*[, *default*])
Returns the name assigned to the Unicode character *unichr* as a string. If no name is defined, *default* is returned, or, if not given, *ValueError* is raised.

`unicodedata.`**`decimal`**(*unichr*[, *default*])
Returns the decimal value assigned to the Unicode character *unichr* as integer. If no such value is defined, *default* is returned, or, if not given, *ValueError* is raised.

`unicodedata.`**`digit`**(*unichr*[, *default*])
Returns the digit value assigned to the Unicode character *unichr* as integer. If no such value is defined, *default* is returned, or, if not given, *ValueError* is raised.

unicodedata.**numeric**(*unichr*[, *default*])
Returns the numeric value assigned to the Unicode character *unichr* as float. If no such value is defined, *default* is returned, or, if not given, `ValueError` is raised.

unicodedata.**category**(*unichr*)
Returns the general category assigned to the Unicode character *unichr* as string.

unicodedata.**bidirectional**(*unichr*)
Returns the bidirectional class assigned to the Unicode character *unichr* as string. If no such value is defined, an empty string is returned.

unicodedata.**combining**(*unichr*)
Returns the canonical combining class assigned to the Unicode character *unichr* as integer. Returns `0` if no combining class is defined.

unicodedata.**east_asian_width**(*unichr*)
Returns the east asian width assigned to the Unicode character *unichr* as string.

2.4 新版功能.

unicodedata.**mirrored**(*unichr*)
Returns the mirrored property assigned to the Unicode character *unichr* as integer. Returns `1` if the character has been identified as a "mirrored" character in bidirectional text, `0` otherwise.

unicodedata.**decomposition**(*unichr*)
Returns the character decomposition mapping assigned to the Unicode character *unichr* as string. An empty string is returned in case no such mapping is defined.

unicodedata.**normalize**(*form*, *unistr*)
返回 Unicode 字符串 *unistr* 的正常形式 *form* 。*form* 的有效值为 'NFC'、'NFKC'、'NFD' 和 'NFKD'。

Unicode 标准基于规范等价和兼容性等效的定义定义了 Unicode 字符串的各种规范化形式。在 Unicode 中，可以以各种方式表示多个字符。例如，字符 U+00C7（带有 CEDILLA 的 LATIN CAPITAL LETTER C ）也可以表示为序列 U+0043（LATIN CAPITAL LETTER C ）U+0327（COMBINING CEDILLA ）。

对于每个字符，有两种正规形式：正规形式 C 和正规形式 D 。正规形式 D（NFD）也称为规范分解，并将每个字符转换为其分解形式。正规形式 C（NFC）首先应用规范分解，然后再次组合预组合字符。

除了这两种形式之外，还有两种基于兼容性等效的其他常规形式。在 Unicode 中，支持某些字符，这些字符通常与其他字符统一。例如，U+2160（ROMAN NUMERAL ONE）与 U+0049（LATIN CAPITAL LETTER I）完全相同。但是，Unicode 支持它与现有字符集（例如 gb2312 ）的兼容性。

正规形式 KD（NFKD）将应用兼容性分解，即用其等价项替换所有兼容性字符。正规形式 KC（NFKC）首先应用兼容性分解，然后是规范组合。

即使两个 unicode 字符串被规范化并且人类读者看起来相同，如果一个具有组合字符而另一个没有，则它们可能无法相等。

2.3 新版功能.

此外，该模块暴露了以下常量：

unicodedata.**unidata_version**
此模块中使用的 Unicode 数据库的版本。

2.3 新版功能.

unicodedata.**ucd_3_2_0**
这是一个与整个模块具有相同方法的对象，但对于需要此特定版本的 Unicode 数据库（如 IDNA ）的应用程序，则使用 Unicode 数据库版本 3.2 。

2.5 新版功能.

示例：

```
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
u'{'
>>> unicodedata.name(u'/')
'SOLIDUS'
>>> unicodedata.decimal(u'9')
9
>>> unicodedata.decimal(u'a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not a decimal
>>> unicodedata.category(u'A')  # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional(u'\u0660') # 'A'rabic, 'N'umber
'AN'
```

7.10 stringprep —因特网字符串预备

2.3 新版功能.

在标识因特网上的事物（例如主机名），经常需要比较这些标识是否（相等）。这种比较的具体执行可能会取决于应用域的不同，例如是否要区分大小写等等。有时也可能需要限制允许的标识为仅由“可打印”字符组成。

RFC 3454 定义了在因特网协议中 Unicode 字符串的“预备”过程。在将字符串连线传输之前，它们会先使用预备过程进行处理，之后它们将具有特定的标准形式。该 RFC 定义了一系列表格，它们可以被组合为选项配置。每个配置必须定义所使用的表格，stringprep 过程的其他可选项也是配置的组成部分。stringprep 配置的一个例子是 nameprep，它被用于国际化域名。

The module *stringprep* only exposes the tables from RFC 3454. As these tables would be very large to represent them as dictionaries or lists, the module uses the Unicode character database internally. The module source code itself was generated using the `mkstringprep.py` utility.

As a result, these tables are exposed as functions, not as data structures. There are two kinds of tables in the RFC: sets and mappings. For a set, *stringprep* provides the “characteristic function”, i.e. a function that returns true if the parameter is part of the set. For mappings, it provides the mapping function: given the key, it returns the associated value. Below is a list of all functions available in the module.

stringprep.**in_table_a1**(*code*)
确定 *code* 是否属于 tableA.1 (Unicode 3.2 中的未分配码位)。

stringprep.**in_table_b1**(*code*)
确定 *code* 是否属于 tableB.1 (通常映射为空值)。

stringprep.**map_table_b2**(*code*)
返回 *code* 依据 tableB.2 (配合 NFKC 使用的大小写转换映射) 所映射的值。

stringprep.**map_table_b3**(*code*)
返回 *code* 依据 tableB.3 (不附带正规化的大小写折叠映射) 所映射的值。

stringprep.**in_table_c11**(*code*)
确定 *code* 是否属于 tableC.1.1 (ASCII 空白字符)。

stringprep.**in_table_c12**(*code*)
确定 *code* 是否属于 tableC.1.2 (非 ASCII 空白字符)。

stringprep.**in_table_c11_c12**(*code*)
确定 *code* 是否属于 tableC.1 (空白字符，C.1.1 和 C.1.2 的并集)。

stringprep.**in_table_c21**(*code*)
确定 *code* 是否属于 tableC.2.1 (ASCII 控制字符)。

stringprep.**in_table_c22**(*code*)
确定 *code* 是否属于 tableC.2.2 (非 ASCII 控制字符)。

stringprep.**in_table_c21_c22**(*code*)
确定 *code* 是否属于 tableC.2 (控制字符，C.2.1 和 C.2.2 的并集)。

stringprep.**in_table_c3**(*code*)
确定 *code* 是否属于 tableC.3 (私有使用)。

stringprep.**in_table_c4**(*code*)
确定 *code* 是否属于 tableC.4 (非字符码位)。

stringprep.**in_table_c5**(*code*)
确定 *code* 是否属于 tableC.5 (替代码)。

stringprep.**in_table_c6**(*code*)
确定 *code* 是否属于 tableC.6 (不适用于纯文本)。

stringprep.**in_table_c7**(*code*)
确定 *code* 是否属于 tableC.7 (不适用于规范表示)。

stringprep.**in_table_c8**(*code*)
确定 *code* 是否属于 tableC.8 (改变显示属性或已弃用)。

stringprep.**in_table_c9**(*code*)
确定 *code* 是否属于 tableC.9 (标记字符)。

stringprep.**in_table_d1**(*code*)
确定 *code* 是否属于 tableD.1 (带有双向属性“R”或“AL”的字符)。

stringprep.**in_table_d2**(*code*)
确定 *code* 是否属于 tableD.2 (带有双向属性“L”的字符)。

7.11 fpformat —Floating point conversions

2.6 版后已移除: The *fpformat* module has been removed in Python 3.

The *fpformat* module defines functions for dealing with floating point numbers representations in 100% pure Python.

注解: This module is unnecessary: everything here can be done using the `%` string interpolation operator described in the *String Formatting Operations* section.

The *fpformat* module defines the following functions and an exception:

fpformat.**fix**(*x*, *digs*)
Format *x* as `[-]ddd.ddd` with *digs* digits after the point and at least one digit before. If `digs <= 0`, the decimal point is suppressed.

x can be either a number or a string that looks like one. *digs* is an integer.

Return value is a string.

fpformat.**sci**(*x*, *digs*)
Format *x* as `[-]d.dddE[+-]ddd` with *digs* digits after the point and exactly one digit before. If `digs <= 0`, one digit is kept and the point is suppressed.

x can be either a real number, or a string that looks like one. *digs* is an integer.

Return value is a string.

exception fpformat.**NotANumber**
Exception raised when a string passed to *fix()* or *sci()* as the *x* parameter does not look like a number. This is a subclass of *ValueError* when the standard exceptions are strings. The exception value is the improperly formatted string that caused the exception to be raised.

Example:

```
>>> import fpformat
>>> fpformat.fix(1.23, 1)
'1.2'
```

CHAPTER 8

数据类型

The modules described in this chapter provide a variety of specialized data types such as dates and times, fixed-type arrays, heap queues, synchronized queues, and sets.

Python also provides some built-in data types, in particular, *dict*, list, *set* (which along with *frozenset*, replaces the deprecated *sets* module), and *tuple*. The *str* class can be used to handle binary data and 8-bit text, and the *unicode* class to handle Unicode text.

本章包含以下模块的文档:

8.1 datetime —基本的日期和时间类型

2.3 新版功能.

datetime 模块提供了可以通过多种方式操作日期和时间的类。在支持日期时间数学运算的同时，实现的关注点更着重于如何能够更有效地解析其属性用于格式化输出和数据操作。相关功能可以参阅*time* 和*calendar* 模块。

有两种日期和时间的对象：“简单型“和”感知型“。

感知型对象有着用足以支持一些应用层面算法和国家层面时间调整的信息，例如时区和夏令时，来让自己和其他的感知型对象区别开来。感知型对象是用来表达不对解释器开放的特定时间信息[1]。

A naive object does not contain enough information to unambiguously locate itself relative to other date/time objects. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other timezone is purely up to the program, just like it’s up to the program whether a particular number represents metres, miles, or mass. Naive objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

For applications requiring aware objects, *datetime* and *time* objects have an optional time zone information attribute, tzinfo, that can be set to an instance of a subclass of the abstract *tzinfo* class. These *tzinfo* objects capture information about the offset from UTC time, the time zone name, and whether Daylight Saving Time is in effect. Note that no concrete *tzinfo* classes are supplied by the *datetime* module. Supporting timezones at whatever level of

[1] 就是说如果我们忽略相对论效应的话。

detail is required is up to the application. The rules for time adjustment across the world are more political than rational, and there is no standard suitable for every application.

The *datetime* module exports the following constants:

datetime.**MINYEAR**
date 或者*datetime* 对象允许的最小年份。常量*MINYEAR* 是 1 。

datetime.**MAXYEAR**
date 或*datetime* 对象允许最大的年份。常量*MAXYEAR* 是 9999 。

参见:

模块*calendar* 日历相关函数

模块*time* 时间的访问和转换

8.1.1 有效的类型

class datetime.**date**
一个理想化的简单型日期，它假设当今的公历在过去和未来永远有效。属性: *year*, *month*, and *day*。

class datetime.**time**
一个理想化的时间，它独立于任何特定的日期，假设每天一共有 24*60*60 秒（这里没有”闰秒”的概念）。属性：*hour*, *minute*, *second*, *microsecond*, 和*tzinfo*。

class datetime.**datetime**
日期和时间的结合。属性：*year*, *month*, *day*, *hour*, *minute*, *second*, *microsecond*, and *tzinfo*.

class datetime.**timedelta**
表示两个*date* 对象或者*time* 对象, 或者*datetime* 对象之间的时间间隔，精确到微秒。

class datetime.**tzinfo**
一个描述时区信息的抽象基类。用于给*datetime* 类和*time* 类提供自定义的时间调整概念（例如，负责时区或者夏令时）。

这些类型的对象都是不可变的。

date 类型的对象都是简单型的。

An object of type *time* or *datetime* may be naive or aware. A *datetime* object *d* is aware if `d.tzinfo` is not `None` and `d.tzinfo.utcoffset(d)` does not return `None`. If `d.tzinfo` is `None`, or if `d.tzinfo` is not `None` but `d.tzinfo.utcoffset(d)` returns `None`, *d* is naive. A *time* object *t* is aware if `t.tzinfo` is not `None` and `t.tzinfo.utcoffset(None)` does not return `None`. Otherwise, *t* is naive.

简单型和感知型之间的差别不适用于*timedelta* 对象。

子类关系

```
object
    timedelta
    tzinfo
    time
    date
        datetime
```

8.1.2 timedelta 类对象

timedelta 对象表示两个 date 或者 time 的时间间隔。

class datetime.**timedelta**([*days*[, *seconds*[, *microseconds*[, *milliseconds*[, *minutes*[, *hours*[, *weeks*]]]]]])

All arguments are optional and default to 0. Arguments may be ints, longs, or floats, and may be positive or negative.

只有 *days*,*seconds* 和 *microseconds* 会存储在内部，即 python 内部以 *days*,*seconds* 和 *microseconds* 三个单位作为存储的基本单位。参数单位转换规则如下：

- 1 毫秒会转换成 1000 微秒。
- 1 分钟会转换成 60 秒。
- 1 小时会转换成 3600 秒。
- 1 星期会转换成 7 天。

days, seconds, microseconds 本身也是标准化的，以保证表达方式的唯一性，例：

- `0 <= microseconds < 1000000`
- `0 <= seconds < 3600*24` (一天的秒数)
- `-999999999 <= days <= 999999999`

If any argument is a float and there are fractional microseconds, the fractional microseconds left over from all arguments are combined and their sum is rounded to the nearest microsecond. If no argument is a float, the conversion and normalization processes are exact (no information is lost).

如果标准化后的 days 数值超过了指定范围，将会抛出*OverflowError* 异常。

需要注意的是，负数被标准化后的结果会让你大吃一惊。例如，

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

类属性：

timedelta.**min**
The most negative *timedelta* object, `timedelta(-999999999)`.

timedelta.**max**
The most positive *timedelta* object, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

timedelta.**resolution**
两个不相等的*timedelta* 类对象最小的间隔为 `timedelta(microseconds=1)`。

需要注意的是，因为标准化的缘故，`timedelta.max > -timedelta.min`，`-timedelta.max` 不可以表示一个*timedelta* 类对象。

实例属性（只读）：

属性	值
days	-999999999 至 999999999，含 999999999
seconds	0 至 86399，包含 86399
microseconds	0 至 999999，包含 999999

支持的运算：

运算	结果:
`t1 = t2 + t3`	*t2* 和 *t3* 的和。运算后 *t1-t2 == t3* and *t1-t3 == t2* 必为真值。(1)
`t1 = t2 - t3`	Difference of *t2* and *t3*. Afterwards *t1 == t2 - t3* and *t2 == t1 + t3* are true. (1)
`t1 = t2 * i or t1 = i * t2`	Delta multiplied by an integer or long. Afterwards *t1* // i == *t2* is true, provided `i != 0`.
	In general, *t1* * i == *t1* * (i-1) + *t1* is true. (1)
`t1 = t2 // i`	The floor is computed and the remainder (if any) is thrown away. (3)
`+t1`	返回一个相同数值的*timedelta* 对象。
`-t1`	等价于*timedelta*(*-t1.days*, *-t1.seconds*, *-t1.microseconds*), 和 *t1** -1. (1)(4)
`abs(t)`	当 `t.days >= 0``时等于 +\ *t*, 当 ``t.days < 0` 时 *-t* 。(2)
`str(t)`	返回一个形如 `[D day[s], ][H]H:MM:SS[.UUUUUU]` 的字符串，当 `t` 为负数的时候，D 也为负数。(5)
`repr(t)`	Returns a string in the form `datetime.timedelta(D[, S[, U]])`, where D is negative for negative `t`. (5)

注释:

(1) 精确但可能会溢出。

(2) 精确且不会溢出。

(3) 除以 0 将会抛出异常*ZeroDivisionError* 。

(4) *-timedelta.max* 不是一个*timedelta* 类对象。

(5) String representations of *timedelta* objects are normalized similarly to their internal representation. This leads to somewhat unusual results for negative timedeltas. For example:

```
>>> timedelta(hours=-5)
datetime.timedelta(-1, 68400)
>>> print(_)
-1 day, 19:00:00
```

除了上面列举的操作以外*timedelta* 对象还支持与*date* 和*datetime* 对象进行特定的相加和相减运算（见下文）。

timedelta 对象与*timedelta* 对象比较的支持是通过将表示较小时间差的 timedelta 对象视为较小值。为了防止将混合类型比较回退为基于对象地址的默认比较，当*timedelta* 对象与不同类型的对象比较时，将会引发*TypeError*，除非比较运算符是 == 或 !=。在后一种情况下将分别返回*False* 或*True*。

timedelta 对象是*hashable* 类型的 (可以作为字典关键字), 支持高效获取, 在布尔上下文中，*timedelta* 对象大多数情况下都被视为真，仅在不等于 `timedelta(0)` 时。

实例方法:

`timedelta.`**`total_seconds`**`()`

Return the total number of seconds contained in the duration. Equivalent to `(td.microseconds + (td.seconds + td.days * 24 * 3600) * 10**6) / 10**6` computed with true division enabled.

需要注意的是，时间间隔较大时，这个方法的结果中的微秒将会失真（大多数平台上大于 270 年视为一个较大的时间间隔）。

2.7 新版功能.

用法示例:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
```

(下页继续)

(续上页)

```
...                           minutes=50, seconds=600)  # adds up to 365 days
>>> year.total_seconds()
31536000.0
>>> year == another_year
True
>>> ten_years = 10 * year
>>> ten_years, ten_years.days // 365
(datetime.timedelta(3650), 10)
>>> nine_years = ten_years - year
>>> nine_years, nine_years.days // 365
(datetime.timedelta(3285), 9)
>>> three_years = nine_years // 3;
>>> three_years, three_years.days // 365
(datetime.timedelta(1095), 3)
>>> abs(three_years - ten_years) == 2 * three_years + year
True
```

8.1.3 date 对象

date 对象代表一个理想化历法中的日期（年、月和日），即当今的格列高利历向前后两个方向无限延伸。公元 1 年 1 月 1 日是第 1 日，公元 1 年 1 月 2 日是第 2 日，依此类推。这与 Dershowitz 与 Reingold 所著 Calendrical Calculations 中“预期格列高利”历法的定义一致，它是适用于该书中所有运算的基础历法。请参阅该书了解在预期格列高利历序列与许多其他历法系统之间进行转换的算法。

class datetime.**date**(*year*, *month*, *day*)

All arguments are required. Arguments may be ints or longs, in the following ranges:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= 日期 <= 给定年月对应的天数`

如果参数不在这些范围内，则抛出*ValueError* 异常。

其它构造器，所有的类方法：

classmethod date.**today**()

返回当地的当前日期。与 “date.fromtimestamp(time.time())“等价。

classmethod date.**fromtimestamp**(*timestamp*)

Return the local date corresponding to the POSIX timestamp, such as is returned by *time.time()*. This may raise *ValueError*, if the timestamp is out of the range of values supported by the platform C `localtime()` function. It’s common for this to be restricted to years from 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by *fromtimestamp()*.

classmethod date.**fromordinal**(*ordinal*)

返回对应于预期格列高利历序号的日期，其中公元 1 年 1 月 1 日的序号为 1。除非 `1 <= 序号 <= date.max.toordinal()` 否则会引发*ValueError*。对于任意日期 *d*，`date.fromordinal(d.toordinal()) == d`。

类属性：

date.**min**

最小的日期 `date(MINYEAR, 1, 1)`。

date.**max**

最大的日期，`date(MAXYEAR, 12, 31)`。

date.**resolution**
两个日期对象的最小间隔，timedelta(days=1)。

实例属性（只读）：

date.**year**
在*MINYEAR* 和*MAXYEAR* 之间，包含边界。

date.**month**
1 至 12（含）

date.**day**
返回 1 到指定年月的天数间的数字。

支持的运算：

运算	结果：
date2 = date1 + timedelta	*date2* 等于从 *date1* 减去 timedelta.days 天。(1)
date2 = date1 - timedelta	计算 *date2* 的值使得 date2 + timedelta == date1。(2)
timedelta = date1 - date2	(3)
date1 < date2	如果 *date1* 的时间在 *date2* 之前则认为 *date1* 小于 *date2* 。(4)

注释:

(1) 如果 timedelta.days > 0 则 *date2* 在时间线上前进，如果 timedelta.days < 0 则后退。操作完成后 date2 - date1 == timedelta.days。timedelta.seconds 和 timedelta.microseconds 会被忽略。如果 date2.year 将小于*MINYEAR* 或大于*MAXYEAR* 则会引发*OverflowError*。.

(2) This isn't quite equivalent to date1 + (-timedelta), because -timedelta in isolation can overflow in cases where date1 - timedelta does not. timedelta.seconds and timedelta.microseconds are ignored.

(3) 精确且不会溢出。操作完成后 timedelta.seconds 和 timedelta.microseconds 均为 0，并且 date2 + timedelta == date1。

(4) In other words, date1 < date2 if and only if date1.toordinal() < date2.toordinal(). In order to stop comparison from falling back to the default scheme of comparing object addresses, date comparison normally raises *TypeError* if the other comparand isn't also a *date* object. However, NotImplemented is returned instead if the other comparand has a timetuple() attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a *date* object is compared to an object of a different type, *TypeError* is raised unless the comparison is == or !=. The latter cases return *False* or *True*, respectively.

日期可以作为字典的关键字。在布尔上下文中，所有的*date* 对象都视为真。

实例方法：

date.**replace**(*year*, *month*, *day*)
返回一个具有同样值的日期，除非通过任何关键字参数给出了某些形参的新值。例如，如果 d == date(2002, 12, 31)，则 d.replace(day=26) == date(2002, 12, 26)。

date.**timetuple**()
返回一个*time.struct_time*，即与*time.localtime()* 的返回类型相同。hours, minutes 和 seconds 值为 0, 且 DST 标志值为 -1。d.timetuple() 等价于 time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))，其中 yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1 是日期在当前年份中的序号，起始序号 1 表示 1 月 1 日。

date.**toordinal**()
返回日期的预期格列高利历序号，其中公元 1 年 1 月 1 日的序号为 1。对于任意*date* 对象 *d*，date.fromordinal(d.toordinal()) == d。

date.**weekday**()
返回一个整数代表星期几，星期一为 0，星期天为 6。例如，date(2002, 12, 4).weekday() == 2，表示的是星期三。参阅*isoweekday()*。

date.**isoweekday**()
返回一个整数代表星期几，星期一为 1，星期天为 7。例如：date(2002, 12, 4).isoweekday() == 3, 表示星期三。参见*weekday()*, *isocalendar()*。

date.**isocalendar**()
返回一个三元元组，(ISO year, ISO week number, ISO weekday) 。

ISO 日历是一个被广泛使用的公历。可以从 https://www.staff.science.uu.nl/~gent0113/calendar/isocalendar.htm 上查看更完整的说明。

ISO 年由 52 或 53 个完整星期构成，每个星期开始于星期一结束于星期日。一个 ISO 年的第一个星期就是（格列高利）历法的的一年中第一个包含星期四的星期。这被称为 1 号星期，其中星期四所在的 ISO 年与其所在的格列高利年相同。

例如，2004 年的第一天是一个星期四，因此 ISO 2004 年的第一个星期开始于 2003 年 12 月 29 日星期一，结束于 2004 年 1 月 4 日星期日，因此 date(2003, 12, 29).isocalendar() == (2004, 1, 1) and date(2004, 1, 4).isocalendar() == (2004, 1, 7)。

date.**isoformat**()
返回一个 ISO 8601 格式的字符串，‘YYYY-MM-DD’。例如 date(2002, 12, 4).isoformat() == '2002-12-04'。

date.**__str__**()
对于日期对象 *d*, str(d) 等价于 d.isoformat() 。

date.**ctime**()
返回一个代表日期的字符串，例如 date(2002, 12, 4).ctime() == 'Wed Dec 4 00:00:00 2002'。在原生 C ctime() 函数 (*time.ctime()* 会发起调用该函数，但*date.ctime()* 则不会) 遵循 C 标准的平台上，d.ctime() 等价于 time.ctime(time.mktime(d.timetuple()))。

date.**strftime**(*format*)
Return a string representing the date, controlled by an explicit format string. Format codes referring to hours, minutes or seconds will see 0 values. For a complete list of formatting directives, see section *strftime() 和 strptime() 的行为*.

date.**__format__**(*format*)
Same as *date.strftime()*. This makes it possible to specify a format string for a *date* object when using *str.format()*. See section *strftime() 和 strptime() 的行为*.

计算距离特定事件天数的例子:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

使用*date*的例子:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)
>>> t = d.timetuple()
>>> for i in t:
...     print i
2002                # year
3                   # month
11                  # day
0
0
0
0                   # weekday (0 = Monday)
70                  # 70th day in the year
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...     print i
2002                # ISO year
11                  # ISO week number
1                   # ISO day number ( 1 = Monday )
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'
```

8.1.4 datetime 对象

*datetime*对象是一个包含了来自*date*对象和*time*对象所有信息的单一对象。与*date*对象一样，*datetime*假定当今的格列高利历向前后两个方向无限延伸；与 time 对象一样，*datetime*假定每一天恰好有 3600*24 秒。

构造器:

class datetime.**datetime**(*year*, *month*, *day*[, *hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]])

The year, month and day arguments are required. *tzinfo* may be `None`, or an instance of a *tzinfo* subclass. The remaining arguments may be ints or longs, in the following ranges:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= 日期 <= 给定年月对应的天数`
- `0 <= hour < 24`
- `0 <= minute < 60`
- `0 <= second < 60`
- `0 <= microsecond < 1000000`

如果参数不在这些范围内，则抛出*ValueError*异常。

其它构造器，所有的类方法：

classmethod datetime.**today**()
返回当前的本地 datetime，*tzinfo* 值为 None。这等价于 datetime.fromtimestamp(time.time())。另请参阅*now()*, *fromtimestamp()*。

classmethod datetime.**now**([*tz*])
返回当前的本地 date 和 time。如果可选参数 *tz* 为 None 或未指定，这就类似于*today()*，但该方法会在可能的情况下提供比通过*time.time()* 时间戳所获时间值更高的精度（例如，在提供了 C gettimeofday() 函数的平台上就可能做到）。

如果 *tz* 不为 None，它必须是*tzinfo* 的子类的一个实例，并且当前日期和时间将转换为 *tz* 时区的日期和时间。在这种情况下结果等价于 tz.fromutc(datetime.utcnow().replace(tzinfo=tz))。另请参阅*today()*, *utcnow()*。

classmethod datetime.**utcnow**()
Return the current UTC date and time, with *tzinfo* None. This is like *now()*, but returns the current UTC date and time, as a naive *datetime* object. See also *now()*.

classmethod datetime.**fromtimestamp**(*timestamp*[, *tz*])
返回对应于 POSIX 时间戳例如*time.time()* 的返回值的本地日期和时间。如果可选参数 *tz* 为 None 或未指定，时间戳会被转换为所在平台的本地日期和时间，返回的*datetime* 对象将为天真型。

如果 *tz* 不为 None，它必须是*tzinfo* 子类的一个实例，并且时间戳将被转换到 *tz* 指定的时区。在这种情况下结果等价于 tz.fromutc(datetime.utcfromtimestamp(timestamp).replace(tzinfo=tz))。

fromtimestamp() may raise *ValueError*, if the timestamp is out of the range of values supported by the platform C localtime() or gmtime() functions. It's common for this to be restricted to years in 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by *fromtimestamp()*, and then it's possible to have two timestamps differing by a second that yield identical *datetime* objects. See also *utcfromtimestamp()*.

classmethod datetime.**utcfromtimestamp**(*timestamp*)
Return the UTC *datetime* corresponding to the POSIX timestamp, with *tzinfo* None. This may raise *ValueError*, if the timestamp is out of the range of values supported by the platform C gmtime() function. It's common for this to be restricted to years in 1970 through 2038. See also *fromtimestamp()*.

classmethod datetime.**fromordinal**(*ordinal*)
返回对应于预期格列高利历序号的*datetime*，其中公元 1 年 1 月 1 日的序号为 1。除非 1 <= ordinal <= datetime.max.toordinal() 否则会引发*ValueError*。结果的 hour, minute, second 和 microsecond 值均为 0，并且*tzinfo* 值为 None。

classmethod datetime.**combine**(*date*, *time*)
Return a new *datetime* object whose date components are equal to the given *date* object's, and whose time components and *tzinfo* attributes are equal to the given *time* object's. For any *datetime* object *d*, d == datetime.combine(d.date(), d.timetz()). If date is a *datetime* object, its time components and *tzinfo* attributes are ignored.

classmethod datetime.**strptime**(*date_string*, *format*)
Return a *datetime* corresponding to *date_string*, parsed according to *format*. This is equivalent to datetime(*(time.strptime(date_string, format)[0:6])). *ValueError* is raised if the date_string and format can't be parsed by *time.strptime()* or if it returns a value which isn't a time tuple. For a complete list of formatting directives, see section *strftime() 和 strptime() 的行为*.

2.5 新版功能.

类属性：

datetime.**min**
最早的可表示*datetime*，datetime(MINYEAR, 1, 1, tzinfo=None)。

datetime.**max**
最晚的可表示*datetime*，datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)。

datetime.**resolution**
两个不相等的*datetime*对象之间可能的最小间隔，timedelta(microseconds=1)。

实例属性（只读）：

datetime.**year**
在*MINYEAR*和*MAXYEAR*之间，包含边界。

datetime.**month**
1 至 12（含）

datetime.**day**
返回 1 到指定年月的天数间的数字。

datetime.**hour**
取值范围是 range(24)。

datetime.**minute**
取值范围是 range(60)。

datetime.**second**
取值范围是 range(60)。

datetime.**microsecond**
取值范围是 range(1000000)。

datetime.**tzinfo**
作为 *tzinfo* 参数被传给*datetime*构造器的对象，如果没有传入值则为 None。

支持的运算：

运算	结果：
datetime2 = datetime1 + timedelta	(1)
datetime2 = datetime1 - timedelta	(2)
timedelta = datetime1 - datetime2	(3)
datetime1 < datetime2	比较*datetime*与*datetime*。(4)

(1) datetime2 是从中去掉的一段 timedelta 的结果，如果 timedelta.days > 0 则是在时间线上前进，如果 timedelta.days < 0 则后退。结果具有与输入的 datetime 相同的*tzinfo*属性，并且操作完成后 datetime2 - datetime1 == timedelta。如果 datetime2.year 将小于*MINYEAR*或大于*MAXYEAR*则会引发*OverflowError*。请注意即使输入的是一个感知型对象，该方法也不会进行时区调整。

(2) Computes the datetime2 such that datetime2 + timedelta == datetime1. As for addition, the result has the same *tzinfo* attribute as the input datetime, and no time zone adjustments are done even if the input is aware. This isn't quite equivalent to datetime1 + (-timedelta), because -timedelta in isolation can overflow in cases where datetime1 - timedelta does not.

(3) 从一个*datetime*减去一个*datetime*仅对两个操作数均为简单型或均为感知型时有定义。如果一个是感知型而另一个是简单型，则会引发*TypeError*。

如果两个操作数都是简单型，或都是感知型且具有相同的*tzinfo*属性，*tzinfo*属性会被忽略，结果是一个使得 datetime2 + t == datetime1 的*timedelta*对象 *t*。在此情况下不会进行时区调整。

如果两个操作数都是感知型且具有不同的*tzinfo*属性，a-b 操作的行为就如同 *a* 和 *b* 被首先转换为简单型 UTC 日期时间。结果将是 (a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset()) 除非具体实现绝对不溢出。

(4) 当 *datetime1* 的时间在 *datetime2* 之前则认为 *datetime1* 小于 *datetime2*。

If one comparand is naive and the other is aware, *TypeError* is raised. If both comparands are aware, and have the same *tzinfo* attribute, the common *tzinfo* attribute is ignored and the base datetimes are compared. If both comparands are aware and have different *tzinfo* attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`).

注解: 为了防止比较操作回退为默认的对象地址比较方案，datetime 比较通常会引发*TypeError*，如果比较目标不同样为*datetime* 对象的话。不过也可能会返回 `NotImplemented` 如果比较目标具有 `timetuple()` 属性的话。这个钩子给予其他日期对象类型实现混合类型比较的机会。否则，当*datetime* 对象与不同类型的对象比较时将会引发*TypeError*，除非 == 或 != 比较。后两种情况将分别返回*False* 或*True*。

---

datetime 对象可以用作字典的键。在布尔运算时，所有*datetime* 对象都被视为真值。

实例方法:

`datetime.`**`date`**`()`
返回具有同样 year, month 和 day 值的*date* 对象。

`datetime.`**`time`**`()`
Return *time* object with same hour, minute, second and microsecond. *tzinfo* is `None`. See also method *timetz()*.

`datetime.`**`timetz`**`()`
Return *time* object with same hour, minute, second, microsecond, and tzinfo attributes. See also method *time()*.

`datetime.`**`replace`**`(`[*year*[, *month*[, *day*[, *hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]]]]]`)`
返回一个具有同样属性值的 datetime，除非通过任何关键字参数指定了某些属性值。请注意可以通过指定 `tzinfo=None` 从一个感知型 datetime 创建一个简单型 datetime 而不必转换日期和时间值。

`datetime.`**`astimezone`**`(`*tz*`)`
返回一个具有新的*tzinfo* 属性 *tz* 的*datetime* 对象，并会调整日期和时间数据使得结果对应的 UTC 时间与 *self* 相同，但为 *tz* 时区的本地时间。

tz must be an instance of a *tzinfo* subclass, and its *utcoffset()* and *dst()* methods must not return `None`. *self* must be aware (`self.tzinfo` must not be `None`, and `self.utcoffset()` must not return `None`).

If `self.tzinfo` is *tz*, `self.astimezone(tz)` is equal to *self*: no adjustment of date or time data is performed. Else the result is local time in time zone *tz*, representing the same UTC time as *self*: after `astz = dt.astimezone(tz), astz - astz.utcoffset()` will usually have the same date and time data as `dt - dt.utcoffset()`. The discussion of class *tzinfo* explains the cases at Daylight Saving Time transition boundaries where this cannot be achieved (an issue only if *tz* models both standard and daylight time).

如果你只想附加一个时区对象 *tz* 给一个 datetime 对象 *dt* 而不调整日期和时间数据，请使用 `dt.replace(tzinfo=tz)`。如果你只想从一个感知型 datetime 对象 *dt* 移除时区对象则不转换日期和时间数据，请使用 `dt.replace(tzinfo=None)`。

请注意默认的*tzinfo.fromutc()* 方法在*tzinfo* 的子类中可以被重载，从而影响*astimezone()* 的返回结果。如果忽略出错的情况，*astimezone()* 的行为就类似于:

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new time zone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

datetime.**utcoffset**()
If *tzinfo* is None, returns None, else returns self.tzinfo.utcoffset(self), and raises an exception if the latter doesn't return None, or a *timedelta* object representing a whole number of minutes with magnitude less than one day.

datetime.**dst**()
If *tzinfo* is None, returns None, else returns self.tzinfo.dst(self), and raises an exception if the latter doesn't return None, or a *timedelta* object representing a whole number of minutes with magnitude less than one day.

datetime.**tzname**()
如果*tzinfo* 为 None，则返回 None，否则返回 self.tzinfo.tzname(self)，如果后者不返回 None 或者一个字符串对象则将引发异常。

datetime.**timetuple**()
返回一个*time.struct_time*，即与*time.localtime()* 的返回类型相同。d.timetuple() 等价于 time.struct_time((d.year, d.month, d.day, d.hour, d.minute, d.second, d.weekday(), yday, dst))，其中 yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1 是日期在当前年份中的序号，起始序号 1 表示 1 月 1 日。结果的 tm_isdst 旗标的设定会依据*dst()* 方法：如果*tzinfo* 为 None 或*dst()* 返回 None，则 tm_isdst 将设为 -1；否则如果*dst()* 返回一个非零值，则 tm_isdst 将设为 1；否则 tm_isdst 将设为 0。

datetime.**utctimetuple**()
如果*datetime* 实例 *d* 为简单型，这类似于 d.timetuple()，不同之处为 tm_isdst 会强设为 0，无论 d.dst() 返回什么结果。DST 对于 UTC 时间永远无效。

If *d* is aware, *d* is normalized to UTC time, by subtracting d.utcoffset(), and a *time.struct_time* for the normalized time is returned. tm_isdst is forced to 0. Note that the result's tm_year member may be *MINYEAR*-1 or *MAXYEAR*+1, if *d*.year was MINYEAR or MAXYEAR and UTC adjustment spills over a year boundary.

datetime.**toordinal**()
返回日期的预期格列高利历序号。与 self.date().toordinal() 相同。

datetime.**weekday**()
返回一个整数代表星期几，星期一为 0，星期天为 6。相当于 self.date().weekday()。另请参阅*isoweekday()*。

datetime.**isoweekday**()
返回一个整数代表星期几，星期一为 1，星期天为 7。相当于 self.date().isoweekday()。另请参阅*weekday()*, *isocalendar()*。

datetime.**isocalendar**()
返回一个 3 元组 (ISO 年份, ISO 周序号, ISO 周日期)。相当于 self.date().isocalendar()。

datetime.**isoformat**([*sep*])
Return a string representing the date and time in ISO 8601 format, YYYY-MM-DDTHH:MM:SS.mmmmmm or, if *microsecond* is 0, YYYY-MM-DDTHH:MM:SS

If *utcoffset()* does not return None, a 6-character string is appended, giving the UTC offset in (signed) hours and minutes: YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM or, if *microsecond* is 0 YYYY-MM-DDTHH:MM:SS+HH:MM

可选参数 *sep* (默认为 'T') 为单个分隔字符，会被放在结果的日期和时间两部分之间。例如

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt): return timedelta(minutes=-399)
...
```

(下页继续)

(续上页)

```
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
```

datetime.**__str__**()
对于*datetime*实例 *d*，str(d) 等价于 d.isoformat(' ')。

datetime.**ctime**()
返回一个代表日期和时间的字符串，例如 datetime(2002, 12, 4, 20, 30, 40).ctime() == 'Wed Dec 4 20:30:40 2002'。在原生 C ctime() 函数 (*time.ctime()* 会发起调用该函数，但*datetime.ctime()* 则不会) 遵循 C 标准的平台上，d.ctime() 等价于 time.ctime(time.mktime(d.timetuple()))。

datetime.**strftime**(*format*)
Return a string representing the date and time, controlled by an explicit format string. For a complete list of formatting directives, see section *strftime() 和 strptime() 的行为*.

datetime.**__format__**(*format*)
Same as *datetime.strftime()*. This makes it possible to specify a format string for a *datetime* object when using *str.format()*. See section *strftime() 和 strptime() 的行为*.

使用 datetime 对象的例子:

```
>>> from datetime import datetime, date, time
>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)
>>> # Using datetime.now() or datetime.utcnow()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
>>> datetime.utcnow()
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060)
>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)
>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print it
...
2006    # year
11      # month
21      # day
16      # hour
30      # minute
0       # second
1       # weekday (0 = Monday)
325     # number of days since 1st January
-1      # dst - method tzinfo.dst() returned None
>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print it
...
2006    # ISO year
```

(下页继续)

(续上页)

```
47       # ISO week
2        # ISO weekday
>>> # Formatting datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format(dt, "day",
↪"month", "time")
'The day is 21, the month is November, the time is 04:30PM.'
```

使用 datetime 并附带 tzinfo:

```
>>> from datetime import timedelta, datetime, tzinfo
>>> class GMT1(tzinfo):
...      def utcoffset(self, dt):
...          return timedelta(hours=1) + self.dst(dt)
...      def dst(self, dt):
...          # DST starts last Sunday in March
...          d = datetime(dt.year, 4, 1)   # ends last Sunday in October
...          self.dston = d - timedelta(days=d.weekday() + 1)
...          d = datetime(dt.year, 11, 1)
...          self.dstoff = d - timedelta(days=d.weekday() + 1)
...          if self.dston <=  dt.replace(tzinfo=None) < self.dstoff:
...              return timedelta(hours=1)
...          else:
...              return timedelta(0)
...      def tzname(self,dt):
...           return "GMT +1"
...
>>> class GMT2(tzinfo):
...      def utcoffset(self, dt):
...          return timedelta(hours=2) + self.dst(dt)
...      def dst(self, dt):
...          d = datetime(dt.year, 4, 1)
...          self.dston = d - timedelta(days=d.weekday() + 1)
...          d = datetime(dt.year, 11, 1)
...          self.dstoff = d - timedelta(days=d.weekday() + 1)
...          if self.dston <=  dt.replace(tzinfo=None) < self.dstoff:
...              return timedelta(hours=1)
...          else:
...              return timedelta(0)
...      def tzname(self,dt):
...          return "GMT +2"
...
>>> gmt1 = GMT1()
>>> # Daylight Saving Time
>>> dt1 = datetime(2006, 11, 21, 16, 30, tzinfo=gmt1)
>>> dt1.dst()
datetime.timedelta(0)
>>> dt1.utcoffset()
datetime.timedelta(0, 3600)
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=gmt1)
>>> dt2.dst()
datetime.timedelta(0, 3600)
>>> dt2.utcoffset()
datetime.timedelta(0, 7200)
>>> # Convert datetime to another time zone
```

(下页继续)

(续上页)

```
>>> dt3 = dt2.astimezone(GMT2())
>>> dt3
datetime.datetime(2006, 6, 14, 14, 0, tzinfo=<GMT2 object at 0x...>)
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=<GMT1 object at 0x...>)
>>> dt2.utctimetuple() == dt3.utctimetuple()
True
```

8.1.5 time 对象

一个 time 对象代表某个日期内的（本地）时间，它独立于任何特定日期，并可通过*tzinfo*对象来调整。

class datetime.**time**([*hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]])

All arguments are optional. *tzinfo* may be None, or an instance of a *tzinfo* subclass. The remaining arguments may be ints or longs, in the following ranges:

- 0 <= hour < 24
- 0 <= minute < 60
- 0 <= second < 60
- 0 <= microsecond < 1000000.

如果给出一个此范围以外的参数，则会引发*ValueError*。所有参数值默认为 0，除了 *tzinfo* 默认为*None*。

类属性：

time.**min**
早最的可表示*time*, time(0, 0, 0, 0)。

time.**max**
最晚的可表示*time*, time(23, 59, 59, 999999)。

time.**resolution**
两个不相等的*time* 对象之间可能的最小间隔，timedelta(microseconds=1)，但是请注意*time* 对象并不支持算术运算。

实例属性（只读）：

time.**hour**
取值范围是 range(24)。

time.**minute**
取值范围是 range(60)。

time.**second**
取值范围是 range(60)。

time.**microsecond**
取值范围是 range(1000000)。

time.**tzinfo**
作为 tzinfo 参数被传给*time* 构造器的对象，如果没有传入值则为 None。

支持的运算：

- comparison of *time* to *time*, where *a* is considered less than *b* when *a* precedes *b* in time. If one comparand is naive and the other is aware, *TypeError* is raised. If both comparands are aware, and have the same *tzinfo* attribute, the common *tzinfo* attribute is ignored and the base times are compared. If both comparands are

aware and have different *tzinfo* attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`). In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a *time* object is compared to an object of a different type, *TypeError* is raised unless the comparison is == or !=. The latter cases return *False* or *True*, respectively.

- 哈希，以便用作字典的键
- 高效的封存
- in Boolean contexts, a *time* object is considered to be true if and only if, after converting it to minutes and subtracting `utcoffset()` (or `0` if that's `None`), the result is non-zero.

实例方法:

`time.`**`replace`**`(`[*hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]]`)`
返回一个具有同样属性值的*time*，除非通过任何关键字参数指定了某些属性值。请注意可以通过指定 `tzinfo=None` 从一个感知型*time* 创建一个简单型*time* 而不必转换时间值。

`time.`**`isoformat`**`()`
Return a string representing the time in ISO 8601 format, HH:MM:SS.mmmmmm or, if self.microsecond is 0, HH:MM:SS If *utcoffset()* does not return `None`, a 6-character string is appended, giving the UTC offset in (signed) hours and minutes: HH:MM:SS.mmmmmm+HH:MM or, if self.microsecond is 0, HH:MM:SS+HH:MM

`time.`**`__str__`**`()`
对于时间对象 *t*, `str(t)` 等价于 `t.isoformat()`。

`time.`**`strftime`**`(`*format*`)`
Return a string representing the time, controlled by an explicit format string. For a complete list of formatting directives, see section *strftime() 和 strptime() 的行为*.

`time.`**`__format__`**`(`*format*`)`
Same as *time.strftime()*. This makes it possible to specify a format string for a *time* object when using *str.format()*. See section *strftime() 和 strptime() 的行为*.

`time.`**`utcoffset`**`()`
If *tzinfo* is `None`, returns `None`, else returns `self.tzinfo.utcoffset(None)`, and raises an exception if the latter doesn't return `None` or a *timedelta* object representing a whole number of minutes with magnitude less than one day.

`time.`**`dst`**`()`
If *tzinfo* is `None`, returns `None`, else returns `self.tzinfo.dst(None)`, and raises an exception if the latter doesn't return `None`, or a *timedelta* object representing a whole number of minutes with magnitude less than one day.

`time.`**`tzname`**`()`
如果*tzinfo* 为 `None`，则返回 `None`，否则返回 `self.tzinfo.tzname(None)`，如果后者不返回 `None` 或者一个字符串对象则将引发异常。

示例:

```
>>> from datetime import time, tzinfo, timedelta
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "Europe/Prague"
...
>>> t = time(12, 10, 30, tzinfo=GMT1())
>>> t
```

(下页继续)

(续上页)

```
datetime.time(12, 10, 30, tzinfo=<GMT1 object at 0x...>)
>>> gmt = GMT1()
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'Europe/Prague'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 Europe/Prague'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.6 tzinfo 对象

class datetime.**tzinfo**

This is an abstract base class, meaning that this class should not be instantiated directly. You need to derive a concrete subclass, and (at least) supply implementations of the standard *tzinfo* methods needed by the *datetime* methods you use. The *datetime* module does not supply any concrete subclasses of *tzinfo*.

tzinfo 的（某个实体子类）的实例可以被传给*datetime* 和*time* 对象的构造器。这些对象会将它们的属性视为对应于本地时间，并且*tzinfo* 对象支持展示本地时间与 UTC 的差值、时区名称以及 DST 差值的方法，都是与传给它们的日期或时间对象的相对值。

对于封存操作的特殊要求：一个*tzinfo* 子类必须具有可不带参数调用的 `__init__()` 方法，否则它虽然可以被封存，但可能无法再次解封。这是个技术性要求，在未来可能会被取消。

一个*tzinfo* 的实体子类可能需要实现以下方法。具体需要实现的方法取决于感知型*datetime* 对象如何使用它。如果有疑问，可以简单地全都实现。

tzinfo.**utcoffset**(*self*, *dt*)

Return offset of local time from UTC, in minutes east of UTC. If local time is west of UTC, this should be negative. Note that this is intended to be the total offset from UTC; for example, if a *tzinfo* object represents both time zone and DST adjustments, *utcoffset()* should return their sum. If the UTC offset isn't known, return `None`. Else the value returned must be a *timedelta* object specifying a whole number of minutes in the range -1439 to 1439 inclusive (1440 = 24*60; the magnitude of the offset must be less than one day). Most implementations of *utcoffset()* will probably look like one of these two:

```
return CONSTANT                 # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```

如果*utcoffset()* 返回值不为 `None`，则*dst()* 也不应返回 `None`。

默认的*utcoffset()* 实现会引发*NotImplementedError*。

tzinfo.**dst**(*self*, *dt*)

Return the daylight saving time (DST) adjustment, in minutes east of UTC, or `None` if DST information isn't known. Return `timedelta(0)` if DST is not in effect. If DST is in effect, return the offset as a *timedelta* object (see *utcoffset()* for details). Note that DST offset, if applicable, has already been added to the UTC offset returned by *utcoffset()*, so there's no need to consult *dst()* unless you're interested in obtaining DST info separately. For example, *datetime.timetuple()* calls its *tzinfo* attribute's *dst()* method to determine how the `tm_isdst` flag should be set, and *tzinfo.fromutc()* calls *dst()* to account for DST changes when crossing time zones.

一个可以同时处理标准时和夏令时的*tzinfo* 子类的实例 *tz* 必须在此情形中保持一致：

```
tz.utcoffset(dt) - tz.dst(dt)
```

必须为具有同样的*tzinfo*子类实例 `dt.tzinfo == tz` 的每个*datetime*对象 *dt* 返回同样的结果，此表达式会产生时区的“标准时差”，它不应取决于具体日期或时间，只取决于地理位置。*datetime.astimezone()* 的实现依赖此方法，但无法检测违反规则的情况；确保符合规则是程序员的责任。如果一个*tzinfo*子类不能保证这一点，也许应该重载*tzinfo.fromutc()*的默认实现以便在任何情况下与 `astimezone()` 配合正常。

大多数*dst()*的实现可能会如以下两者之一:

```
def dst(self, dt):
    # a fixed-offset class:  doesn't account for DST
    return timedelta(0)
```

或者

```
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and expressed
    # in standard local time.  Then

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

默认的*dst()*实现会引发*NotImplementedError*。

`tzinfo.`**`tzname`**`(`*self*, *dt*`)`

将对应于*datetime*对象 *dt* 的时区名称作为字符串返回。*datetime*模块没有定义任何字符串名称相关内容，也不要求名称有任何特定含义。例如，” GMT”，“UTC”，“-500”，“-5:00”，“EDT”，“US/Eastern”，“America/New York”都是有效的返回值。如果字符串名称未知则返回 `None`。请注意这是一个方法而不是一个固定的字符串，这主要是因为某些*tzinfo*子类可能需要根据所传入的特定 *dt* 值返回不同的名称，特别是当*tzinfo*类要负责处理夏令时的时候。

默认的*tzname()*实现会引发*NotImplementedError*。

这些方法会被*datetime*或*time*对象调用，用来对应它们的同名方法。*datetime*对象会将自身作为传入参数，而*time*对象会将 `None` 作为传入参数。这样*tzinfo*子类的方法应当准备好接受 *dt* 参数值为 `None` 或是*datetime*类的实例。

当传入 `None` 时，应当由类的设计者来决定最佳回应方式。例如，返回 `None` 适用于希望该类提示时间对象不参与*tzinfo*协议处理。让 `utcoffset(None)` 返回标准 UTC 时差也许会更有用处，如果没有其他用于发现标准时差的约定。

当传入一个*datetime*对象来回应*datetime*方法时，`dt.tzinfo` 与 *self* 是同一对象。*tzinfo*方法可以依赖这一点，除非用户代码直接调用了*tzinfo*方法。此行为的目的是使得*tzinfo*方法将 *dt* 解读为本地时间，而不需要担心其他时区的相关对象。

还有一个额外的*tzinfo*方法，某个子类可能会希望重载它:

`tzinfo.`**`fromutc`**`(`*self*, *dt*`)`

此方法会由默认的*datetime.astimezone()*实现来调用。当被调用时，`dt.tzinfo` 为 *self*，并且 *dt* 的日期和时间数据会被视为代表 UTC 时间。*fromutc()*的目标是调整日期和时间数据，返回一个等价的 datetime 来表示 *self* 的本地时间。

大多数*tzinfo*子类应该能够毫无问题地继承默认的*fromutc()*实现。它的健壮性足以处理固定差值的时区以及同时负责标准时和夏令时的时区，对于后者甚至还能处理 DST 转换时间在各个年份有变化的情况。一个默认*fromutc()*实现可能无法在所有情况下正确处理的例子是（与 UTC 的）标准差值取决于所经过的特定日期和时间，这种情况可能由于政治原因而出现。默认的 `astimezone()` 和*fromutc()*实现可能无法生成你希望的结果，如果这个结果恰好是跨越了标准差值发生改变的时刻当中的某个小时值的话。

忽略针对错误情况的代码，默认*fromutc()* 实现的行为方式如下:

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta   # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

Example *tzinfo* classes:

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)

# A UTC class.

class UTC(tzinfo):
    """UTC"""

    def utcoffset(self, dt):
        return ZERO

    def tzname(self, dt):
        return "UTC"

    def dst(self, dt):
        return ZERO

utc = UTC()

# A class building tzinfo objects for fixed-offset time zones.
# Note that FixedOffset(0, "UTC") is a different way to build a
# UTC tzinfo object.

class FixedOffset(tzinfo):
    """Fixed offset in minutes east from UTC."""

    def __init__(self, offset, name):
        self.__offset = timedelta(minutes = offset)
        self.__name = name

    def utcoffset(self, dt):
        return self.__offset

    def tzname(self, dt):
        return self.__name

    def dst(self, dt):
```

(下页继续)

(续上页)

```
        return ZERO

# A class capturing the platform's idea of local time.

import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO

    def tzname(self, dt):
        return _time.tzname[self._isdst(dt)]

    def _isdst(self, dt):
        tt = (dt.year, dt.month, dt.day,
              dt.hour, dt.minute, dt.second,
              dt.weekday(), 0, 0)
        stamp = _time.mktime(tt)
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()


# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt


# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of rules for US
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
```

(下页继续)

(续上页)

```
# http://www.twinsun.com/tz/tz-link.htm
# http://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time; 1am standard time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 1)
# From 1987 to 2006, DST used to start at 2am (standard time) on the first
# Sunday in April and to end at 2am (DST time; 1am standard time) on the last
# Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 1)
# From 1967 to 1986, DST used to start at 2am (standard time) on the last
# Sunday in April (the one on or after April 24) and to end at 2am (DST time;
# 1am standard time) on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006

class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both cases.
            # It depends on how you want to treat them.  The default
            # fromutc() implementation (called by the default astimezone()
            # implementation) passes a datetime with dt.tzinfo is self.
            return ZERO
        assert dt.tzinfo is self

        # Find start and end times for US DST. For years before 1967, return
        # ZERO for no DST.
        if 2006 < dt.year:
            dststart, dstend = DSTSTART_2007, DSTEND_2007
        elif 1986 < dt.year < 2007:
            dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
        elif 1966 < dt.year < 1987:
```

(下页继续)

(续上页)

```
            dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
        else:
            return ZERO

        start = first_sunday_on_or_after(dststart.replace(year=dt.year))
        end = first_sunday_on_or_after(dstend.replace(year=dt.year))

        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        if start <= dt.replace(tzinfo=None) < end:
            return HOUR
        else:
            return ZERO

Eastern  = USTimeZone(-5, "Eastern",  "EST", "EDT")
Central  = USTimeZone(-6, "Central",  "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific  = USTimeZone(-8, "Pacific",  "PST", "PDT")
```

请注意同时负责标准时和夏令时的*tzinfo*子类在每年两次的 DST 转换点上会出现不可避免的微妙问题。具体而言，考虑美国东部时区 (UTC -0500)，它的 EDT 从三月的第二个星期天 1:59 (EST) 之后一分钟开始，并在十一月的第一个星期天 1:59 (EDT) 之后一分钟结束:

```
  UTC    3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
  EST   22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
  EDT   23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start   22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

  end   23:MM  0:MM  1:MM  1:MM  2:MM  3:MM
```

When DST starts (the "start" line), the local wall clock leaps from 1:59 to 3:00. A wall time of the form 2:MM doesn't really make sense on that day, so `astimezone(Eastern)` won't deliver a result with `hour == 2` on the day DST begins. In order for `astimezone()` to make this guarantee, the `rzinfo.dst()` method must consider times in the "missing hour" (2:MM for Eastern) to be in daylight time.

When DST ends (the "end" line), there's a potentially worse problem: there's an hour that can't be spelled unambiguously in local wall time: the last hour of daylight time. In Eastern, that's times of the form 5:MM UTC on the day daylight time ends. The local wall clock leaps from 1:59 (daylight time) back to 1:00 (standard time) again. Local times of the form 1:MM are ambiguous. `astimezone()` mimics the local clock's behavior by mapping two adjacent UTC hours into the same local hour then. In the Eastern example, UTC times of the form 5:MM and 6:MM both map to 1:MM when converted to Eastern. In order for `astimezone()` to make this guarantee, the *tzinfo.dst()* method must consider times in the "repeated hour" to be in standard time. This is easily arranged, as in the example, by expressing DST switch times in the time zone's standard local time.

Applications that can't bear such ambiguities should avoid using hybrid *tzinfo* subclasses; there are no ambiguities when using UTC, or any other fixed-offset *tzinfo* subclass (such as a class representing only EST (fixed offset -5 hours), or only EDT (fixed offset -4 hours)).

参见:

pytz The standard library has no *tzinfo* instances, but there exists a third-party library which brings the *IANA timezone database* (also known as the Olson database) to Python: *pytz*.

pytz contains up-to-date information and its usage is recommended.

IANA 时区数据库 The Time Zone Database (often called tz or zoneinfo) contains code and data that represent the history of local time for many representative locations around the globe. It is updated periodically to reflect changes

made by political bodies to time zone boundaries, UTC offsets, and daylight-saving rules.

8.1.7 strftime() 和 strptime() 的行为

date, *datetime* 和*time* 对象都支持 strftime(format) 方法，可用来创建一个由指定格式字符串所控制的表示时间的字符串。总体而言，d.strftime(fmt) 类似于*time* 模块的 time.strftime(fmt, d.timetuple()) 但是并非所有对象都支持 timetuple() 方法。

相反地，*datetime.strptime()* 类方法可根据一个表示时间的字符串和对应的格式字符串创建来一个*datetime* 对象。datetime.strptime(date_string, format) 等价于 datetime(*(time.strptime(date_string, format)[0:6]))，差别在于当 format 包含小于秒的部分或者时区差值信息的时候，这些信息被 datetime.strptime 所支持但会被 time.strptime 所丢弃。

对于*time* 对象，年、月、日的格式代码不应被使用，因为 time 对象没有这些值。如果它们被使用，年份将被替换为 1900 而月和日将被替换为 1。

对于*date* 对象，时、分、秒和微秒的格式代码不应被使用，因为*date* 对象没有这些值。如果它们被使用，它们都将被替换为 0。

对完整格式代码集的支持在不同平台上有所差异，因为 Python 要调用所在平台 C 库的 strftime() 函数，而不同平台的差异是很常见的。要查看你所用平台所支持的完整格式代码集，请参阅 *strftime(3)* 文档。

出于相同的原因，对于包含当前区域设置字符集所无法表示的 Unicode 码位的格式字符串的处理方式也取决于具体平台。在某些平台上这样的码位会不加修改地原样输出，而在其他平台上 strftime 则可能引发*UnicodeError* 或只返回一个空字符串。

以下列表显示了 C 标准（1989 版）所要求的全部格式代码，它们在带有标准 C 实现的所有平台上均可用。请注意 1999 版 C 标准又添加了额外的格式代码。

The exact range of years for which strftime() works also varies across platforms. Regardless of platform, years before 1900 cannot be used.

指令	含义	示例	注释
%a	当地工作日的缩写。	Sun, Mon, …, Sat (美国); So, Mo, …, Sa (德国)	(1)
%A	当地工作日的全名。	Sunday, Monday, …, Saturday (美国); Sonntag, Montag, …, Samstag (德国)	(1)
%w	以十进制数显示的工作日，其中 0 表示星期日，6 表示星期六。	0, 1, …, 6	
%d	补零后，以十进制数显示的月份中的一天。	01, 02, …, 31	
%b	当地月份的缩写。	Jan, Feb, …, Dec (美国); Jan, Feb, …, Dez (德国)	(1)
%B	当地月份的全名。	January, February, …, December (美国); Januar, Februar, …, Dezember (德国)	(1)
%m	补零后，以十进制数显示的月份。	01, 02, …, 12	
%y	补零后，以十进制数表示的，不带世纪的年份。	00, 01, …, 99	
%Y	十进制数表示的带世纪的年份。	1970, 1988, 2001, 2013	
%H	以补零后的十进制数表示的小时（24 小时制）。	00, 01, …, 23	
%I	以补零后的十进制数表示的小时（12 小时制）。	01, 02, …, 12	
%p	本地化的 AM 或 PM 。	AM, PM (美国); am, pm (德国)	(1), (2)
%M	补零后，以十进制数显示的分钟。	00, 01, …, 59	
%S	补零后，以十进制数显示的秒。	00, 01, …, 59	(3)
%f	以十进制数表示的毫秒，在左侧补零。	000000, 000001, …, 999999	(4)
%z	UTC offset in the form +HHMM or -HHMM (empty string if the the object is naive).	(empty), +0000, -0400, +1030	(5)
%Z	时区名称（如果对象为简单型则为空字符串）。	(空), UTC, EST, CST	
%j	以补零后的十进制数表示的一年中的日序号。	001, 002, …, 366	
%U	以补零后的十进制数	00, 01, …, 53	(6)

注释:

(1) 由于此格式依赖于当前区域设置，因此对具体输出值应当保持谨慎预期。字段顺序会发生改变（例如“month/day/year”与“day/month/year”），并且输出可能包含使用区域设置所指定的默认编码格式的 Unicode 字符（例如如果当前区域为 `ja_JP`，则默认编码格式可能为 `eucJP`, `SJIS` 或 `utf-8` 中的一个；使用*`locale.getlocale()`* 可确定当前区域设置的编码格式）。

(2) 当与 `strptime()` 方法一起使用时，如果使用 `%I` 指令来解析小时，`%p` 指令只影响输出小时字段。

(3) 与*`time`* 模块不同的是，*`datetime`* 模块不支持闰秒。

(4) `%f` is an extension to the set of format characters in the C standard (but implemented separately in datetime objects, and therefore always available). When used with the `strptime()` method, the `%f` directive accepts from one to six digits and zero pads on the right.

2.6 新版功能.

(5) 对于简单型对象，`%z` and `%Z` 格式代码会被替换为空字符串。

对于一个觉悟型对象而言:

`%z` `utcoffset()` is transformed into a 5-character string of the form +HHMM or -HHMM, where HH is a 2-digit string giving the number of UTC offset hours, and MM is a 2-digit string giving the number of UTC offset minutes. For example, if `utcoffset()` returns `timedelta(hours=-3, minutes=-30)`, `%z` is replaced with the string `'-0330'`.

`%Z` 如果 `tzname()` 返回 `None`，`%Z` 会被替换为一个空字符串。在其他情况下 `%Z` 会被替换为返回值，这必须为一个字符串。

(6) When used with the `strptime()` method, `%U` and `%W` are only used in calculations when the day of the week and the year are specified.

备注

8.2 `calendar` —日历相关函数

源代码: Lib/calendar.py

这个模块让你可以输出像 Unix **`cal`** 那样的日历，它还提供了其它与日历相关的实用函数。默认情况下，这些日历把星期一当作一周的第一天，星期天为一周的最后一天（按照欧洲惯例）。可以使用*`setfirstweekday()`* 方法设置一周的第一天为星期天 (6) 或者其它任意一天。使用整数作为指定日期的参数。更多相关的函数，参见*`datetime`* 和*`time`* 模块。

Most of these functions and classes rely on the *`datetime`* module which uses an idealized calendar, the current Gregorian calendar indefinitely extended in both directions. This matches the definition of the “proleptic Gregorian” calendar in Dershowitz and Reingold’s book “Calendrical Calculations”, where it’s the base calendar for all computations.

`class` `calendar.`**`Calendar`**`(`[*firstweekday*]`)`

创建一个*`Calendar`* 对象。*firstweekday* 是一个整数，用于指定一周的第一天。`0` 是星期一（默认值），`6` 是星期天。

`Calendar` 对象提供了一些可被用于准备日历数据格式化的方法。这个类本身不执行任何格式化操作。这部分任务应由子类来完成。

2.5 新版功能.

`Calendar` 类的实例有下列方法:

iterweekdays()
返回一个迭代器，迭代器的内容为一星期的数字。迭代器的第一个值与`firstweekday`属性的值一至。

itermonthdates(*year*, *month*)
返回一个迭代器，迭代器的内容为 *year* 年 *month* 月 (1-12) 的日期。这个迭代器返回当月的所有日期 (`datetime.date` 对象)，日期包含了本月头尾用于组成完整一周的日期。

itermonthdays2(*year*, *month*)
Return an iterator for the month *month* in the year *year* similar to `itermonthdates()`. Days returned will be tuples consisting of a day number and a week day number.

itermonthdays(*year*, *month*)
Return an iterator for the month *month* in the year *year* similar to `itermonthdates()`. Days returned will simply be day numbers.

monthdatescalendar(*year*, *month*)
返回一个表示指定年月的周列表。周列表由七个`datetime.date`对象组成。

monthdays2calendar(*year*, *month*)
返回一个表示指定年月的周列表。周列表由七个代表日期的数字和代表周几的数字组成的二元元组。

monthdayscalendar(*year*, *month*)
返回一个表示指定年月的周列表。周列表由七个代表日期的数字组成。

yeardatescalendar(*year*[, *width*])
返回可以用来格式化的指定年月的数据。返回的值是一个列表，列表是月份组成的行。每一行包含了最多 *width* 个月 (默认为 3)。每个月包含了 4 到 6 周，每周包含 1–7 天。每一天使用`datetime.date`对象。

yeardays2calendar(*year*[, *width*])
返回可以用来模式化的指定年月的数据 (与`yeardatescalendar()` 类似)。周列表的元素是由表示日期的数字和表示星期几的数字组成的元组。不在这个月的日子为 0。

yeardayscalendar(*year*[, *width*])
返回可以用来模式化的指定年月的数据 (与`yeardatescalendar()` 类似)。周列表的元素是表示日期的数字。不在这个月的日子为 0。

class calendar.**TextCalendar**([*firstweekday*])
可以使用这个类生成纯文本日历。

2.5 新版功能.

`TextCalendar` 实例有以下方法：

formatmonth(*theyear*, *themonth*[, *w*[, *l*]])
返回一个多行字符串来表示指定年月的日历。*w* 为日期的宽度，但始终保持日期居中。*l* 指定了每星期占用的行数。以上这些还依赖于构造器或者`setfirstweekday()` 方法指定的周的第一天是哪一天。

prmonth(*theyear*, *themonth*[, *w*[, *l*]])
与`formatmonth()` 方法一样，返回一个月的日历。

formatyear(*theyear*[, *w*[, *l*[, *c*[, *m*]]]])
返回一个多行字符串，这个字符串为一个 *m* 列日历。可选参数 *w*, *l*, 和 *c* 分别表示日期列数，周的行数，和月之间的间隔。同样，以上这些还依赖于构造器或者`setfirstweekday()` 指定哪一天为一周的第一天。日历的第一年由平台依赖于使用的平台。

pryear(*theyear*[, *w*[, *l*[, *c*[, *m*]]]])
与`formatyear()` 方法一样，返回一整年的日历。

class calendar.**HTMLCalendar**([*firstweekday*])
可以使用这个类生成 HTML 日历。

2.5 新版功能.

HTMLCalendar instances have the following methods:

formatmonth(*theyear*, *themonth*[, *withyear*])
返回一个 HTML 表格作为指定年月的日历。*withyear* 为真，则年份将会包含在表头，否则只显示月份。

formatyear(*theyear*[, *width*])
返回一个 HTML 表格作为指定年份的日历。*width* (默认为 3) 用于规定每一行显示月份的数量。

formatyearpage(*theyear*[, *width*[, *css*[, *encoding*]]])
返回一个完整的 HTML 页面作为指定年份的日历。*width**(默认为 3) 用于规定每一行显示的月份数量。**css* 为层叠样式表的名字。如果不使用任何层叠样式表，可以使用*None* 。*encoding* 为输出页面的编码 (默认为系统的默认编码)。

class calendar.**LocaleTextCalendar**([*firstweekday*[, *locale*]])
这个子类*TextCalendar* 可以在构造函数中传递一个语言环境名称，并且返回月份和星期几的名称在特定语言环境中。如果此语言环境包含编码，则包含月份和工作日名称的所有字符串将作为 unicode 返回。

2.5 新版功能.

class calendar.**LocaleHTMLCalendar**([*firstweekday*[, *locale*]])
This subclass of *HTMLCalendar* can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

2.5 新版功能.

注解: 这两个类的 `formatweekday()` 和 `formatmonthname()` 方法临时更改 dang 当前区域至给定 *locale* 。由于当前的区域设置是进程范围的设置，因此它们不是线程安全的。

对于简单的文本日历，这个模块提供了以下方法。

calendar.**setfirstweekday**(*weekday*)
设置每一周的开始 (`0` 表示星期一，`6` 表示星期天)。calendar 还提供了 `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` 和 `SUNDAY` 几个常量方便使用。例如，设置每周的第一天为星期天

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

2.0 新版功能.

calendar.**firstweekday**()
返回当前设置的每星期的第一天的数值。

2.0 新版功能.

calendar.**isleap**(*year*)
如果 *year* 是闰年则返回*True* , 否则返回*False*。

calendar.**leapdays**(*y1*, *y2*)
返回在范围 *y1* 至 *y2* （包含 y1 和 y2 ）之间的闰年的年数，其中 *y1* 和 *y2* 是年份。

在 2.0 版更改: This function didn’t work for ranges spanning a century change in Python 1.5.2.

calendar.**weekday**(*year*, *month*, *day*)
返回一周中的某一天（0 是周一）以年（1970–…），月（1–12），日（1–31）的格式。

calendar.**weekheader**(*n*)
返回一个包含星期几的缩写名的头。*n* 指定星期几缩写的字符宽度。

calendar.**monthrange**(*year*, *month*)
返回指定年份的指定月份第一天是星期几和这个月的天数。

calendar.**monthcalendar**(*year*, *month*)
返回表示一个月的日历的矩阵。每一行代表一周；此月份外的日子由零表示。每周从周一开始，除非使用*setfirstweekday()* 改变设置。

calendar.**prmonth**(*theyear*, *themonth*[, *w*[, *l*]])
打印由*month()* 返回的一个月的日历。

calendar.**month**(*theyear*, *themonth*[, *w*[, *l*]])
使用*TextCalendar* 类的 formatmonth() 以多行字符串形式返回月份日历。

2.0 新版功能.

calendar.**prcal**(*year[, w[, l[c]]]*)
打印由*calendar()* 返回的整年的日历。

calendar.**calendar**(*year[, w[, l[c]]]*)
使用*TextCalendar* 类的 formatyear() 返回整年的三列的日历以多行字符串的形式。

2.0 新版功能.

calendar.**timegm**(*tuple*)
一个不相关但很好用的函数，它接受一个时间元组例如*time* 模块中的*gmtime()* 函数的返回并返回相应的 Unix 时间戳，假定 1970 年开始计数，POSIX 编码。实际上，*time.gmtime()* 和*timegm()* 是彼此相反的。

2.0 新版功能.

calendar 模块导出以下数据属性：

calendar.**day_name**
在当前的语言环境下表示星期几。

calendar.**day_abbr**
在当前语言环境下表示星期几缩写的数组。

calendar.**month_name**
在当前语言环境下表示一年中月份的数组。这遵循一月的月号为 1 的通常惯例，所以它的长度为 13 且 month_name[0] 是空字符串。

calendar.**month_abbr**
在当前语言环境下表示月份简写的数组。这遵循一月的月号为 1 的通常惯例，所以它的长度为 13 且 month_abbr[0] 是空字符串。

参见:

模块*datetime* 为日期和时间提供与*time* 模块相似功能的面向对象接口。

模块*time* 底层时间相关函数。

8.3 collections —High-performance container datatypes

2.4 新版功能.

Source code: Lib/collections.py and Lib/_abcoll.py

这个模块实现了特定目标的容器，以提供 Python 标准内建容器*dict*，list，*set*，和*tuple*的替代选择。

namedtuple()	创建命名元组子类的工厂函数	2.6 新版功能.
deque	类似列表 (list) 的容器，实现了在两端快速添加 (append) 和弹出 (pop)	2.4 新版功能.
Counter	字典的子类，提供了可哈希对象的计数功能	2.7 新版功能.
OrderedDict	字典的子类，保存了他们被添加的顺序	2.7 新版功能.
defaultdict	字典的子类，提供了一个工厂函数，为字典查询提供一个默认值	2.5 新版功能.

In addition to the concrete container classes, the collections module provides *abstract base classes* that can be used to test whether a class provides a particular interface, for example, whether it is hashable or a mapping.

8.3.1 Counter 对象

一个计数器工具提供快速和方便的计数。比如

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class collections.**Counter**([*iterable-or-mapping*])

A *Counter* is a *dict* subclass for counting hashable objects. It is an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The *Counter* class is similar to bags or multisets in other languages.

元素从一个 *iterable* 被计数或从其他的 *mapping* (or counter) 初始化：

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter from keyword args
```

Counter 对象有一个字典接口，如果引用的键没有任何记录，就返回一个 0，而不是弹出一个*KeyError*：

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a missing element is zero
0
```

设置一个计数为 0 不会从计数器中移去一个元素。使用 `del` 来删除它:

```
>>> c['sausage'] = 0                    # counter entry with a zero count
>>> del c['sausage']                    # del actually removes the entry
```

2.7 新版功能.

计数器对象除了字典方法以外，还提供了三个其他的方法:

elements()
返回一个迭代器，每个元素重复计数的个数。元素顺序是任意的。如果一个元素的计数小于 1，*elements()* 就会忽略它。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([*n*])
Return a list of the *n* most common elements and their counts from the most common to the least. If *n* is omitted or `None`, *most_common()* returns *all* elements in the counter. Elements with equal counts are ordered arbitrarily:

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('r', 2), ('b', 2)]
```

subtract([*iterable-or-mapping*])
从 迭代对象或 映射对象减去元素。像*dict.update()* 但是是减去，而不是替换。输入和输出都可以是 0 或者负数。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

通常字典方法都可用于*Counter* 对象，除了有两个方法工作方式与字典并不相同。

fromkeys(*iterable*)
这个类方法没有在*Counter* 中实现。

update([*iterable-or-mapping*])
从 迭代对象计数元素或者从另一个 映射对象 (或计数器) 添加。像*dict.update()* 但是是加上，而不是替换。另外，迭代对象应该是序列元素，而不是一个 `(key, value)` 对。

Counter 对象的常用案例

```
sum(c.values())                 # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
dict(c)                         # convert to a regular dictionary
c.items()                       # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs))    # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]       # n least common elements
c += Counter()                  # remove zero and negative counts
```

提供了几个数学操作，可以结合*Counter* 对象，以生产 multisets (计数器中大于 0 的元素)。加和减，结合计数器，通过加上或者减去元素的相应计数。交集和并集返回相应计数的最小或最大值。每种操作都可以接受带符号的计数，但是输出会忽略掉结果为零或者小于零的计数。

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                       # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                       # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                       # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                       # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
```

注解: 计数器主要是为了表达运行的正的计数而设计；但是，小心不要预先排除负数或者其他类型。为了帮助这些用例，这一节记录了最小范围和类型限制。

- *Counter* 类是一个字典的子类，不限制键和值。值用于表示计数，但你实际上 可以存储任何其他值。
- The `most_common()` method requires only that the values be orderable.
- For in-place operations such as `c[key] += 1`, the value type need only support addition and subtraction. So fractions, floats, and decimals would work and negative values are supported. The same is also true for `update()` and `subtract()` which allow negative and zero values for both inputs and outputs.
- Multiset 多集合方法只为正值的使用情况设计。输入可以是负数或者 0，但只输出计数为正的值。没有类型限制，但值类型需要支持加，减和比较操作。
- The `elements()` method requires integer counts. It ignores zero and negative counts.

参见:

- Counter class adapted for Python 2.5 and an early Bag recipe for Python 2.4.
- Bag class 在 Smalltalk。
- Wikipedia 链接 Multisets.
- C++ multisets 教程和例子。
- 数学操作和多集合用例，参考 *Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19* 。
- To enumerate all distinct multisets of a given size over a given set of elements, see *itertools.combinations_with_replacement()*.

  map(Counter, combinations_with_replacement(‘ABC’, 2)) –> AA AB AC BB BC CC

8.3.2 deque 对象

class collections.**deque**([*iterable*[, *maxlen*]])

返回一个新的双向队列对象，从左到右初始化 (用方法*append()*)，从 *iterable* （迭代对象) 数据创建。如果 *iterable* 没有指定，新队列为空。

Deque 队列是由栈或者 queue 队列生成的（发音是 “deck”,” double-ended queue” 的简称)。Deque 支持线程安全，内存高效添加 (append) 和弹出 (pop)，从两端都可以，两个方向的大概开销都是 O(1) 复杂度。

虽然 `list` 对象也支持类似操作，不过这里优化了定长操作和 `pop(0)` 和 `insert(0, v)` 的开销。它们引起 O(n) 内存移动的操作，改变底层数据表达的大小和位置。

2.4 新版功能.

如果 *maxlen* 没有指定或者是 `None` ，deques 可以增长到任意长度。否则，deque 就限定到指定最大长度。一旦限定长度的 deque 满了，当新项加入时，同样数量的项就从另一端弹出。限定长度 deque 提供类似 Unix filter `tail` 的功能。它们同样可以用与追踪最近的交换和其他数据池活动。

在 2.6 版更改: Added *maxlen* parameter.

双向队列 (deque) 对象支持以下方法：

append(*x*)
添加 *x* 到右端。

appendleft(*x*)
添加 *x* 到左端。

clear()
移除所有元素，使其长度为 0.

count(*x*)
计算 deque 中元素等于 *x* 的个数。

2.7 新版功能.

extend(*iterable*)
扩展 deque 的右侧，通过添加 iterable 参数中的元素。

extendleft(*iterable*)
扩展 deque 的左侧，通过添加 iterable 参数中的元素。注意，左添加时，在结果中 iterable 参数中的顺序将被反过来添加。

pop()
移去并且返回一个元素，deque 最右侧的那一个。如果没有元素的话，就引发一个*IndexError*。

popleft()
移去并且返回一个元素，deque 最左侧的那一个。如果没有元素的话，就引发*IndexError*。

remove(*value*)
移除找到的第一个 *value*。如果没有的话就引发*ValueError*。

2.5 新版功能.

reverse()
将 deque 逆序排列。返回 `None` 。

2.7 新版功能.

rotate(*n=1*)
向右循环移动 *n* 步。如果 *n* 是负数，就向左循环。

如果 deque 不是空的，向右循环移动一步就等价于 `d.appendleft(d.pop())` ，向左循环一步就等价于 `d.append(d.popleft())` 。

Deque 对象同样提供了一个只读属性:

maxlen
Deque 的最大尺寸，如果没有限定的话就是 `None` 。

2.7 新版功能.

除了以上，deque 还支持迭代，清洗，`len(d)`, `reversed(d)`, `copy.copy(d)`, `copy.deepcopy(d)`, 成员测试 `in` 操作符，和下标引用 `d[-1]` 。索引存取在两端的复杂度是 O(1)，在中间的复杂度比 O(n) 略低。要快速存取，使用 list 来替代。

示例:

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print elem.upper()
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                             # peek at leftmost item
'g'
>>> d[-1]                            # peek at rightmost item
'i'

>>> list(reversed(d))                # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                         # search the deque
True
>>> d.extend('jkl')                  # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                      # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                     # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))               # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                        # empty the deque
>>> d.pop()                          # cannot pop from an empty deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')              # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

deque 用法

这一节展示了 deque 的多种用法。

限长 deque 提供了类似 Unix `tail` 过滤功能

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    return deque(open(filename), n)
```

另一个用法是维护一个近期添加元素的序列，通过从右边添加和从左边弹出

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / float(n)
```

The `rotate()` method provides a way to implement *deque* slicing and deletion. For example, a pure Python implementation of `del d[n]` relies on the `rotate()` method to position elements to be popped:

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

To implement *deque* slicing, use a similar approach applying `rotate()` to bring a target element to the left side of the deque. Remove old entries with `popleft()`, add new entries with `extend()`, and then reverse the rotation. With minor variations on that approach, it is easy to implement Forth style stack manipulations such as `dup`, `drop`, `swap`, `over`, `pick`, `rot`, and `roll`.

8.3.3 defaultdict 对象

class collections.**defaultdict**([*default_factory*[, ...]])

返回一个新的类似字典的对象。*defaultdict* 是内置*dict* 类的子类。它重载了一个方法并添加了一个可写的实例变量。其余的功能与*dict* 类相同，此处不再重复说明。

第一个参数*default_factory* 提供了一个初始值。它默认为 `None` 。所有的其他参数都等同与*dict* 构建器中的参数对待，包括关键词参数。

2.5 新版功能.

defaultdict 对象除了支持*dict* 的操作，还支持下面的方法作为扩展:

__missing__(*key*)

如果*default_factory* 属性为 `None`，则调用本方法会抛出*KeyError* 异常，附带参数 *key*。

如果*default_factory* 不为 `None` ，它就会会被调用，不带参数，为 *key* 提供一个默认值，这个值和 *key* 作为一个对被插入到字典中，并返回。

如果调用*default_factory* 时抛出了异常，这个异常会原封不动地向外层传递。

在无法找到所需键值时，本方法会被*dict* 中的 __getitem__() 方法调用。无论本方法返回了值还是抛出了异常，都会被 __getitem__() 传递。

注意*__missing__()* 不会被 __getitem__() 以外的其他方法调用。意思就是 get() 会向正常的 dict 那样返回 None ，而不是使用*default_factory* 。

defaultdict 支持以下实例变量:

default_factory
这个属性被*__missing__()* 方法使用；它从构建器的第一个参数初始化，如果提供了的话，否则就是 None 。

defaultdict 例子

Using `list` as the `default_factory`, it is easy to group a sequence of key-value pairs into a dictionary of lists:

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

When each key is encountered for the first time, it is not already in the mapping; so an entry is automatically created using the `default_factory` function which returns an empty `list`. The `list.append()` operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the `list.append()` operation adds another value to the list. This technique is simpler and faster than an equivalent technique using *dict.setdefault()*:

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> d.items()
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Setting the `default_factory` to *int* makes the *defaultdict* useful for counting (like a bag or multiset in other languages):

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> d.items()
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]
```

When a letter is first encountered, it is missing from the mapping, so the `default_factory` function calls *int()* to supply a default count of zero. The increment operation then builds up the count for each letter.

The function *int()* which always returns zero is just a special case of constant functions. A faster and more flexible way to create constant functions is to use *itertools.repeat()* which can supply any constant value (not just zero):

```
>>> def constant_factory(value):
...     return itertools.repeat(value).next
>>> d = defaultdict(constant_factory('<missing>'))
```

(下页继续)

(续上页)

```
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

Setting the `default_factory` to *set* makes the *defaultdict* useful for building a dictionary of sets:

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> d.items()
[('blue', set([2, 4])), ('red', set([1, 3]))]
```

8.3.4 namedtuple() 命名元组的工厂函数

命名元组赋予每个位置一个含义，提供可读性和自文档性。它们可以用于任何普通元组，并添加了通过名字获取值的能力，通过索引值也是可以的。

`collections.`**`namedtuple`**(*typename*, *field_names*[, *verbose=False*][, *rename=False*])

返回一个新的元组子类，名为 *typename* 。这个新的子类用于创建类元组的对象，可以通过域名来获取属性值，同样也可以通过索引和迭代获取值。子类实例同样有文档字符串（类名和域名）另外一个有用的 `__repr__()` 方法，以 `name=value` 格式列明了元组内容。

field_names 是一个像 `['x', 'y']` 一样的字符串序列。另外 *field_names* 可以是一个纯字符串，用空白或逗号分隔开元素名，比如 `'x y'` 或者 `'x, y'` 。

Any valid Python identifier may be used for a fieldname except for names starting with an underscore. Valid identifiers consist of letters, digits, and underscores but do not start with a digit or underscore and cannot be a *keyword* such as *class*, *for*, *return*, *global*, *pass*, *print*, or *raise*.

如果 *rename* 为真，无效域名会自动转换成位置名。比如 `['abc', 'def', 'ghi', 'abc']` 转换成 `['abc', '_1', 'ghi', '_3']` ，消除关键词 `def` 和重复域名 `abc` 。

If *verbose* is true, the class definition is printed just before being built.

命名元组实例没有字典，所以它们要更轻量，并且占用更小内存。

2.6 新版功能.

在 2.7 版更改: added support for *rename*.

示例:

```
>>> Point = namedtuple('Point', ['x', 'y'], verbose=True)
class Point(tuple):
    'Point(x, y)'

    __slots__ = ()

    _fields = ('x', 'y')

    def __new__(_cls, x, y):
        'Create new instance of Point(x, y)'
        return _tuple.__new__(_cls, (x, y))

    @classmethod
```

(下页继续)

(续上页)

```
    def _make(cls, iterable, new=tuple.__new__, len=len):
        'Make a new Point object from a sequence or iterable'
        result = new(cls, iterable)
        if len(result) != 2:
            raise TypeError('Expected 2 arguments, got %d' % len(result))
        return result

    def __repr__(self):
        'Return a nicely formatted representation string'
        return 'Point(x=%r, y=%r)' % self

    def _asdict(self):
        'Return a new OrderedDict which maps field names to their values'
        return OrderedDict(zip(self._fields, self))

    def _replace(_self, **kwds):
        'Return a new Point object replacing specified fields with new values'
        result = _self._make(map(kwds.pop, ('x', 'y'), _self))
        if kwds:
            raise ValueError('Got unexpected field names: %r' % kwds.keys())
        return result

    def __getnewargs__(self):
        'Return self as a plain tuple.  Used by copy and pickle.'
        return tuple(self)

    __dict__ = _property(_asdict)

    def __getstate__(self):
        'Exclude the OrderedDict from pickling'
        pass

    x = _property(_itemgetter(0), doc='Alias for field number 0')

    y = _property(_itemgetter(1), doc='Alias for field number 1')



>>> p = Point(11, y=22)     # instantiate with positional or keyword arguments
>>> p[0] + p[1]             # indexable like the plain tuple (11, 22)
33
>>> x, y = p                # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y               # fields also accessible by name
33
>>> p                       # readable __repr__ with a name=value style
Point(x=11, y=22)
```

命名元组尤其有用于赋值*csv* *sqlite3* 模块返回的元组

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade
↪')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
```

(下页继续)

(续上页)

```
    print emp.name, emp.title

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print emp.name, emp.title
```

In addition to the methods inherited from tuples, named tuples support three additional methods and one attribute. To prevent conflicts with field names, the method and attribute names start with an underscore.

classmethod somenamedtuple.**_make**(*iterable*)
类方法从存在的序列或迭代实例创建一个新实例。

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

somenamedtuple.**_asdict**()
Return a new *OrderedDict* which maps field names to their corresponding values:

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
OrderedDict([('x', 11), ('y', 22)])
```

在 2.7 版更改: 返回一个*OrderedDict* 而不是*dict* 。

somenamedtuple.**_replace**(***kwargs*)
返回一个新的命名元组实例，并将指定域替换为新的值

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum],␣
↪timestamp=time.now())
```

somenamedtuple.**_fields**
字符串元组列出了域名。用于提醒和从现有元组创建一个新的命名元组类型。

```
>>> p._fields            # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

要获取这个名字域的值，使用*getattr()* 函数:

```
>>> getattr(p, 'x')
11
```

To convert a dictionary to a named tuple, use the double-star-operator (as described in tut-unpacking-arguments):

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

因为一个命名元组是一个正常的 Python 类，它可以很容易的通过子类更改功能。这里是如何添加一个计算域和定宽输出打印格式:

```
>>> class Point(namedtuple('Point', 'x y')):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f' % (self.x, self.y, self.
↪hypot)
...
>>> for p in Point(3, 4), Point(14, 5/7.):
...     print p
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

上面的子类设置 `__slots__` 为一个空元组。通过阻止创建实例字典保持了较低的内存开销。

Subclassing is not useful for adding new, stored fields. Instead, simply create a new named tuple type from the `_fields` attribute:

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

Default values can be implemented by using `_replace()` to customize a prototype instance:

```
>>> Account = namedtuple('Account', 'owner balance transaction_count')
>>> default_account = Account('<owner name>', 0.0, 0)
>>> johns_account = default_account._replace(owner='John')
```

Enumerated constants can be implemented with named tuples, but it is simpler and more efficient to use a simple class declaration:

```
>>> Status = namedtuple('Status', 'open pending closed')._make(range(3))
>>> Status.open, Status.pending, Status.closed
(0, 1, 2)
>>> class Status:
...     open, pending, closed = range(3)
```

参见:

Named tuple recipe adapted for Python 2.4.

8.3.5 OrderedDict 对象

Ordered dictionaries are just like regular dictionaries but they remember the order that items were inserted. When iterating over an ordered dictionary, the items are returned in the order their keys were first added.

class collections.**OrderedDict**([*items*])

Return an instance of a dict subclass, supporting the usual *dict* methods. An *OrderedDict* is a dict that remembers the order that keys were first inserted. If a new entry overwrites an existing entry, the original insertion position is left unchanged. Deleting an entry and reinserting it will move it to the end.

2.7 新版功能.

OrderedDict.**popitem**(*last=True*)

The *popitem()* method for ordered dictionaries returns and removes a (key, value) pair. The pairs are returned in LIFO order if *last* is true or FIFO order if false.

相对于通常的映射方法，有序字典还另外提供了逆序迭代的支持，通过*reversed()*。

Equality tests between *OrderedDict* objects are order-sensitive and are implemented as `list(od1.items())==list(od2.items())`. Equality tests between *OrderedDict* objects and other *Mapping* objects are order-insensitive like regular dictionaries. This allows *OrderedDict* objects to be substituted anywhere a regular dictionary is used.

The *OrderedDict* constructor and `update()` method both accept keyword arguments, but their order is lost because Python's function call semantics pass-in keyword arguments using a regular unordered dictionary.

参见:

Equivalent OrderedDict recipe that runs on Python 2.4 or later.

OrderedDict 例子和用法

Since an ordered dictionary remembers its insertion order, it can be used in conjunction with sorting to make a sorted dictionary:

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

The new sorted dictionaries maintain their sort order when entries are deleted. But when new keys are added, the keys are appended to the end and the sort is not maintained.

It is also straight-forward to create an ordered dictionary variant that remembers the order the keys were *last* inserted. If a new entry overwrites an existing entry, the original insertion position is changed and moved to the end:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'
```

(下页继续)

(续上页)

```
    def __setitem__(self, key, value):
        if key in self:
            del self[key]
        OrderedDict.__setitem__(self, key, value)
```

An ordered dictionary can be combined with the *Counter* class so that the counter remembers the order elements are first encountered:

```
class OrderedCounter(Counter, OrderedDict):
     'Counter that remembers the order elements are first encountered'

     def __repr__(self):
         return '%s(%r)' % (self.__class__.__name__, OrderedDict(self))

     def __reduce__(self):
         return self.__class__, (OrderedDict(self),)
```

8.3.6 Collections Abstract Base Classes

The collections module offers the following *ABCs*:

ABC	Inherits from	Abstract Methods	Mixin Methods
Container		`__contains__`	
Hashable		`__hash__`	
Iterable		`__iter__`	
Iterator	*Iterable*	`next`	`__iter__`
Sized		`__len__`	
Callable		`__call__`	
Sequence	*Sized*, *Iterable*, *Container*	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, and `count`
MutableSequence	*Sequence*	`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`	Inherited *Sequence* methods and `append`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`
Set	*Sized*, *Iterable*, *Container*	`__contains__`, `__iter__`, `__len__`	`__le__`, `__lt__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__and__`, `__or__`, `__sub__`, `__xor__`, and `isdisjoint`
MutableSet	*Set*	`__contains__`, `__iter__`, `__len__`, `add`, `discard`	Inherited *Set* methods and `clear`, `pop`, `remove`, `__ior__`, `__iand__`, `__ixor__`, and `__isub__`
Mapping	*Sized*, *Iterable*, *Container*	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__`, and `__ne__`
MutableMapping	*Mapping*	`__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`	Inherited *Mapping* methods and `pop`, `popitem`, `clear`, `update`, and `setdefault`
MappingView	*Sized*		`__len__`
ItemsView	*MappingView*, *Set*		`__contains__`, `__iter__`
KeysView	*MappingView*, *Set*		`__contains__`, `__iter__`
ValuesView	*MappingView*		`__contains__`, `__iter__`

class collections.**Container**
class collections.**Hashable**
class collections.**Sized**
class collections.**Callable**
ABCs for classes that provide respectively the methods `__contains__()`, `__hash__()`, `__len__()`, and `__call__()`.

class collections.**Iterable**
ABC for classes that provide the `__iter__()` method. See also the definition of *iterable*.

class collections.**Iterator**
ABC for classes that provide the *`__iter__()`* and *`next()`* methods. See also the definition of *iterator*.

class collections.**Sequence**
class collections.**MutableSequence**
ABCs for read-only and mutable *sequences*.

class collections.**Set**
class collections.**MutableSet**
ABCs for read-only and mutable sets.

class collections.**Mapping**
class collections.**MutableMapping**
ABCs for read-only and mutable *mappings*.

class collections.**MappingView**
class collections.**ItemsView**
class collections.**KeysView**
class collections.**ValuesView**
ABCs for mapping, items, keys, and values *views*.

These ABCs allow us to ask classes or instances if they provide particular functionality, for example:

```
size = None
if isinstance(myvar, collections.Sized):
    size = len(myvar)
```

Several of the ABCs are also useful as mixins that make it easier to develop classes supporting container APIs. For example, to write a class supporting the full *Set* API, it only necessary to supply the three underlying abstract methods: `__contains__()`, `__iter__()`, and `__len__()`. The ABC supplies the remaining methods such as `__and__()` and `isdisjoint()`

```
class ListBasedSet(collections.Set):
     ''' Alternate set implementation favoring space over speed
         and not requiring the set elements to be hashable. '''
     def __init__(self, iterable):
         self.elements = lst = []
         for value in iterable:
             if value not in lst:
                 lst.append(value)

     def __iter__(self):
         return iter(self.elements)

     def __contains__(self, value):
         return value in self.elements

     def __len__(self):
         return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # The __and__() method is supported automatically
```

Notes on using *Set* and *MutableSet* as a mixin:

(1) Since some set operations create new sets, the default mixin methods need a way to create new instances from an iterable. The class constructor is assumed to have a signature in the form `ClassName(iterable)`. That assumption is factored-out to an internal classmethod called `_from_iterable()` which calls `cls(iterable)` to produce a new set. If the *Set* mixin is being used in a class with a different constructor signature, you will need to override `_from_iterable()` with a classmethod that can construct new instances from an iterable argument.

(2) To override the comparisons (presumably for speed, as the semantics are fixed), redefine `__le__()` and `__ge__()`, then the other operations will automatically follow suit.

(3) The *Set* mixin provides a `_hash()` method to compute a hash value for the set; however, `__hash__()` is not defined because not all sets are hashable or immutable. To add set hashability using mixins, inherit from both *Set()* and *Hashable()*, then define `__hash__ = Set._hash`.

参见:

- OrderedSet recipe for an example built on *MutableSet*.
- For more about ABCs, see the *abc* module and **PEP 3119**.

8.4 heapq —堆队列算法

2.3 新版功能.

源码： Lib/heapq.py

这个模块提供了堆队列算法的实现，也称为优先队列算法。

堆是一个二叉树，它的每个父节点的值都只会小于或大于所有孩子节点。它使用了数组来实现：从零开始计数，对于所有的 *k* ，都有 "heap[k] <= heap[2*k+1]" 和 `heap[k] <= heap[2*k+2]` 。为了便于比较，不存在的元素被认为是无限大。堆最有趣的特性在于最小的元素总是在根结点：`heap[0]` 。

这个 API 与教材中堆算法的实现不太一样，在于两方面：(a) 我们使用了基于零开始的索引。这使得节点和其孩子节点之间的索引关系不太直观，但是由于 Python 使用了从零开始的索引，所以这样做更加合适。(b) 我们的 pop 方法返回了最小的元素，而不是最大的（这在教材中叫做 "最小堆"；而 "最大堆" 在课本中更加常见，因为它更加适用于原地排序）。

基于这两方面，把堆看作原生的 Python list 也没什么奇怪的：`heap[0]` 表示最小的元素，同时 `heap.sort()` 维护了堆的不变性！

要创建一个堆，可以使用 list 来初始化为 `[]` ，或者你可以通过一个函数*heapify()* ，来把一个 list 转换成堆。

定义了以下函数：

heapq.**heappush**(*heap*, *item*)
将 *item* 的值加入 *heap* 中，保持堆的不变性。

heapq.**heappop**(*heap*)
弹出并返回 *heap* 的最小的元素，保持堆的不变性。如果堆为空，抛出*IndexError* 。使用 `heap[0]` ，可以只访问最小的元素而不弹出它。

heapq.**heappushpop**(*heap*, *item*)
将 *item* 放入堆中，然后弹出并返回 *heap* 的最小元素。该组合操作比先调用*heappush()* 再调用*heappop()* 运行起来更有效率。

2.6 新版功能.

heapq.**heapify**(*x*)
将 list *x* 转换成堆，原地，线性时间内。

heapq.**heapreplace**(*heap*, *item*)
弹出并返回 *heap* 中最小的一项，同时推入新的 *item*。堆的大小不变。如果堆为空则引发*IndexError*。

这个单步骤操作比*heappop()* 加*heappush()* 更高效，并且在使用固定大小的堆时更为适宜。pop/push 组合总是会从堆中返回一个元素并将其替换为 *item*。

返回的值可能会比添加的 *item* 更大。如果不希望如此，可考虑改用*heappushpop()*。它的 push/pop 组合会返回两个值中较小的一个，将较大的值留在堆中。

该模块还提供了三个基于堆的通用功能函数。

heapq.**merge**(**iterables*)
将多个已排序的输入合并为一个已排序的输出（例如，合并来自多个日志文件的带时间戳的条目）。返回已排序值的*iterator*。

类似于 `sorted(itertools.chain(*iterables))` 但返回一个可迭代对象，不会一次性地将数据全部放入内存，并假定每个输入流都是已排序的（从小到大）。

2.6 新版功能.

`heapq.nlargest`(*n*, *iterable*[, *key*])

Return a list with the *n* largest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in the iterable: `key=str.lower` Equivalent to: `sorted(iterable, key=key, reverse=True)[:n]`

2.4 新版功能.

在 2.5 版更改: Added the optional *key* argument.

`heapq.nsmallest`(*n*, *iterable*[, *key*])

Return a list with the *n* smallest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in the iterable: `key=str.lower` Equivalent to: `sorted(iterable, key=key)[:n]`

2.4 新版功能.

在 2.5 版更改: Added the optional *key* argument.

后两个函数在 *n* 值较小时性能最好。对于更大的值，使用*sorted()* 函数会更有效率。此外，当 `n==1` 时，使用内置的*min()* 和*max()* 函数会更有效率。如果需要重复使用这些函数，请考虑将可迭代对象转为真正的堆。

8.4.1 基本示例

堆排序 可以通过将所有值推入堆中然后每次弹出一个最小值项来实现。

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

这类似于 `sorted(iterable)`，但与*sorted()* 不同的是这个实现是不稳定的。

堆元素可以为元组。这适用于将比较值（例如任务优先级）与跟踪的主记录进行赋值的场合:

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

8.4.2 优先队列实现说明

优先队列 是堆的常用场合，并且它的实现包含了多个挑战：

- 排序稳定性：你该如何令相同优先级的两个任务按它们最初被加入时的顺序返回？
- In the future with Python 3, tuple comparison breaks for (priority, task) pairs if the priorities are equal and the tasks do not have a default comparison order.
- 如果任务优先级发生改变，你该如何将其移至堆中的新位置？
- 或者如果一个挂起的任务需要被删除，你该如何找到它并将其移出队列？

针对前两项挑战的一种解决方案是将条目保存为包含优先级、条目计数和任务对象 3 个元素的列表。条目计数可用来打破平局，这样具有相同优先级的任务将按它们的添加顺序返回。并且由于没有哪两个条目计数是相同的，元组比较将永远不会直接比较两个任务。

其余的挑战主要包括找到挂起的任务并修改其优先级或将其完全移除。找到一个任务可使用一个指向队列中条目的字典来实现。

Removing the entry or changing its priority is more difficult because it would break the heap structure invariants. So, a possible solution is to mark the existing entry as removed and add a new entry with the revised priority:

```
pq = []                         # list of entries arranged in a heap
entry_finder = {}               # mapping of tasks to entries
REMOVED = '<removed-task>'      # placeholder for a removed task
counter = itertools.count()     # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    'Mark an existing task as REMOVED.  Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
```

8.4.3 理论

堆是通过数组来实现的，其中的元素从 0 开始计数，对于所有的 *k* 都有 `a[k] <= a[2*k+1]` 且 `a[k] <= a[2*k+2]`。为了便于比较，不存在的元素被视为无穷大。堆最有趣的特性在于 `a[0]` 总是其中最小的元素。

上面的特殊不变量是用来作为一场锦标赛的高效内存表示。下面的数字是 *k* 而不是 `a[k]`:

```
                                0

             1                                  2

      3              4                   5               6

  7       8       9       10      11      12      13      14

15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

在上面的树中，每个 *k* 单元都位于 `2*k+1` 和 `2*k+2` 之上。体育运动中我们经常见到二元锦标赛模式，每个胜者单元都位于另两个单元之上，并且我们可以沿着树形图向下追溯胜者所遇到的所有对手。但是，在许多采用这种锦标赛模式的计算机应用程序中，我们并不需要追溯胜者的历史。为了获得更高的内存利用效率，当一个胜者晋级时，我们会用较低层级的另一条目来替代它，因此规则变为一个单元和它之下的两个单元包含三个不同条目，上方单元“胜过”了两个下方单元。

如果此堆的不变量始终受到保护，则序号 0 显然是最后的赢家。删除它并找出“下一个”赢家的最简单算法方式是家某个输家（让我们假定是上图中的 30 号单元）移至 0 号位置，然后将这个新的 0 号沿树下行，不断进行值的交换，直到不变量重新建立。这显然会是树中条目总数的对数。通过迭代所有条目，你将得到一个 O(n log n) 复杂度的排序。

此排序有一个很好的特性就是你可以在排序进行期间高效地插入新条目，前提是插入的条目不比你最近取出的 0 号元素“更好”。这在模拟上下文时特别有用，在这种情况下树保存的是所有传入事件，“胜出”条件是最小调度时间。当一个事件将其他事件排入执行计划时，它们的调试时间向未来方向延长，这样它们可方便地入堆。因此，堆结构很适宜用来实现调度器，我的 MIDI 音序器就是用的这个:-)。

用于实现调度器的各种结构都得到了充分的研究，堆是非常适宜的一种，因为它们的速度相当快，并且几乎是恒定的，最坏的情况与平均情况没有太大差别。虽然还存在其他总体而言更高效的实现方式，但其最坏的情况却可能非常糟糕。

堆在大磁盘排序中也非常有用。你应该已经了解大规模排序会有多个“运行轮次”（即预排序的序列，其大小通常与 CPU 内存容量相关），随后这些轮次会进入合并通道，轮次合并的组织往往非常巧妙[1]。非常重要的一点是初始排序应产生尽可能长的运行轮次。锦标赛模式是达成此目标的好办法。如果你使用全部有用内存来进行锦标赛，替换和安排恰好适合当前运行轮次的条目，你将可以对于随机输入生成两倍于内存大小的运行轮次，对于模糊排序的输入还会有更好的效果。

另外，如果你输出磁盘上的第 0 个条目并获得一个可能不适合当前锦标赛的输入（因为其值要“胜过”上一个输出值），它无法被放入堆中，因此堆的尺寸将缩小。被释放的内存可以被巧妙地立即重用以逐步构建第二个堆，其增长速度与第一个堆的缩减速度正好相同。当第一个堆完全消失时，你可以切换新堆并启动新的运行轮次。这样做既聪明又高效!

总之，堆是值得了解的有用内存结构。我在一些应用中用到了它们，并且认为保留一个‘heap’模块是很有意义的。:-)

[1] 当前时代的磁盘平衡算法与其说是巧妙，不如说是麻烦，这是由磁盘的寻址能力导致的结果。在无法寻址的设备例如大型磁带机上，情况则相当不同，开发者必须非常聪明地（极为提前地）确保每次磁带转动都尽可能地高效（就是说能够最好地加入到合并“进程”中）。有些磁带甚至能够反向读取，这也被用来避免倒带的耗时。请相信我，真正优秀的磁带机排序看起来是极其壮观的，排序一向都是一门伟大的艺术！:-)

备注

8.5 bisect —数组二分查找算法

2.1 新版功能.

源代码： Lib/bisect.py

这个模块对有序列表提供了支持，使得他们可以在插入新数据仍然保持有序。对于长列表，如果其包含元素的比较操作十分昂贵的话，这可以是对更常见方法的改进。这个模块叫做*bisect* 因为其使用了基本的二分（bisection）算法。源代码也可以作为很棒的算法示例（边界判断也做好啦!）

定义了以下函数：

bisect.**bisect_left**(*a*, *x*, *lo=0*, *hi=len(a)*)
: 在 *a* 中找到 *x* 合适的插入点以维持有序。参数 *lo* 和 *hi* 可以被用于确定需要考虑的子集；默认情况下整个列表都会被使用。如果 *x* 已经在 *a* 里存在，那么插入点会在已存在元素之前（也就是左边）。如果 *a* 是列表（list）的话，返回值是可以被放在 `list.insert()` 的第一个参数的。

  返回的插入点 *i* 可以将数组 *a* 分成两部分。左侧是 `all(val < x for val in a[lo:i])` ，右侧是 `all(val >= x for val in a[i:hi])` 。

bisect.**bisect_right**(*a*, *x*, *lo=0*, *hi=len(a)*)
bisect.**bisect**(*a*, *x*, *lo=0*, *hi=len(a)*)
: 类似于*bisect_left()*，但是返回的插入点是 *a* 中已存在元素 *x* 的右侧。

  返回的插入点 *i* 可以将数组 *a* 分成两部分。左侧是 `all(val <= x for val in a[lo:i])`，右侧是 `all(val > x for val in a[i:hi])` for the right side。

bisect.**insort_left**(*a*, *x*, *lo=0*, *hi=len(a)*)
: 将 *x* 插入到一个有序序列 *a* 里，并维持其有序。如果 *a* 有序的话，这相当于 `a.insert(bisect.bisect_left(a, x, lo, hi), x)`。要注意搜索是 O(log n) 的，插入却是 O(n) 的。

bisect.**insort_right**(*a*, *x*, *lo=0*, *hi=len(a)*)
bisect.**insort**(*a*, *x*, *lo=0*, *hi=len(a)*)
: 类似于*insort_left()*，但是把 *x* 插入到 *a* 中已存在元素 *x* 的右侧。

参见:

SortedCollection recipe 使用 bisect 构造了一个功能完整的集合类，提供了直接的搜索方法和对用于搜索的 key 方法的支持。所有用于搜索的键都是预先计算的，以避免在搜索时对 key 方法的不必要调用。

8.5.1 搜索有序列表

上面的*bisect()* 函数对于找到插入点是有用的，但在一般的搜索任务中可能会有点尴尬。下面 5 个函数展示了如何将其转变成有序列表中的标准查找函数

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
```

(下页继续)

(续上页)

```
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

8.5.2 其他示例

函数*bisect()*还可以用于数字表查询。这个例子是使用*bisect()*从一个给定的考试成绩集合里，通过一个有序数字表，查出其对应的字母等级：90 分及以上是‘A’，80 到 89 是‘B’，以此类推

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
        i = bisect(breakpoints, score)
        return grades[i]

>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

与*sorted()*函数不同，对于*bisect()*函数来说，*key* 或者 *reversed* 参数并没有什么意义。因为这会导致设计效率低下（连续调用 bisect 函数时，是不会“记住”过去查找过的键的）。

正相反，最好去搜索预先计算好的键列表，来查找相关记录的索引。

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])
>>> keys = [r[1] for r in data]         # precomputed list of keys
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.6 array —高效的数值数组

此模块定义了一种对象类型，可以紧凑地表示基本类型值的数组：字符、整数、浮点数等。数组属于序列类型，其行为与列表非常相似，不同之处在于其中存储的对象类型是受限的。类型在对象创建时使用单个字符的 *类型码* 来指定。已定义的类型码如下：

类型码	C 类型	Python 类型	以字节表示的最小尺寸
'c'	char	character	1
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	Unicode 字符	2 (see note)
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	long	2
'l'	signed long	int	4
'L'	unsigned long	long	4
'f'	float	float	4
'd'	double	float	8

注解： The 'u' typecode corresponds to Python's unicode character. On narrow Unicode builds this is 2-bytes, on wide builds this is 4-bytes.

The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the itemsize attribute. The values stored for 'L' and 'I' items will be represented as Python long integers when retrieved, because Python's plain integer type cannot represent the full range of C's unsigned (long) integers.

这个模块定义了以下类型：

class array.**array**(*typecode*[, *initializer*])

A new array whose items are restricted by *typecode*, and initialized from the optional *initializer* value, which must be a list, string, or iterable over elements of the appropriate type.

在 2.4 版更改: Formerly, only lists or strings were accepted.

If given a list or string, the initializer is passed to the new array's *fromlist()*, *fromstring()*, or *fromunicode()* method (see below) to add initial items to the array. Otherwise, the iterable initializer is passed to the *extend()* method.

array.**ArrayType**

Obsolete alias for *array*.

Array objects support the ordinary sequence operations of indexing, slicing, concatenation, and multiplication. When using slice assignment, the assigned value must be an array object with the same type code; in all other cases, *TypeError* is raised. Array objects also implement the buffer interface, and may be used wherever buffer objects are supported.

以下数据项和方法也受到支持：

array.**typecode**

用于创建数组的类型码字符。

array.**itemsize**

在内部表示中一个数组项的字节长度。

array.**append**(*x*)
添加一个值为 *x* 的新项到数组末尾。

array.**buffer_info**()
返回一个元组 (address, length) 以给出用于存放数组内容的缓冲区元素的当前内存地址和长度。以字节表示的内存缓冲区大小可通过 array.buffer_info()[1] * array.itemsize 来计算。这在使用需要内存地址的低层级（因此不够安全）I/O 接口时会很有用，例如某些 ioctl() 操作。只要数组存在并且没有应用改变长度的操作，返回数值就是有效的。

注解: 当在 C 或 C++ 编写的代码中使用数组对象时（这是有效使用此类信息的唯一方式），使用数组对象所支持的缓冲区接口更为适宜。此方法仅保留用作向下兼容，应避免在新代码中使用。缓冲区接口的文档参见 bufferobjects。

---

array.**byteswap**()
“字节对调”所有数组项。此方法只支持大小为 1, 2, 4 或 8 字节的值；对于其他值类型将引发*RuntimeError*。它适用于从不同字节序机器所生成的文件中读取数据的情况。

array.**count**(*x*)
返回 *x* 在数组中的出现次数。

array.**extend**(*iterable*)
将来自 *iterable* 的项添加到数组末尾。如果 *iterable* 是另一个数组，它必须具有 完全相同的类型码；否则将引发*TypeError*。如果 *iterable* 不是一个数组，则它必须为可迭代对象并且其元素必须为可添加到数组的适当类型。

在 2.4 版更改: Formerly, the argument could only be another array.

array.**fromfile**(*f*, *n*)
Read *n* items (as machine values) from the file object *f* and append them to the end of the array. If less than *n* items are available, *EOFError* is raised, but the items that were available are still inserted into the array. *f* must be a real built-in file object; something else with a *read()* method won’t do.

array.**fromlist**(*list*)
添加来自 list 的项。这等价于 for x in list: a.append(x)，区别在于如果发生类型错误，数组将不会被改变。

array.**fromstring**(*s*)
添加来自字符串的项，将字符串解读为机器值的数组（相当于使用*fromfile()* 方法从文件中读取数据）。

array.**fromunicode**(*s*)
Extends this array with data from the given unicode string. The array must be a type 'u' array; otherwise a *ValueError* is raised. Use array.fromstring(unicodestring.encode(enc)) to append Unicode data to an array of some other type.

array.**index**(*x*)
返回最小的 *i* 使得 *i* 为 *x* 在数组中首次出现的序号。

array.**insert**(*i*, *x*)
将值 *x* 作为新项插入数组的 *i* 位置之前。负值将被视为相对于数组末尾的位置。

array.**pop**([*i*])
从数组中移除序号为 *i* 的项并将其返回。可选参数值默认为 -1，因此默认将移除并返回末尾项。

array.**read**(*f*, *n*)
1.5.1 版后已移除: Use the *fromfile()* method.

Read *n* items (as machine values) from the file object *f* and append them to the end of the array. If less than *n* items are available, *EOFError* is raised, but the items that were available are still inserted into the array. *f* must

be a real built-in file object; something else with a *read()* method won't do.

`array.`**remove**(*x*)
从数组中移除首次出现的 *x*。

`array.`**reverse**()
反转数组中各项的顺序。

`array.`**tofile**(*f*)
Write all items (as machine values) to the file object *f*.

`array.`**tolist**()
将数组转换为包含相同项的普通列表。

`array.`**tostring**()
Convert the array to an array of machine values and return the string representation (the same sequence of bytes that would be written to a file by the *tofile()* method.)

`array.`**tounicode**()
Convert the array to a unicode string. The array must be a type `'u'` array; otherwise a *ValueError* is raised. Use `array.tostring().decode(enc)` to obtain a unicode string from an array of some other type.

`array.`**write**(*f*)
1.5.1 版后已移除: Use the *tofile()* method.

Write all items (as machine values) to the file object *f*.

When an array object is printed or converted to a string, it is represented as `array(typecode, initializer)`. The *initializer* is omitted if the array is empty, otherwise it is a string if the *typecode* is `'c'`, otherwise it is a list of numbers. The string is guaranteed to be able to be converted back to an array with the same type and value using *eval()*, so long as the *array* class has been imported using `from array import array`. Examples:

```
array('l')
array('c', 'hello world')
array('u', u'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

参见:

模块*struct* 打包和解包异构二进制数据。

模块*xdrlib* 打包和解包用于某些远程过程调用系统的 External Data Representation (XDR) 数据。

The Numerical Python Documentation Numeric Python 扩展 (NumPy) 定义了另一种数组类型；请访问 http://www.numpy.org/ 了解有关 Numerical Python 的更多信息。

8.7 `sets` —Unordered collections of unique elements

2.3 新版功能.

2.6 版后已移除: The built-in *set*/*frozenset* types replace this module.

The *sets* module provides classes for constructing and manipulating unordered collections of unique elements. Common uses include membership testing, removing duplicates from a sequence, and computing standard math operations on sets such as intersection, union, difference, and symmetric difference.

Like other collections, sets support `x in set`, `len(set)`, and `for x in set`. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

Most set applications use the *Set* class which provides every set method except for `__hash__()`. For advanced applications requiring a hash method, the *ImmutableSet* class adds a `__hash__()` method but omits methods which alter the contents of the set. Both *Set* and *ImmutableSet* derive from `BaseSet`, an abstract class useful for determining whether something is a set: `isinstance(obj, BaseSet)`.

The set classes are implemented using dictionaries. Accordingly, the requirements for set elements are the same as those for dictionary keys; namely, that the element defines both `__eq__()` and `__hash__()`. As a result, sets cannot contain mutable elements such as lists or dictionaries. However, they can contain immutable collections such as tuples or instances of *ImmutableSet*. For convenience in implementing sets of sets, inner sets are automatically converted to immutable form, for example, `Set([Set(['dog'])])` is transformed to `Set([ImmutableSet(['dog'])])`.

class `sets.`**Set**([*iterable*])

Constructs a new empty *Set* object. If the optional *iterable* parameter is supplied, updates the set with elements obtained from iteration. All of the elements in *iterable* should be immutable or be transformable to an immutable using the protocol described in section *Protocol for automatic conversion to immutable*.

class `sets.`**ImmutableSet**([*iterable*])

Constructs a new empty *ImmutableSet* object. If the optional *iterable* parameter is supplied, updates the set with elements obtained from iteration. All of the elements in *iterable* should be immutable or be transformable to an immutable using the protocol described in section *Protocol for automatic conversion to immutable*.

Because *ImmutableSet* objects provide a `__hash__()` method, they can be used as set elements or as dictionary keys. *ImmutableSet* objects do not have methods for adding or removing elements, so all of the elements must be known when the constructor is called.

8.7.1 Set Objects

Instances of *Set* and *ImmutableSet* both provide the following operations:

Operation	Equivalent	Result
`len(s)`		number of elements in set *s* (cardinality)
`x in s`		test *x* for membership in *s*
`x not in s`		test *x* for non-membership in *s*
`s.issubset(t)`	`s <= t`	test whether every element in *s* is in *t*
`s.issuperset(t)`	`s >= t`	test whether every element in *t* is in *s*
`s.union(t)`	`s | t`	new set with elements from both *s* and *t*
`s.intersection(t)`	`s & t`	new set with elements common to *s* and *t*
`s.difference(t)`	`s - t`	new set with elements in *s* but not in *t*
`s.symmetric_difference(t)`	`s ^ t`	new set with elements in either *s* or *t* but not both
`s.copy()`		new set with a shallow copy of *s*

Note, the non-operator versions of `union()`, `intersection()`, `difference()`, and `symmetric_difference()` will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like `Set('abc') & 'cbs'` in favor of the more readable `Set('abc').intersection('cbs')`.

在 2.3.1 版更改: Formerly all arguments were required to be sets.

In addition, both *Set* and *ImmutableSet* support set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

The subset and equality comparisons do not generalize to a complete ordering function. For example, any two disjoint sets are not equal and are not subsets of each other, so *all* of the following return `False`: `a<b`, `a==b`, or `a>b`. Accordingly, sets do not implement the `__cmp__()` method.

Since sets only define partial ordering (subset relationships), the output of the `list.sort()` method is undefined for lists of sets.

The following table lists operations available in *ImmutableSet* but not found in *Set*:

Operation	Result
`hash(s)`	returns a hash value for *s*

The following table lists operations available in *Set* but not found in *ImmutableSet*:

Operation	Equivalent	Result
`s.update(t)`	*s* |= *t*	return set *s* with elements added from *t*
`s.intersection_update(t)`	*s* &= *t*	return set *s* keeping only elements also found in *t*
`s.difference_update(t)`	*s* -= *t*	return set *s* after removing elements found in *t*
`s.symmetric_difference_update(t)`	*s* ^= *t*	return set *s* with elements from *s* or *t* but not both
`s.add(x)`		add element *x* to set *s*
`s.remove(x)`		remove *x* from set *s*; raises *KeyError* if not present
`s.discard(x)`		removes *x* from set *s* if present
`s.pop()`		remove and return an arbitrary element from *s*; raises *KeyError* if empty
`s.clear()`		remove all elements from set *s*

Note, the non-operator versions of `update()`, `intersection_update()`, `difference_update()`, and `symmetric_difference_update()` will accept any iterable as an argument.

在 2.3.1 版更改: Formerly all arguments were required to be sets.

Also note, the module also includes a `union_update()` method which is an alias for `update()`. The method is included for backwards compatibility. Programmers should prefer the `update()` method because it is supported by the built-in *set()* and *frozenset()* types.

8.7.2 Example

```
>>> from sets import Set
>>> engineers = Set(['John', 'Jane', 'Jack', 'Janice'])
>>> programmers = Set(['Jack', 'Sam', 'Susan', 'Janice'])
>>> managers = Set(['Jane', 'Jack', 'Susan', 'Zack'])
>>> employees = engineers | programmers | managers           # union
>>> engineering_management = engineers & managers            # intersection
>>> fulltime_management = managers - engineers - programmers # difference
>>> engineers.add('Marvin')                                  # add element
>>> print engineers
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
>>> employees.issuperset(engineers)     # superset test
False
>>> employees.update(engineers)         # update from another set
>>> employees.issuperset(engineers)
True
>>> for group in [engineers, programmers, managers, employees]:
...     group.discard('Susan')          # unconditionally remove element
...     print group
...
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
```

(下页继续)

(续上页)

```
Set(['Janice', 'Jack', 'Sam'])
Set(['Jane', 'Zack', 'Jack'])
Set(['Jack', 'Sam', 'Jane', 'Marvin', 'Janice', 'John', 'Zack'])
```

8.7.3 Protocol for automatic conversion to immutable

Sets can only contain immutable elements. For convenience, mutable *Set* objects are automatically copied to an *ImmutableSet* before being added as a set element.

The mechanism is to always add a *hashable* element, or if it is not hashable, the element is checked to see if it has an `__as_immutable__()` method which returns an immutable equivalent.

Since *Set* objects have a `__as_immutable__()` method returning an instance of *ImmutableSet*, it is possible to construct sets of sets.

A similar mechanism is needed by the `__contains__()` and `remove()` methods which need to hash an element to check for membership in a set. Those methods check an element for hashability and, if not, check for a `__as_temporarily_immutable__()` method which returns the element wrapped by a class that provides temporary methods for `__hash__()`, `__eq__()`, and `__ne__()`.

The alternate mechanism spares the need to build a separate copy of the original mutable object.

Set objects implement the `__as_temporarily_immutable__()` method which returns the *Set* object wrapped by a new class `_TemporarilyImmutableSet`.

The two mechanisms for adding hashability are normally invisible to the user; however, a conflict can arise in a multi-threaded environment where one thread is updating a set while another has temporarily wrapped it in `_TemporarilyImmutableSet`. In other words, sets of mutable sets are not thread-safe.

8.7.4 Comparison to the built-in `set` types

The built-in *set* and *frozenset* types were designed based on lessons learned from the *sets* module. The key differences are:

- *Set* and *ImmutableSet* were renamed to *set* and *frozenset*.
- There is no equivalent to `BaseSet`. Instead, use `isinstance(x, (set, frozenset))`.
- The hash algorithm for the built-ins performs significantly better (fewer collisions) for most datasets.
- The built-in versions have more space efficient pickles.
- The built-in versions do not have a `union_update()` method. Instead, use the `update()` method which is equivalent.
- The built-in versions do not have a `_repr(sorted=True)` method. Instead, use the built-in *repr()* and *sorted()* functions: `repr(sorted(s))`.
- The built-in version does not have a protocol for automatic conversion to immutable. Many found this feature to be confusing and no one in the community reported having found real uses for it.

8.8 sched —事件调度器

源码： Lib/sched.py

---

sched 模块定义了一个实现通用事件调度程序的类：

class sched.**scheduler**(*timefunc*, *delayfunc*)

scheduler 类定义了一个调度事件的通用接口。它需要两个函数来实际处理 “外部世界” ——*timefunc* 应当不带参数地调用，并返回一个数字（“时间”，可以为任意单位）。*delayfunc* 函数应当带一个参数调用，与 *timefunc* 的输出相兼容，并且应当延迟其所指定的时间单位。每个事件运行后还将调用 *delayfunc* 并传入参数 0 以允许其他线程有机会在多线程应用中运行。

示例：

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(): print "From print_time", time.time()
...
>>> def print_some_times():
...     print time.time()
...     s.enter(5, 1, print_time, ())
...     s.enter(10, 1, print_time, ())
...     s.run()
...     print time.time()
...
>>> print_some_times()
930343690.257
From print_time 930343695.274
From print_time 930343700.273
930343700.276
```

In multi-threaded environments, the *scheduler* class has limitations with respect to thread-safety, inability to insert a new task before the one currently pending in a running scheduler, and holding up the main thread until the event queue is empty. Instead, the preferred approach is to use the *threading.Timer* class instead.

示例：

```
>>> import time
>>> from threading import Timer
>>> def print_time():
...     print "From print_time", time.time()
...
>>> def print_some_times():
...     print time.time()
...     Timer(5, print_time, ()).start()
...     Timer(10, print_time, ()).start()
...     time.sleep(11)  # sleep while time-delay events execute
...     print time.time()
...
>>> print_some_times()
930343690.257
From print_time 930343695.274
From print_time 930343700.273
930343701.301
```

8.8.1 调度器对象

scheduler 实例拥有以下方法和属性：

scheduler.**enterabs**(*time*, *priority*, *action*, *argument*)
安排一个新事件。*time* 参数应该有一个数字类型兼容的返回值，与传递给构造函数的 *timefunc* 函数的返回值兼容。计划在相同 *time* 的事件将按其 *priority* 的顺序执行。数字越小表示优先级越高。

Executing the event means executing `action(*argument)`. *argument* must be a sequence holding the parameters for *action*.

返回值是一个事件，可用于以后取消事件（参见*cancel()*）。

scheduler.**enter**(*delay*, *priority*, *action*, *argument*)
安排延后 *delay* 时间单位的事件。除了相对时间，其他参数、效果和返回值与*enterabs()* 的相同。

scheduler.**cancel**(*event*)
从队列中删除事件。如果 *event* 不是当前队列中的事件，则此方法将引发*ValueError*。

scheduler.**empty**()
Return true if the event queue is empty.

scheduler.**run**()
Run all scheduled events. This function will wait (using the `delayfunc()` function passed to the constructor) for the next event, then execute it and so on until there are no more scheduled events.

action 或 *delayfunc* 都可以引发异常。在任何一种情况下，调度程序都将保持一致状态并传播异常。如果 *action* 引发异常，则在将来调用*run()* 时不会尝试该事件。

如果一系列事件的运行时间比下一个事件之前的可用时间长，那么调度程序将完全落后。不会发生任何事件；调用代码负责取消不再相关的事件。

scheduler.**queue**
Read-only attribute returning a list of upcoming events in the order they will be run. Each event is shown as a *named tuple* with the following fields: time, priority, action, argument.

2.6 新版功能.

8.9 mutex —Mutual exclusion support

2.6 版后已移除: The *mutex* module has been removed in Python 3.

The *mutex* module defines a class that allows mutual-exclusion via acquiring and releasing locks. It does not require (or imply) *threading* or multi-tasking, though it could be useful for those purposes.

The *mutex* module defines the following class:

class mutex.**mutex**
Create a new (unlocked) mutex.

A mutex has two pieces of state —a "locked" bit and a queue. When the mutex is not locked, the queue is empty. Otherwise, the queue contains zero or more `(function, argument)` pairs representing functions (or methods) waiting to acquire the lock. When the mutex is unlocked while the queue is not empty, the first queue entry is removed and its `function(argument)` pair called, implying it now has the lock.

Of course, no multi-threading is implied –hence the funny interface for *lock()*, where a function is called once the lock is acquired.

8.9.1 Mutex Objects

mutex objects have following methods:

mutex.**test**()
Check whether the mutex is locked.

mutex.**testandset**()
"Atomic" test-and-set, grab the lock if it is not set, and return True, otherwise, return False.

mutex.**lock**(*function*, *argument*)
Execute function(argument), unless the mutex is locked. In the case it is locked, place the function and argument on the queue. See *unlock()* for explanation of when function(argument) is executed in that case.

mutex.**unlock**()
Unlock the mutex if queue is empty, otherwise execute the first element in the queue.

8.10 Queue —A synchronized queue class

注解: The *Queue* module has been renamed to queue in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/Queue.py

The *Queue* module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The *Queue* class in this module implements all the required locking semantics. It depends on the availability of thread support in Python; see the *threading* module.

The module implements three types of queue, which differ only in the order in which the entries are retrieved. In a FIFO queue, the first tasks added are the first retrieved. In a LIFO queue, the most recently added entry is the first retrieved (operating like a stack). With a priority queue, the entries are kept sorted (using the *heapq* module) and the lowest valued entry is retrieved first.

The *Queue* module defines the following classes and exceptions:

class Queue.**Queue**(*maxsize=0*)
Constructor for a FIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

class Queue.**LifoQueue**(*maxsize=0*)
Constructor for a LIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

2.6 新版功能.

class Queue.**PriorityQueue**(*maxsize=0*)
优先级队列构造函数。*maxsize* 是个整数，用于设置可以放入队列中的项目数的上限。当达到这个大小的时候，插入操作将阻塞至队列中的项目被消费掉。如果 *maxsize* 小于等于零，队列尺寸为无限大。

最小值先被取出 (最小值条目是由 sorted(list(entries))[0] 返回的条目)。条目的典型模式是一个以下形式的元组：(priority_number, data) 。

2.6 新版功能.

exception Queue.**Empty**
Exception raised when non-blocking *get()* (or *get_nowait()*) is called on a *Queue* object which is empty.

exception Queue.**Full**
Exception raised when non-blocking *put()* (or *put_nowait()*) is called on a *Queue* object which is full.

参见:

collections.deque is an alternative implementation of unbounded queues with fast atomic `append()` and `popleft()` operations that do not require locking.

8.10.1 Queue 对象

Queue objects (*Queue*, *LifoQueue*, or *PriorityQueue*) provide the public methods described below.

Queue.**qsize**()
返回队列的大致大小。注意，qsize() > 0 不保证后续的 get() 不被阻塞，qsize() < maxsize 也不保证 put() 不被阻塞。

Queue.**empty**()
如果队列为空，返回 `True` ，否则返回 `False` 。如果 empty() 返回 `True` ，不保证后续调用的 put() 不被阻塞。类似的，如果 empty() 返回 `False` ，也不保证后续调用的 get() 不被阻塞。

Queue.**full**()
如果队列是满的返回 `True` ，否则返回 `False` 。如果 full() 返回 `True` 不保证后续调用的 get() 不被阻塞。类似的，如果 full() 返回 `False` 也不保证后续调用的 put() 不被阻塞。

Queue.**put**(*item*[, *block*[, *timeout*]])
将 *item* 放入队列。如果可选参数 *block* 是 true 并且 *timeout* 是 `None` (默认)，则在必要时阻塞至有空闲插槽可用。如果 *timeout* 是个正数，将最多阻塞 *timeout* 秒，如果在这段时间没有可用的空闲插槽，将引发*Full* 异常。反之 (*block* 是 false)，如果空闲插槽立即可用，则把 *item* 放入队列，否则引发*Full* 异常 (在这种情况下，*timeout* 将被忽略)。

2.3 新版功能: The *timeout* parameter.

Queue.**put_nowait**(*item*)
相当于 `put(item, False)` 。

Queue.**get**([*block*[, *timeout*]])
从队列中移除并返回一个项目。如果可选参数 *block* 是 true 并且 *timeout* 是 `None` (默认值)，则在必要时阻塞至项目可得到。如果 *timeout* 是个正数，将最多阻塞 *timeout* 秒，如果在这段时间内项目不能得到，将引发*Empty* 异常。反之 (*block* 是 false) , 如果一个项目立即可得到，则返回一个项目，否则引发*Empty* 异常 (这种情况下，*timeout* 将被忽略)。

2.3 新版功能: The *timeout* parameter.

Queue.**get_nowait**()
相当于 `get(False)` 。

提供了两个方法，用于支持跟踪排队的任务是否被守护的消费者线程完整的处理。

Queue.**task_done**()
表示前面排队的任务已经被完成。被队列的消费者线程使用。每个*get()* 被用于获取一个任务，后续调用*task_done()* 告诉队列，该任务的处理已经完成。

如果*join()* 当前正在阻塞，在所有条目都被处理后，将解除阻塞 (意味着每个*put()* 进队列的条目的*task_done()* 都被收到)。

如果被调用的次数多于放入队列中的项目数量，将引发*ValueError* 异常。

2.5 新版功能.

Queue.**join**()
阻塞至队列中所有的元素都被接收和处理完毕。

当条目添加到队列的时候，未完成任务的计数就会增加。每当消费者线程调用*task_done()* 表示这个条目已经被回收，该条目所有工作已经完成，未完成计数就会减少。当未完成计数降到零的时候，*join()* 阻塞被解除。

2.5 新版功能.

如何等待排队的任务被完成的示例：

```
def worker():
    while True:
        item = q.get()
        do_work(item)
        q.task_done()

q = Queue()
for i in range(num_worker_threads):
     t = Thread(target=worker)
     t.daemon = True
     t.start()

for item in source():
    q.put(item)

q.join()       # block until all tasks are done
```

8.11 weakref —弱引用

2.1 新版功能.

源码： Lib/weakref.py

---

weakref 模块允许 Python 程序员创建对象的 *weak references* 。

在下文中，术语 *referent* 表示由弱引用引用的对象。

A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, *garbage collection* is free to destroy the referent and reuse its memory for something else. A primary use for weak references is to implement caches or mappings holding large objects, where it's desired that a large object not be kept alive solely because it appears in a cache or mapping.

例如，如果您有许多大型二进制图像对象，则可能希望将名称与每个对象关联起来。如果您使用 Python 字典将名称映射到图像，或将图像映射到名称，则图像对象将保持活动状态，因为它们在字典中显示为值或键。*weakref* 模块提供的*WeakKeyDictionary* 和*WeakValueDictionary* 类可以替代 Python 字典，使用弱引用来构造映射，这些映射不会仅仅因为它们出现在映射对象中而使对象保持存活。例如，如果一个图像对象是*WeakValueDictionary* 中的值，那么当对该图像对象的剩余引用是弱映射对象所持有的弱引用时，垃圾回收可以回收该对象并将其在弱映射对象中相应的条目删除。

WeakKeyDictionary and *WeakValueDictionary* use weak references in their implementation, setting up callback functions on the weak references that notify the weak dictionaries when a key or value has been reclaimed by garbage collection. Most programs should find that using one of these weak dictionary types is all they need –it's not usually necessary to create your own weak references directly. The low-level machinery used by the weak dictionary implementations is exposed by the *weakref* module for the benefit of advanced uses.

Not all objects can be weakly referenced; those objects which can include class instances, functions written in Python (but not in C), methods (both bound and unbound), sets, frozensets, file objects, *generator*s, type objects, `DBcursor` objects from the *bsddb* module, sockets, arrays, deques, regular expression pattern objects, and code objects.

在 2.4 版更改: Added support for files, sockets, arrays, and patterns.

在 2.7 版更改: 添加了对 thread.lock，threading.Lock 和代码对象的支持。

几个内建类型如 `list` 和 *dict* 不直接支持弱引用，但可以通过子类化添加支持:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)    # this object is weak referenceable
```

CPython implementation detail: Other built-in types such as *tuple* and *long* do not support weak references even when subclassed.

Extension types can easily be made to support weak references; see weakref-support.

class `weakref.`**ref**(*object*[, *callback*])
返回对 对象的弱引用。如果原始对象仍然存活，则可以通过调用引用对象来检索原始对象；如果引用的原始对象不再存在，则调用引用对象将得到*None*。如果提供了 回调而且值不是*None*，并且返回的弱引用对象仍然存活，则在对象即将终结时将调用回调; 弱引用对象将作为回调的唯一参数传递；指示物将不再可用。

许多弱引用也允许针对相同对象来构建。为每个弱引用注册的回调将按从最近注册的回调到最早注册的回调的顺序被调用。

回调所引发的异常将记录于标准错误输出，但无法被传播；它们会按与对象的 `__del__()` 方法所引发的异常相同的方式被处理。

如果 *object* 可哈希，则弱引用也为*hashable*。即使在 *object* 被删除之后它们仍将保持其哈希值。如果*hash()* 在 *object* 被删除之后才首次被调用，则该调用将引发*TypeError*。

弱引用支持相等检测，但不支持排序比较。如果被引用对象仍然存在，两个引用具有与它们的被引用对象一致的相等关系（无论 *callback* 是否相同）。如果删除了任一被引用对象，则仅在两个引用对象为同一对象时两者才相等。

在 2.4 版更改: This is now a subclassable type rather than a factory function; it derives from *object*.

`weakref.`**proxy**(*object*[, *callback*])
返回 *object* 的一个使用弱引用的代理。此函数支持在大多数上下文中使用代理，而不要求显式地对所使用的弱引用对象解除引用。返回的对象类型将为 `ProxyType` 或 `CallableProxyType`，具体取决于 *object* 是否可调用。Proxy 对象不是*hashable* 对象，无论被引用对象是否可哈希；这可避免与它们的基本可变性质相关的多种问题，并可防止它们被用作字典键。*callback* 与*ref()* 函数的同名形参含义相同。

`weakref.`**getweakrefcount**(*object*)
返回指向 *object* 的弱引用和代理的数量。

`weakref.`**getweakrefs**(*object*)
返回由指向 *object* 的所有弱引用和代理构成的列表。

class `weakref.`**WeakKeyDictionary**([*dict*])
弱引用键的映射类。当不再有对键的强引用时字典中的条目将被丢弃。这可被用来将额外数据关联到一个应用中其他部分所拥有的对象而无需在那些对象中添加属性。这对于重载了属性访问的对象来说特别有用。

注解: 注意：由于*WeakKeyDictionary* 是基于 Python 字典构建的，因而在对进行迭代时不可改变其大小。对于*WeakKeyDictionary* 来说要确保这一点可能很困难，因为程序在迭代期间执行的操作可

能导致字典中的项“神奇地”消失（这是垃圾回收机制的一个副作用）。

WeakKeyDictionary objects have the following additional methods. These expose the internal references directly. The references are not guaranteed to be “live” at the time they are used, so the result of calling the references needs to be checked before being used. This can be used to avoid creating references that will cause the garbage collector to keep the keys around longer than needed.

WeakKeyDictionary.**iterkeyrefs**()
返回包含对键的弱引用的可迭代对象。

2.5 新版功能.

WeakKeyDictionary.**keyrefs**()
Return a list of weak references to the keys.

2.5 新版功能.

class weakref.**WeakValueDictionary**([*dict*])
弱引用值的映射类。当不再有对键的强引用时字典中的条目将被丢弃。

注解: 注意：由于*WeakValueDictionary* 是基于 Python 字典构建的，因而在进行迭代时不可改变其大小。对于*WeakValueDictionary* 来说要确保这一点可能很困难，因为程序在迭代期间执行的操作可能导致字典中的项“神奇”地消失（这是垃圾回收机制的一个副作用）。

WeakValueDictionary objects have the following additional methods. These methods have the same issues as the `iterkeyrefs()` and `keyrefs()` methods of *WeakKeyDictionary* objects.

WeakValueDictionary.**itervaluerefs**()
返回包含对值的弱引用的可迭代对象。

2.5 新版功能.

WeakValueDictionary.**valuerefs**()
Return a list of weak references to the values.

2.5 新版功能.

class weakref.**WeakSet**([*elements*])
保持对其元素弱引用的集合类。当不再有对某个元素的强引用时元素将被丢弃。

2.7 新版功能.

weakref.**ReferenceType**
弱引用对象的类型对象。

weakref.**ProxyType**
不可调用的对象的代理的类型对象。

weakref.**CallableProxyType**
可调用对象的代理的类型对象。

weakref.**ProxyTypes**
包含所有代理的类型对象的序列。这可以用于更方便地检测一个对象是否是代理，而不必依赖于两种代理对象的名称。

exception weakref.**ReferenceError**
Exception raised when a proxy object is used but the underlying object has been collected. This is the same as the standard *ReferenceError* exception.

参见:

PEP 205 - 弱引用 此特性的提议和理由，包括早期实现的链接和其他语言中类似特性的相关信息。

8.11.1 弱引用对象

Weak reference objects have no attributes or methods, but do allow the referent to be obtained, if it still exists, by calling it:

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

如果引用已不存在，则调用引用对象将返回*None*:

```
>>> del o, o2
>>> print r()
None
```

检测一个弱引用对象是否仍然存在应该使用表达式 `ref() is not None`。通常，需要使用引用对象的应用代码应当遵循这样的模式:

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print "Object has been deallocated; can't frobnicate."
else:
    print "Object is still live!"
    o.do_something_useful()
```

使用单独的“存活”测试会在多线程应用中制造竞争条件；其他线程可能导致某个弱引用在该弱引用被调用前就失效；上述的写法在多线程应用和单线程应用中都是安全的。

特别版本的*ref* 对象可以通过子类化来创建。在*WeakValueDictionary* 的实现中就使用了这种方式来减少映射中每个条目的内存开销。这对于将附加信息关联到引用的情况最为适用，但也可以被用于在调用中插入额外处理来提取引用。

这个例子演示了如何将*ref* 的一个子类用于存储有关对象的附加信息并在引用被访问时影响其所返回的值:

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, **annotations):
        super(ExtendedRef, self).__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.iteritems():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of
        times the reference has been called.
        """
```

(下页继续)

(续上页)

```
        ob = super(ExtendedRef, self).__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

8.11.2 示例

这个简单的例子演示了一个应用如何使用对象 ID 来提取之前出现过的对象。然后对象的 ID 可以在其它数据结构中使用，而无须强制对象保持存活，但处于存活状态的对象也仍然可以通过 ID 来提取。

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

8.12 `UserDict` —Class wrapper for dictionary objects

Source code: Lib/UserDict.py

The module defines a mixin, *DictMixin*, defining all dictionary methods for classes that already have a minimum mapping interface. This greatly simplifies writing classes that need to be substitutable for dictionaries (such as the shelve module).

This module also defines a class, *UserDict*, that acts as a wrapper around dictionary objects. The need for this class has been largely supplanted by the ability to subclass directly from *dict* (a feature that became available starting with Python version 2.2). Prior to the introduction of *dict*, the *UserDict* class was used to create dictionary-like sub-classes that obtained new behaviors by overriding existing methods or adding new ones.

The *UserDict* module defines the *UserDict* class and *DictMixin*:

class UserDict.**UserDict**([*initialdata*])

Class that simulates a dictionary. The instance’s contents are kept in a regular dictionary, which is accessible via the `data` attribute of *UserDict* instances. If *initialdata* is provided, `data` is initialized with its contents; note that a reference to *initialdata* will not be kept, allowing it be used for other purposes.

注解： For backward compatibility, instances of *UserDict* are not iterable.

---

class UserDict.**IterableUserDict**([*initialdata*])

Subclass of *UserDict* that supports direct iteration (e.g. `for key in myDict`).

In addition to supporting the methods and operations of mappings (see section 映射类型—*dict*), *UserDict* and *IterableUserDict* instances provide the following attribute:

IterableUserDict.**data**
A real dictionary used to store the contents of the *UserDict* class.

class UserDict.**DictMixin**
Mixin defining all dictionary methods for classes that already have a minimum dictionary interface including `__getitem__()`, `__setitem__()`, `__delitem__()`, and `keys()`.

This mixin should be used as a superclass. Adding each of the above methods adds progressively more functionality. For instance, defining all but `__delitem__()` will preclude only `pop()` and `popitem()` from the full interface.

In addition to the four base methods, progressively more efficiency comes with defining `__contains__()`, `__iter__()`, and `iteritems()`.

Since the mixin has no knowledge of the subclass constructor, it does not define `__init__()` or *copy()*.

Starting with Python version 2.6, it is recommended to use *collections.MutableMapping* instead of *DictMixin*.

Note that DictMixin does not implement the *viewkeys()*, *viewvalues()*, or *viewitems()* methods.

8.13 `UserList` —Class wrapper for list objects

注解: When Python 2.2 was released, many of the use cases for this class were subsumed by the ability to subclass `list` directly. However, a handful of use cases remain.

This module provides a list-interface around an underlying data store. By default, that data store is a `list`; however, it can be used to wrap a list-like interface around other objects (such as persistent storage).

In addition, this class can be mixed-in with built-in classes using multiple inheritance. This can sometimes be useful. For example, you can inherit from *UserList* and *str* at the same time. That would not be possible with both a real `list` and a real *str*.

This module defines a class that acts as a wrapper around list objects. It is a useful base class for your own list-like classes, which can inherit from them and override existing methods or add new ones. In this way one can add new behaviors to lists.

The *UserList* module defines the *UserList* class:

class UserList.**UserList**([*list*])
Class that simulates a list. The instance's contents are kept in a regular list, which is accessible via the *data* attribute of *UserList* instances. The instance's contents are initially set to a copy of *list*, defaulting to the empty list `[]`. *list* can be any iterable, e.g. a real Python list or a *UserList* object.

注解: The *UserList* class has been moved to the *collections* module in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

In addition to supporting the methods and operations of mutable sequences (see section *Sequence Types —str, unicode, list, tuple, bytearray, buffer, xrange*), *UserList* instances provide the following attribute:

UserList.**data**
A real Python list object used to store the contents of the *UserList* class.

Subclassing requirements: Subclasses of *UserList* are expected to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the actual

implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.

If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided in that case.

在 2.0 版更改: Python versions 1.5.2 and 1.6 also required that the constructor be callable with no parameters, and offer a mutable `data` attribute. Earlier versions of Python did not attempt to create instances of the derived class.

8.14 `UserString` —Class wrapper for string objects

注解: This *UserString* class from this module is available for backward compatibility only. If you are writing code that does not need to work with versions of Python earlier than Python 2.2, please consider subclassing directly from the built-in *str* type instead of using *UserString* (there is no built-in equivalent to *MutableString*).

This module defines a class that acts as a wrapper around string objects. It is a useful base class for your own string-like classes, which can inherit from them and override existing methods or add new ones. In this way one can add new behaviors to strings.

It should be noted that these classes are highly inefficient compared to real string or Unicode objects; this is especially the case for *MutableString*.

The *UserString* module defines the following classes:

class UserString.**UserString**([*sequence*])
Class that simulates a string or a Unicode string object. The instance's content is kept in a regular string or Unicode string object, which is accessible via the `data` attribute of *UserString* instances. The instance's contents are initially set to a copy of *sequence*. *sequence* can be either a regular Python string or Unicode string, an instance of *UserString* (or a subclass) or an arbitrary sequence which can be converted into a string using the built-in *str()* function.

注解: The *UserString* class has been moved to the *collections* module in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

class UserString.**MutableString**([*sequence*])
This class is derived from the *UserString* above and redefines strings to be *mutable*. Mutable strings can't be used as dictionary keys, because dictionaries require *immutable* objects as keys. The main intention of this class is to serve as an educational example for inheritance and necessity to remove (override) the `__hash__()` method in order to trap attempts to use a mutable object as dictionary key, which would be otherwise very error prone and hard to track down.

2.6 版后已移除: The *MutableString* class has been removed in Python 3.

In addition to supporting the methods and operations of string and Unicode objects (see section 字符串的方法), *UserString* instances provide the following attribute:

MutableString.**data**
A real Python string or Unicode object used to store the content of the *UserString* class.

8.15 types —Names for built-in types

源代码: Lib/types.py

This module defines names for some object types that are used by the standard Python interpreter, but not for the types defined by various extension modules. Also, it does not include some of the types that arise during processing such as the `listiterator` type. It is safe to use `from types import *` —the module does not export any names besides the ones listed here. New names exported by future versions of this module will all end in `Type`.

Typical use is for functions that do different things depending on their argument types, like the following:

```
from types import *
def delete(mylist, item):
    if type(item) is IntType:
        del mylist[item]
    else:
        mylist.remove(item)
```

Starting in Python 2.2, built-in factory functions such as *int()* and *str()* are also names for the corresponding types. This is now the preferred way to access the type instead of using the *types* module. Accordingly, the example above should be written as follows:

```
def delete(mylist, item):
    if isinstance(item, int):
        del mylist[item]
    else:
        mylist.remove(item)
```

The module defines the following names:

types.**NoneType**
The type of `None`.

types.**TypeType**
The type of type objects (such as returned by *type()*); alias of the built-in *type*.

types.**BooleanType**
The type of the *bool* values `True` and `False`; alias of the built-in *bool*.

2.3 新版功能.

types.**IntType**
The type of integers (e.g. `1`); alias of the built-in *int*.

types.**LongType**
The type of long integers (e.g. `1L`); alias of the built-in *long*.

types.**FloatType**
The type of floating point numbers (e.g. `1.0`); alias of the built-in *float*.

types.**ComplexType**
The type of complex numbers (e.g. `1.0j`). This is not defined if Python was built without complex number support.

types.**StringType**
The type of character strings (e.g. `'Spam'`); alias of the built-in *str*.

types.**UnicodeType**
The type of Unicode character strings (e.g. `u'Spam'`). This is not defined if Python was built without Unicode support. It's an alias of the built-in *unicode*.

types.**TupleType**
The type of tuples (e.g. `(1, 2, 3, 'Spam')`); alias of the built-in *tuple*.

types.**ListType**
The type of lists (e.g. `[0, 1, 2, 3]`); alias of the built-in `list`.

types.**DictType**
The type of dictionaries (e.g. `{'Bacon': 1, 'Ham': 0}`); alias of the built-in *dict*.

types.**DictionaryType**
An alternate name for `DictType`.

types.**FunctionType**
types.**LambdaType**
The type of user-defined functions and functions created by `lambda` expressions.

types.**GeneratorType**
The type of *generator*-iterator objects, produced by calling a generator function.

2.2 新版功能.

types.**CodeType**
代码对象的类型，例如*compile()* 的返回值。

types.**ClassType**
The type of user-defined old-style classes.

types.**InstanceType**
The type of instances of user-defined old-style classes.

types.**MethodType**
用户自定义类实例方法的类型。

types.**UnboundMethodType**
An alternate name for `MethodType`.

types.**BuiltinFunctionType**
types.**BuiltinMethodType**
内置函数例如*len()* 或*sys.exit()* 以及内置类方法的类型。(这里所说的“内置”是指“以 C 语言编写”。)

types.**ModuleType**
The type of modules.

types.**FileType**
The type of open file objects such as `sys.stdout`; alias of the built-in *file*.

types.**XRangeType**
The type of range objects returned by *xrange()*; alias of the built-in *xrange*.

types.**SliceType**
The type of objects returned by *slice()*; alias of the built-in *slice*.

types.**EllipsisType**
The type of `Ellipsis`.

types.**TracebackType**
The type of traceback objects such as found in `sys.exc_traceback`.

types.**FrameType**
帧对象的类型，例如 tb.tb_frame 中的对象，其中 tb 是一个回溯对象。

types.**BufferType**
The type of buffer objects created by the *buffer()* function.

types.**DictProxyType**
The type of dict proxies, such as TypeType.__dict__.

types.**NotImplementedType**
The type of NotImplemented

types.**GetSetDescriptorType**
使用 PyGetSetDef 在扩展模块中定义的对象的类型，例如 FrameType.f_locals 或 array.array.typecode。此类型被用作对象属性的描述器；它的目的与*property* 类型相同，但专门针对在扩展模块中定义的类。

2.5 新版功能.

types.**MemberDescriptorType**
使用 PyMemberDef 在扩展模块中定义的对象的类型，例如 datetime.timedelta.days。此类型被用作使用标准转换函数的简单 C 数据成员的描述器；它的目的与*property* 类型相同，但专门针对在扩展模块中定义的类。

在 Python 的其它实现中，此类型可能与 GetSetDescriptorType 完全相同。

2.5 新版功能.

types.**StringTypes**
A sequence containing StringType and UnicodeType used to facilitate easier checking for any string object. Using this is more portable than using a sequence of the two string types constructed elsewhere since it only contains UnicodeType if it has been built in the running version of Python. For example: isinstance(s, types.StringTypes).

2.2 新版功能.

8.16 new —Creation of runtime internal objects

2.6 版后已移除: The *new* module has been removed in Python 3. Use the *types* module's classes instead.

The *new* module allows an interface to the interpreter object creation functions. This is for use primarily in marshal-type functions, when a new object needs to be created "magically" and not by using the regular creation functions. This module provides a low-level interface to the interpreter, so care must be exercised when using this module. It is possible to supply non-sensical arguments which crash the interpreter when the object is used.

The *new* module defines the following functions:

new.**instance**(*class*[, *dict*])
This function creates an instance of *class* with dictionary *dict* without calling the __init__() constructor. If *dict* is omitted or None, a new, empty dictionary is created for the new instance. Note that there are no guarantees that the object will be in a consistent state.

new.**instancemethod**(*function*, *instance*, *class*)
This function will return a method object, bound to *instance*, or unbound if *instance* is None. *function* must be callable.

new.**function**(*code*, *globals*[, *name*[, *argdefs*[, *closure*]]])
Returns a (Python) function with the given code and globals. If *name* is given, it must be a string or None. If it is a string, the function will have the given name, otherwise the function name will be taken from code.co_name.

If *argdefs* is given, it must be a tuple and will be used to determine the default values of parameters. If *closure* is given, it must be `None` or a tuple of cell objects containing objects to bind to the names in `code.co_freevars`.

`new.`**`code`**(*argcount*, *nlocals*, *stacksize*, *flags*, *codestring*, *constants*, *names*, *varnames*, *filename*, *name*, *firstlineno*, *lnotab*)
This function is an interface to the `PyCode_New()` C function.

`new.`**`module`**(*name*[, *doc*])
This function returns a new module object with name *name*. *name* must be a string. The optional *doc* argument can have any type.

`new.`**`classobj`**(*name*, *baseclasses*, *dict*)
This function returns a new class object, with name *name*, derived from *baseclasses* (which should be a tuple of classes) and with namespace *dict*.

8.17 copy —浅层 (shallow) 和深层 (deep) 复制操作

Python 中赋值语句不复制对象，而是在目标和对象之间创建绑定 (bindings) 关系。对于自身可变或者包含可变项的集合对象，开发者有时会需要生成其副本用于改变操作，进而避免改变原对象。本模块提供了通用的浅层复制和深层复制操作（如下所述）。

接口摘要：

`copy.`**`copy`**(*x*)
返回 *x* 的浅层复制。

`copy.`**`deepcopy`**(*x*)
返回 *x* 的深层复制。

exception `copy.`**`error`**
针对模块特定错误引发。

浅层复制和深层复制之间的区别仅与复合对象 (即包含其他对象的对象，如列表或类的实例) 相关：

- 一个 *浅层复制*会构造一个新的复合对象，然后（在可能的范围内）将原对象中找到的 *引用*插入其中。
- 一个 *深层复制*会构造一个新的复合对象，然后递归地将原始对象中所找到的对象的 *副本*插入。

深度复制操作通常存在两个问题，而浅层复制操作并不存在这些问题：

- 递归对象 (直接或间接包含对自身引用的复合对象) 可能会导致递归循环。
- 由于深层复制会复制所有内容，因此可能会过多复制（例如本应该在副本之间共享的数据）。

The *deepcopy()* function avoids these problems by:

- keeping a "memo" dictionary of objects already copied during the current copying pass; and
- 允许用户定义的类重载复制操作或复制的组件集合。

该模块不复制模块、方法、栈追踪（stack trace）、栈帧（stack frame）、文件、套接字、窗口、数组以及任何类似的类型。它通过不改变地返回原始对象来（浅层或深层地）"复制"函数和类；这与*pickle*模块处理这类问题的方式是相似的。

制作字典的浅层复制可以使用*dict.copy()*方法，而制作列表的浅层复制可以通过赋值整个列表的切片完成，例如，`copied_list = original_list[:]`。

在 2.5 版更改: Added copying functions.

Classes can use the same interfaces to control copying that they use to control pickling. See the description of module *pickle* for information on these methods. The *copy* module does not use the *copy_reg* registration module.

In order for a class to define its own copy implementation, it can define special methods `__copy__()` and `__deepcopy__()`. The former is called to implement the shallow copy operation; no additional arguments are passed. The latter is called to implement the deep copy operation; it is passed one argument, the memo dictionary. If the `__deepcopy__()` implementation needs to make a deep copy of a component, it should call the *deepcopy()* function with the component as first argument and the memo dictionary as second argument.

参见:

模块*pickle* 讨论了支持对象状态检索和恢复的特殊方法。

8.18 pprint —数据美化输出

源代码: Lib/pprint.py

The *pprint* module provides a capability to "pretty-print" arbitrary Python data structures in a form which can be used as input to the interpreter. If the formatted structures include objects which are not fundamental Python types, the representation may not be loadable. This may be the case if objects such as files, sockets, classes, or instances are included, as well as many other built-in objects which are not representable as Python constants.

格式化后的形式会在可能的情况下以单行来表示对象，并在无法在允许宽度内容纳对象的情况下将其分为多行。如果你需要调整宽度限制则应显式地构造*PrettyPrinter* 对象。

在 2.5 版更改: Dictionaries are sorted by key before the display is computed; before 2.5, a dictionary was sorted only if its display required more than one line, although that wasn't documented.

在 2.6 版更改: Added support for *set* and *frozenset*.

pprint 模块定义了一个类:

class pprint.**PrettyPrinter**(*indent=1*, *width=80*, *depth=None*, *stream=None*)

Construct a *PrettyPrinter* instance. This constructor understands several keyword parameters. An output stream may be set using the *stream* keyword; the only method used on the stream object is the file protocol's `write()` method. If not specified, the *PrettyPrinter* adopts `sys.stdout`. Three additional parameters may be used to control the formatted representation. The keywords are *indent*, *depth*, and *width*. The amount of indentation added for each recursive level is specified by *indent*; the default is one. Other values can cause output to look a little odd, but can make nesting easier to spot. The number of levels which may be printed is controlled by *depth*; if the data structure being printed is too deep, the next contained level is replaced by `...`. By default, there is no constraint on the depth of the objects being formatted. The desired output width is constrained using the *width* parameter; the default is 80 characters. If a structure cannot be formatted within the constrained width, a best effort will be made.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[   ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
    'knights',
    'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',)))))))))
>>> pp = pprint.PrettyPrinter(depth=6)
```

(下页继续)

(续上页)

```
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...)))))))
```

The *PrettyPrinter* class supports several derivative functions:

pprint.**pformat**(*object*, *indent=1*, *width=80*, *depth=None*)
Return the formatted representation of *object* as a string. *indent*, *width* and *depth* will be passed to the *PrettyPrinter* constructor as formatting parameters.

在 2.4 版更改: The parameters *indent*, *width* and *depth* were added.

pprint.**pprint**(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*)
Prints the formatted representation of *object* on *stream*, followed by a newline. If *stream* is `None`, `sys.stdout` is used. This may be used in the interactive interpreter instead of a `print` statement for inspecting values. *indent*, *width* and *depth* will be passed to the *PrettyPrinter* constructor as formatting parameters.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

在 2.4 版更改: The parameters *indent*, *width* and *depth* were added.

pprint.**isreadable**(*object*)
Determine if the formatted representation of *object* is "readable," or can be used to reconstruct the value using *eval()*. This always returns `False` for recursive objects.

```
>>> pprint.isreadable(stuff)
False
```

pprint.**isrecursive**(*object*)
确定 *object* 是否需要递归表示。

此外还定义了一个支持函数:

pprint.**saferepr**(*object*)
返回 *object* 的字符串表示，并为递归数据结构提供保护。如果 *object* 的表示形式公开了一个递归条目，该递归引用会被表示为 `<Recursion on typename with id=number>`。该表示因而不会进行其它格式化。

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni']"
```

8.18.1 PrettyPrinter 对象

PrettyPrinter 的实例具有下列方法：

PrettyPrinter.**pformat**(*object*)
返回 *object* 格式化表示。这会将传给*PrettyPrinter* 构造器的选项纳入考虑。

PrettyPrinter.**pprint**(*object*)
在所配置的流上打印 *object* 的格式化表示，并附加一个换行符。

下列方法提供了与同名函数相对应的实现。在实例上使用这些方法效率会更高一些，因为不需要创建新的*PrettyPrinter* 对象。

PrettyPrinter.**isreadable**(*object*)
确定对象的格式化表示是否“可读”，或者是否可使用*eval()* 重建对象值。请注意此方法对于递归对象将返回 False。如果设置了*PrettyPrinter* 的 *depth* 形参并且对象深度超出允许范围，此方法将返回 False。

PrettyPrinter.**isrecursive**(*object*)
确定对象是否需要递归表示。

此方法作为一个钩子提供，允许子类修改将对象转换为字符串的方式。默认实现使用*saferepr()* 实现的内部方式。

PrettyPrinter.**format**(*object*, *context*, *maxlevels*, *level*)
返回三个值：字符串形式的 *object* 已格式化版本，指明结果是否可读的旗标，以及指明是否检测到递归的旗标。第一个参数是要表示的对象。第二个是以对象*id()* 为键的字典，这些对象是当前表示上下文的一部分（影响 *object* 表示的直接和间接容器）；如果需要呈现一个已经在 *context* 中表示的对象，则第三个返回值应当为 True。对*format()* 方法的递归调用应当将容器的附加条目添加到此字典中。第三个参数 *maxlevels* 给出了对递归的请求限制；如果没有请求限制则其值将为 0。此参数应当不加修改地传给递归调用。第四个参数 *level* 给出于当前层级；传给递归调用的参数值应当小于当前调用的值。

2.3 新版功能.

8.18.2 pprint Example

This example demonstrates several uses of the *pprint()* function and its parameters.

```
>>> import pprint
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',)))))))))
>>> stuff = ['a' * 10, tup, ['a' * 30, 'b' * 30], ['c' * 20, 'd' * 20]]
>>> pprint.pprint(stuff)
['aaaaaaaaaa',
 ('spam',
  ('eggs',
   ('lumberjack',
    ('knights', ('ni', ('dead', ('parrot', ('fresh fruit',)))))))),
 ['aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa', 'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb'],
 ['cccccccccccccccccccc', 'ddddddddddddddddddd']]
>>> pprint.pprint(stuff, depth=3)
['aaaaaaaaaa',
 ('spam', ('eggs', (...))),
 ['aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa', 'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb'],
 ['cccccccccccccccccccc', 'ddddddddddddddddddd']]
>>> pprint.pprint(stuff, width=60)
['aaaaaaaaaa',
 ('spam',
```

(下页继续)

(续上页)

```
  ('eggs',
   ('lumberjack',
    ('knights',
     ('ni', ('dead', ('parrot', ('fresh fruit',)))))))),
 ['aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa',
  'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb'],
 ['ccccccccccccccccccc', 'ddddddddddddddddddddd']]
```

8.19 repr —Alternate repr() implementation

注解: The *repr* module has been renamed to `reprlib` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/repr.py

The *repr* module provides a means for producing object representations with limits on the size of the resulting strings. This is used in the Python debugger and may be useful in other contexts as well.

This module provides a class, an instance, and a function:

class repr.**Repr**
: Class which provides formatting services useful in implementing functions similar to the built-in *repr()*; size limits for different object types are added to avoid the generation of representations which are excessively long.

repr.**aRepr**
: This is an instance of *Repr* which is used to provide the *repr()* function described below. Changing the attributes of this object will affect the size limits used by *repr()* and the Python debugger.

repr.**repr**(*obj*)
: This is the *repr()* method of `aRepr`. It returns a string similar to that returned by the built-in function of the same name, but with limits on most sizes.

8.19.1 Repr Objects

Repr instances provide several attributes which can be used to provide size limits for the representations of different object types, and methods which format specific object types.

Repr.**maxlevel**
: Depth limit on the creation of recursive representations. The default is `6`.

Repr.**maxdict**
Repr.**maxlist**
Repr.**maxtuple**
Repr.**maxset**
Repr.**maxfrozenset**
Repr.**maxdeque**
Repr.**maxarray**
: Limits on the number of entries represented for the named object type. The default is `4` for *maxdict*, `5` for *maxarray*, and `6` for the others.

  2.4 新版功能: *maxset*, *maxfrozenset*, and *set*.

Repr.**maxlong**
Maximum number of characters in the representation for a long integer. Digits are dropped from the middle. The default is 40.

Repr.**maxstring**
Limit on the number of characters in the representation of the string. Note that the "normal" representation of the string is used as the character source: if escape sequences are needed in the representation, these may be mangled when the representation is shortened. The default is 30.

Repr.**maxother**
This limit is used to control the size of object types for which no specific formatting method is available on the *Repr* object. It is applied in a similar manner as *maxstring*. The default is 20.

Repr.**repr**(*obj*)
The equivalent to the built-in *repr()* that uses the formatting imposed by the instance.

Repr.**repr1**(*obj*, *level*)
Recursive implementation used by *repr()*. This uses the type of *obj* to determine which formatting method to call, passing it *obj* and *level*. The type-specific methods should call *repr1()* to perform recursive formatting, with `level - 1` for the value of *level* in the recursive call.

Repr.**repr_TYPE**(*obj*, *level*)
Formatting methods for specific types are implemented as methods with a name based on the type name. In the method name, **TYPE** is replaced by `string.join(string.split(type(obj).__name__, '_'))`. Dispatch to these methods is handled by *repr1()*. Type-specific methods which need to recursively format a value should call `self.repr1(subobj, level - 1)`.

8.19.2 Subclassing Repr Objects

The use of dynamic dispatching by *Repr.repr1()* allows subclasses of *Repr* to add support for additional built-in object types or to modify the handling of types already supported. This example shows how special support for file objects could be added:

```
import repr as reprlib
import sys

class MyRepr(reprlib.Repr):
    def repr_file(self, obj, level):
        if obj.name in ['<stdin>', '<stdout>', '<stderr>']:
            return obj.name
        else:
            return repr(obj)

aRepr = MyRepr()
print aRepr.repr(sys.stdin)          # prints '<stdin>'
```

CHAPTER 9

数字和数学模块

The modules described in this chapter provide numeric and math-related functions and data types. The *numbers* module defines an abstract hierarchy of numeric types. The *math* and *cmath* modules contain various mathematical functions for floating-point and complex numbers. For users more interested in decimal accuracy than in speed, the *decimal* module supports exact representations of decimal numbers.

本章包含以下模块的文档:

9.1 numbers —数字的抽象基类

2.6 新版功能.

numbers 模块 (**PEP 3141**) 定义了数字抽象基类 的层次结构，其中逐级定义了更多操作。此模块中所定义的类型都不可被实例化。

class numbers.**Number**
数字的层次结构的基础。如果你只想确认参数 *x* 是不是数字而不关心其类型，则使用 “isinstance(x, Number)“。

9.1.1 数字的层次

class numbers.**Complex**
内置在类型*complex* 里的子类描述了复数和它的运算操作。这些操作有：转化至*complex* 和*bool*，*real*、*imag*、+、-、*、/、*abs()*、*conjugate()*、== 和 !=。所有的异常，- 和 != ，都是抽象的。

real
抽象的。得到该数字的实数部分。

imag
抽象的。得到该数字的虚数部分。

conjugate()
抽象的。返回共轭复数。例如 `(1+3j).conjugate() == (1-3j)`。

class numbers.**Real**
相对于*Complex*，*Real* 加入了只有实数才能进行的操作。

简单的说，它们是：转化至*float*，*math.trunc()*、*round()*、*math.floor()*、*math.ceil()*、*divmod()*、//、%、<、<=、>、和 >=。

实数同样默认支持*complex()*、*real*、*imag* 和*conjugate()*。

class numbers.**Rational**
子类型*Real* 并加入*numerator* 和*denominator* 两种属性，这两种属性应该属于最低的级别。加入后，这默认支持*float()*。

numerator
摘要。

denominator
摘要。

class numbers.**Integral**
子类型*Rational* 加上转化至*int*。默认支持*float()*、*numerator* 和*denominator*。在 ** 中加入抽象方法和比特字符串的操作：<<、>>、&、^、|、~。

9.1.2 类型接口注释。

实现者需要注意使相等的数字相等并拥有同样的值。当这两个数使用不同的扩展模块时，这其中的差异是很微妙的。例如，用*fractions.Fraction* 实现*hash()* 如下:

```
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

加入更多数字的 ABC

当然，这里有更多支持数字的 ABC，如果不加入这些，就将缺少层次感。你可以用如下方法在*Complex* 和*Real* 中加入 “MyFoo“:

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

实现算数运算

我们希望实现计算，因此，混合模式操作要么调用一个作者知道参数类型的实现，要么转变成为最接近的内置类型并对这个执行操作。对于子类*Integral*，这意味着 __add__() 和 __radd__() 必须用如下方式定义：

```
class MyIntegral(Integral):

    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

There are 5 different cases for a mixed-type operation on subclasses of *Complex*. I'll refer to all of the above code that doesn't refer to `MyIntegral` and `OtherTypeIKnowAbout` as "boilerplate". `a` will be an instance of `A`, which is a subtype of *Complex* (`a : A <: Complex`), and `b : B <: Complex`. I'll consider `a + b`:

1. 如果 `A` 被定义成一个承认 "b" 的 __add__()，一切都没有问题。
2. 如果 `A` 转回成"模板"失败，它将返回一个属于 __add__() 的值，我们需要避免 `B` 定义了一个更加智能的 __radd__()，因此模板需要返回一个属于 __add__() 的*NotImplemented*。(或者 `A` 可能完全不实现 __add__()。)
3. 接着看 `B` 的 __radd__()。如果它承认 `a`，一切都没有问题。
4. 如果没有成功回退到模板，就没有更多的方法可以去尝试，因此这里将使用默认的实现。
5. 如果 `B <: A`，Python 在 `A.__add__` 之前尝试 `B.__radd__`。这是可行的，是通过对 `A` 的认识实现的，因此这可以在交给*Complex*处理之前处理这些实例。

如果 `A <: Complex` 和 `B <: Real` 没有共享任何资源，那么适当的共享操作涉及内置的*complex*，并且分别获得 __radd__()，因此 `a+b == b+a`。

由于对任何一直类型的大部分操作是十分相似的，可以定义一个帮助函数，即一个生成后续或相反的实例的生成器。例如，使用*fractions.Fraction*如下：

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, long, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
```

(下页继续)

(续上页)

```
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
            return monomorphic_operator(a, b)
        elif isinstance(a, numbers.Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, numbers.Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```

9.2 math —数学函数

This module is always available. It provides access to the mathematical functions defined by the C standard.

这些函数不适用于复数；如果你需要计算复数，请使用*cmath*模块中的同名函数。将支持计算复数的函数区分开的目的，来自于大多数开发者并不愿意像数学家一样需要学习复数的概念。得到一个异常而不是一个复数结果使得开发者能够更早地监测到传递给这些函数的参数中包含复数，进而调查其产生的原因。

该模块提供了以下函数。除非另有明确说明，否则所有返回值均为浮点数。

9.2.1 数论与表示函数

math.**ceil**(*x*)
Return the ceiling of *x* as a float, the smallest integer value greater than or equal to *x*.

math.**copysign**(*x*, *y*)
Return *x* with the sign of *y*. On a platform that supports signed zeros, `copysign(1.0, -0.0)` returns *-1.0*.

2.6 新版功能.

math.**fabs**(*x*)
返回 *x* 的绝对值。

math.**factorial**(*x*)
Return *x* factorial. Raises *ValueError* if *x* is not integral or is negative.

2.6 新版功能.

math.**floor**(*x*)
Return the floor of *x* as a float, the largest integer value less than or equal to *x*.

math.**fmod**(*x*, *y*)
返回 `fmod(x, y)`，由平台 C 库定义。请注意，Python 表达式 `x % y` 可能不会返回相同的结果。C 标准的目的是 `fmod(x, y)` 完全（数学上；到无限精度）等于 `x - n*y` 对于某个整数 *n*，使得结果具有与 *x* 相同的符号和小于 `abs(y)` 的幅度。Python 的 `x % y` 返回带有 *y* 符号的结果，并且可能不能完全计算浮点参数。例如，`fmod(-1e-100, 1e100)` 是 `-1e-100`，但 Python 的 `-1e-100 % 1e100` 的结果是 `1e100-1e-100`，它不能完全表示为浮点数，并且取整为令人惊讶的 `1e100`。出于这个原因，函数*fmod()* 在使用浮点数时通常是首选，而 Python 的 `x % y` 在使用整数时是首选。

math.**frexp**(*x*)
返回 *x* 的尾数和指数作为对 "(m, e)"。*m* 是一个浮点数，*e* 是一个整数，正好是 `x == m * 2**e`。如果 *x* 为零，则返回 `(0.0, 0)`，否则返回 `0.5 <= abs(m) < 1`。这用于以可移植方式 "分离" 浮点数的内部表示。

math.**fsum**(*iterable*)
返回迭代中的精确浮点值。通过跟踪多个中间部分和来避免精度损失:

```
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

该算法的准确性取决于 IEEE-754 算术保证和舍入模式为半偶的典型情况。在某些非 Windows 版本中，底层 C 库使用扩展精度添加，并且有时可能会使中间和加倍，导致它在最低有效位中关闭。

有关待进一步讨论和两种替代方法，参见 ASPN cookbook recipes for accurate floating point summation。

2.6 新版功能.

math.**isinf**(*x*)
Check if the float *x* is positive or negative infinity.

2.6 新版功能.

math.**isnan**(*x*)
Check if the float *x* is a NaN (not a number). For more information on NaNs, see the IEEE 754 standards.

2.6 新版功能.

math.**ldexp**(*x*, *i*)
返回 `x * (2**i)`。这基本上是函数*frexp()* 的反函数。

math.**modf**(*x*)
返回 *x* 的小数和整数部分。两个结果都带有 *x* 的符号并且是浮点数。

math.**trunc**(*x*)
Return the *Real* value *x* truncated to an *Integral* (usually a long integer). Uses the __trunc__ method.

2.6 新版功能.

注意*frexp()* 和*modf()* 具有与它们的 C 等价函数不同的调用/返回模式：它们采用单个参数并返回一对值，而不是通过 '输出形参' 返回它们的第二个返回参数（Python 中没有这样的东西）。

对于*ceil()*，*floor()* 和*modf()* 函数，请注意 所有足够大的浮点数都是精确整数。Python 浮点数通常不超过 53 位的精度（与平台 C double 类型相同），在这种情况下，任何浮点 *x* 与 `abs(x) >= 2**52` 必然没有小数位。

9.2.2 幂函数与对数函数

math.**exp**(*x*)
Return `e**x`.

math.**expm1**(*x*)
Return `e**x - 1`. For small floats *x*, the subtraction in `exp(x) - 1` can result in a significant loss of precision; the *expm1()* function provides a way to compute this quantity to full precision:

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

2.7 新版功能.

math.**log**(*x*[, *base*])
使用一个参数，返回 *x* 的自然对数（底为 *e* ）。

使用两个参数，返回给定的 *base* 的对数 *x* ，计算为 `log(x)/log(base)` 。

在 2.3 版更改: *base* argument added.

math.**log1p**(*x*)
返回 *1+x* (base *e*) 的自然对数。以对于接近零的 *x* 精确的方式计算结果。

2.6 新版功能.

math.**log10**(*x*)
返回 *x* 底为 10 的对数。这通常比 `log(x, 10)` 更准确。

math.**pow**(*x*, *y*)
将返回 `x` 的 `y` 次幂。特殊情况尽可能遵循 C99 标准的附录’F’。特别是，`pow(1.0, x)` 和 `pow(x, 0.0)` 总是返回 `1.0` ，即使 `x` 是零或 NaN。如果 `x` 和 `y` 都是有限的，`x` 是负数，`y` 不是整数那么 `pow(x, y)` 是未定义的，并且引发*ValueError* 。

与内置的 `**` 运算符不同，*math.pow()* 将其参数转换为*float* 类型。使用 `**` 或内置的*pow()* 函数来计算精确的整数幂。

在 2.6 版更改: The outcome of `1**nan` and `nan**0` was undefined.

math.**sqrt**(*x*)
返回 *x* 的平方根。

9.2.3 三角函数

math.**acos**(*x*)
以弧度为单位返回 *x* 的反余弦值。

math.**asin**(*x*)
以弧度为单位返回 *x* 的反正弦值。

math.**atan**(*x*)
以弧度为单位返回 *x* 的反正切值。

math.**atan2**(*y*, *x*)
以弧度为单位返回 `atan(y / x)` 。结果是在 `-pi` 和 `pi` 之间。从原点到点 `(x, y)` 的平面矢量使该角度与正 X 轴成正比。*atan2()* 的点的两个输入的符号都是已知的，因此它可以计算角度的正确象限。例如，`atan(1)` 和 `atan2(1, 1)` 都是 `pi/4` ，但 `atan2(-1, -1)` 是 `-3*pi/4` 。

math.**cos**(*x*)
返回 *x* 弧度的余弦值。

math.**hypot**(*x*, *y*)
返回欧几里德范数，`sqrt(x*x + y*y)`。这是从原点到点 `(x, y)` 的向量长度。

math.**sin**(*x*)
返回 *x* 弧度的正弦值。

math.**tan**(*x*)
返回 *x* 弧度的正切值。

9.2.4 角度转换

math.**degrees**(*x*)
将角度 *x* 从弧度转换为度数。

math.**radians**(*x*)
将角度 *x* 从度数转换为弧度。

9.2.5 双曲函数

math.**acosh**(*x*)
返回 *x* 的反双曲余弦值。

2.6 新版功能.

math.**asinh**(*x*)
返回 *x* 的反双曲正弦值。

2.6 新版功能.

math.**atanh**(*x*)
返回 *x* 的反双曲正切值。

2.6 新版功能.

math.**cosh**(*x*)
返回 *x* 的双曲余弦值。

math.**sinh**(*x*)
返回 *x* 的双曲正弦值。

math.**tanh**(*x*)
返回 *x* 的双曲正切值。

9.2.6 特殊函数

math.**erf**(*x*)
Return the error function at *x*.

2.7 新版功能.

math.**erfc**(*x*)
Return the complementary error function at *x*.

2.7 新版功能.

math.**gamma**(*x*)
Return the Gamma function at *x*.

2.7 新版功能.

math.**lgamma**(*x*)
返回 Gamma 函数在 *x* 绝对值的自然对数。

2.7 新版功能.

9.2.7 常量

math.**pi**
The mathematical constant π = 3.141592…, to available precision.

math.**e**
The mathematical constant e = 2.718281…, to available precision.

math 模块主要包含围绕平台 C 数学库函数的简单包装器。特殊情况下的行为在适当情况下遵循 C99 标准的附录 F。当前的实现将引发*ValueError* 用于无效操作，如 `sqrt(-1.0)` 或 `log(0.0)` （其中 C99 附件 F 建议发出无效操作信号或被零除），和*OverflowError* 用于溢出的结果（例如，`exp(1000.0)` ）。除非一个或多个输入参数是 NaN，否则不会从上述任何函数返回 NaN；在这种情况下，大多数函数将返回一个 NaN，但是（再次遵循 C99 附件 F）这个规则有一些例外，例如 `pow(float('nan'), 0.0)` 或 `hypot(float('nan'), float('inf'))` 。

请注意，Python 不会将显式 NaN 与静默 NaN 区分开来，并且显式 NaN 的行为仍未明确。典型的行为是将所有 NaN 视为静默的。

在 2.6 版更改: Behavior in special cases now aims to follow C99 Annex F. In earlier versions of Python the behavior in special cases was loosely specified.

参见:

***cmath* 模块** 这里很多函数的复数版本。

9.3 cmath ——关于复数的数学函数

This module is always available. It provides access to mathematical functions for complex numbers. The functions in this module accept integers, floating-point numbers or complex numbers as arguments. They will also accept any Python object that has either a `__complex__()` or a `__float__()` method: these methods are used to convert the object to a complex or floating-point number, respectively, and the function is then applied to the result of the conversion.

注解: 在具有对于有符号零的硬件和系统级支持的平台上，涉及分歧点的函数在分歧点的两侧都是连续的：零的符号可用来区别分歧点的一侧和另一侧。在不支持有符号零的平台上，连续性的规则见下文。

9.3.1 到极坐标和从极坐标的转换

使用 矩形坐标或 笛卡尔坐标在内部存储 Python 复数 z。这完全取决于它的 实部 z.real 和 虚部 z.imag。换句话说:

```
z == z.real + z.imag*1j
```

极坐标提供了另一种复数的表示方法。在极坐标中，一个复数 *z* 由模量 *r* 和相位角 *phi* 来定义。模量 *r* 是从 *z* 到坐标原点的距离，而相位角 *phi* 是以弧度为单位的，逆时针的，从正 X 轴到连接原点和 *z* 的线段间夹角的角度。

下面的函数可用于原生直角坐标与极坐标的相互转换。

cmath.**phase**(*x*)
Return the phase of *x* (also known as the *argument* of *x*), as a float. `phase(x)` is equivalent to `math.atan2(x.imag, x.real)`. The result lies in the range [-π, π], and the branch cut for this operation lies along the negative real axis, continuous from above. On systems with support for signed zeros (which includes most systems in current use), this means that the sign of the result is the same as the sign of `x.imag`, even when `x.imag` is zero:

```
>>> phase(complex(-1.0, 0.0))
3.1415926535897931
>>> phase(complex(-1.0, -0.0))
-3.1415926535897931
```

2.6 新版功能.

注解: 一个复数 *x* 的模数（绝对值）可以通过内置函数*abs()* 计算。没有单独的*cmath* 模块函数用于这个操作。

---

cmath.**polar**(*x*)
在极坐标中返回 *x* 的表达方式。返回一个数对 `(r, phi)`，*r* 是 *x* 的模数，*phi* 是 *x* 的相位角。`polar(x)` 相当于 `(abs(x), phase(x))`。

2.6 新版功能.

cmath.**rect**(*r*, *phi*)
通过极坐标的 *r* 和 *phi* 返回复数 *x*。相当于 `r * (math.cos(phi) + math.sin(phi)*1j)`。

2.6 新版功能.

9.3.2 幂函数与对数函数

cmath.**exp**(*x*)
Return the exponential value `e**x`.

cmath.**log**(*x*[, *base*])
返回给定 *base* 的 *x* 的对数。如果没有给定 *base*，返回 *x* 的自然对数。从 0 到 -∞ 存在一个分歧点，沿负实轴之上连续。

在 2.4 版更改: *base* argument added.

cmath.**log10**(*x*)
返回底数为 10 的 *x* 的对数。它具有与*log()* 相同的分歧点。

cmath.**sqrt**(*x*)
返回 *x* 的平方根。它具有与*log()* 相同的分歧点。

9.3.3 三角函数

cmath.**acos**(*x*)
返回 *x* 的反余弦。这里有两个分歧点：一个沿着实轴从 1 向右延伸到 ∞，从下面连续延伸。另外一个沿着实轴从 -1 向左延伸到 -∞，从上面连续延伸。

cmath.**asin**(*x*)
返回 *x* 的反正弦。它与*acos()* 有相同的分歧点。

cmath.**atan**(*x*)
返回 *x* 的反正切。它具有两个分歧点：一个沿着虚轴从 `1j` 延伸到 `∞j`，向右持续延伸。另一个是沿着虚轴从 `-1j` 延伸到 `-∞j` ，向左持续延伸。

在 2.6 版更改: direction of continuity of upper cut reversed

cmath.**cos**(*x*)
返回 *x* 的余弦。

cmath.**sin**(*x*)
返回 *x* 的正弦。

cmath.**tan**(*x*)
返回 *x* 的正切。

9.3.4 双曲函数

cmath.**acosh**(*x*)
返回 *x* 的反双曲余弦。它有一个分歧点沿着实轴从 1 到 -∞ 向左延伸，从上方持续延伸。

cmath.**asinh**(*x*)
返回 *x* 的反双曲正弦。它有两个分歧点：一个沿着虚轴从 `1j` 向右持续延伸到 `∞j`。另一个是沿着虚轴从 `-1j` 向左持续延伸到 `-∞j`。

在 2.6 版更改: branch cuts moved to match those recommended by the C99 standard

cmath.**atanh**(*x*)
返回 *x* 的反双曲正切。它有两个分歧点：一个是沿着实轴从 `1` 延展到 ∞，从下面持续延展。另一个是沿着实轴从 `-1` 延展到 -∞，从上面持续延展。

在 2.6 版更改: direction of continuity of right cut reversed

cmath.**cosh**(*x*)
返回 *x* 的双曲余弦值。

cmath.**sinh**(*x*)
返回 *x* 的双曲正弦值。

cmath.**tanh**(*x*)
返回 *x* 的双曲正切值。

9.3.5 分类函数

cmath.**isinf**(*x*)
Return `True` if the real or the imaginary part of x is positive or negative infinity.

2.6 新版功能.

cmath.**isnan**(*x*)
Return `True` if the real or imaginary part of x is not a number (NaN).

2.6 新版功能.

9.3.6 常量

cmath.**pi**
数学常数 π，作为一个浮点数。

cmath.**e**
数学常数 e，作为一个浮点数。

请注意，函数的选择与模块*math*中的函数选择相似，但不完全相同。拥有两个模块的原因是因为有些用户对复数不感兴趣，甚至根本不知道它们是什么。它们宁愿 `math.sqrt(-1)` 引发异常，也不想返回一个复数。另请注意，被*cmath*定义的函数始终会返回一个复数，尽管答案可以表示为一个实数（在这种情况下，复数的虚数部分为零）。

关于分歧点的注释：它们是沿着给定函数无法连续的曲线。它们是许多复杂函数的必要特征。假设您需要使用复杂函数进行计算，您将了解分歧点。请参阅几乎所有关于复杂变量的（不太基本）的书来进行启发。关于分歧点数值目的的正确选择信息，应提供以下良好参考：

参见:

Kahan, W: Branch cuts for complex elementary functions; or, Much ado about nothing’s sign bit. In Iserles, A., and Powell, M. (eds.), The state of the art in numerical analysis. Clarendon Press (1987) pp165–211.

9.4 decimal —十进制定点和浮点运算

2.4 新版功能.

The *decimal* module provides support for decimal floating point arithmetic. It offers several advantages over the *float* datatype:

- Decimal 类型的“设计是基于考虑人类习惯的浮点数模型，并且因此具有以下最高指导原则——计算机必须提供与人们在学校所学习的算术相一致的算术。”——摘自 decimal 算术规范描述。
- Decimal numbers can be represented exactly. In contrast, numbers like `1.1` and `2.2` do not have exact representations in binary floating point. End users typically would not expect `1.1 + 2.2` to display as `3.3000000000000003` as it does with binary floating point.
- 精确性会延续到算术类操作中。对于 decimal 浮点数，`0.1 + 0.1 + 0.1 - 0.3` 会精确地等于零。而对于二进制浮点数，结果则为 `5.5511151231257827e-017` 。虽然接近于零，但其中的误差将妨碍可靠的相等性检验，并且误差还会不断累积。因此，decimal 更适合具有严格相等不变性要求的会计类应用。
- decimal 模块包含了有效位的概念，使得 `1.30 + 1.20` 是 `2.50` 。保留尾随零以表示有效位。这是货币类应用的习惯表示法。对于乘法，“教科书”方式使用被乘数中的所有数位。例如，`1.3 * 1.2` 给出 `1.56` 而 `1.30 * 1.20` 给出 `1.5600` 。

- 与基于硬件的二进制浮点数不同，decimal 模块具有用户可更改的精度（默认为 28 位），可以与给定问题所需的一样大：

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- 二进制和 decimal 浮点数都是根据已发布的标准实现的。虽然内置浮点类型只公开其功能的一小部分，但 decimal 模块公开了标准的所有必需部分。在需要时，程序员可以完全控制舍入和信号处理。这包括通过使用异常来阻止任何不精确操作来强制执行精确算术的选项。
- decimal 模块旨在支持“无偏差，精确无舍入的十进制算术（有时称为定点数算术）和有舍入的浮点数算术”。——摘自 decimal 算术规范说明。

该模块的设计以三个概念为中心：decimal 数值，算术上下文和信号。

decimal 数值是不可变对象。它由符号，系数和指数位组成。为了保持有效位，系数位不会截去末尾零。decimal 数值也包括特殊值例如 `Infinity` ，`-Infinity` 和 `NaN` 。该标准还区分 `-0` 和 `+0` 。

算术的上下文是指定精度、舍入规则、指数限制、指示操作结果的标志以及确定符号是否被视为异常的陷阱启用器的环境。舍入选项包括 `ROUND_CEILING` 、`ROUND_DOWN` 、`ROUND_FLOOR` 、`ROUND_HALF_DOWN`, `ROUND_HALF_EVEN` 、`ROUND_HALF_UP` 、`ROUND_UP` 以及 `ROUND_05UP`.

Signals are groups of exceptional conditions arising during the course of computation. Depending on the needs of the application, signals may be ignored, considered as informational, or treated as exceptions. The signals in the decimal module are: *Clamped*, *InvalidOperation*, *DivisionByZero*, *Inexact*, *Rounded*, *Subnormal*, *Overflow*, and *Underflow*.

对于每个信号，都有一个标志和一个陷阱启动器。遇到信号时，其标志设置为 1 ，然后，如果陷阱启用器设置为 1 ，则引发异常。标志是粘性的，因此用户需要在监控计算之前重置它们。

参见：

- IBM’s General Decimal Arithmetic Specification, The General Decimal Arithmetic Specification.

9.4.1 快速入门教程

通常使用 decimal 的方式是先导入该模块，通过*getcontext()* 查看当前上下文，并在必要时为精度、舍入或启用的陷阱设置新值：

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,
        capitals=1, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7       # Set a new precision
```

Decimal instances can be constructed from integers, strings, floats, or tuples. Construction from an integer or a float performs an exact conversion of the value of that integer or float. Decimal numbers include special values such as `NaN` which stands for “Not a number”, positive and negative `Infinity`, and `-0`.

```
>>> getcontext().prec = 28
>>> Decimal(10)
```

(下页继续)

(续上页)

```
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.41421356237')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.414213562373095048801688724')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

新 Decimal 的重要性仅由输入的位数决定。上下文精度和舍入仅在算术运算期间发挥作用。

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

Decimal 数字能很好地与 Python 的其余部分交互。以下是一个小小的 decimal 浮点数飞行马戏团：

```
>>> data = map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split())
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)      # round() first converts to binary floating point
1.3
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

Decimal 也可以使用一些数学函数：

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.4142135623730950488016887242')
>>> Decimal(1).exp()
Decimal('2.7182818284590452353602874714')
>>> Decimal('10').ln()
Decimal('2.3025850929940456840179914547')
>>> Decimal('10').log10()
Decimal('1')
```

`quantize()` 方法将数字四舍五入为固定指数。此方法对于将结果舍入到固定的位置的货币应用程序非常有用：

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

如上所示，*getcontext()* 函数访问当前上下文并允许更改设置。这种方法满足大多数应用程序的需求。

对于更高级的工作，使用 Context() 构造函数创建备用上下文可能很有用。要使用备用活动，请使用*setcontext()* 函数。

根据标准，*decimal* 模块提供了两个现成的标准上下文*BasicContext* 和*ExtendedContext* 。前者对调试特别有用，因为许多陷阱都已启用：

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,
        capitals=1, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

上下文还具有用于监视计算期间遇到的异常情况的信号标志。标志保持设置直到明确清除，因此最好通过使用 `clear_flags()` 方法清除每组受监控计算之前的标志。：

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,
        capitals=1, flags=[Rounded, Inexact], traps=[])
```

flags 条目显示对 Pi 的有理逼近被舍入（超出上下文精度的数字被抛弃）并且结果是不精确的（一些丢弃的数字不为零）。

使用上下文的 traps 字段中的字典设置单个陷阱：

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

大多数程序仅在程序开始时调整当前上下文一次。并且，在许多应用程序中，数据在循环内单个强制转换为*Decimal* 。通过创建上下文集和小数，程序的大部分操作数据与其他 Python 数字类型没有区别。

9.4.2 Decimal 对象

class decimal.**Decimal**([*value*[, *context*]])

根据 *value* 构造一个新的*Decimal* 对象。

value can be an integer, string, tuple, *float*, or another *Decimal* object. If no *value* is given, returns Decimal('0'). If *value* is a string, it should conform to the decimal numeric string syntax after leading and trailing whitespace characters are removed:

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

If *value* is a unicode string then other Unicode decimal digits are also permitted where digit appears above. These include decimal digits from various other alphabets (for example, Arabic-Indic and Devanāgarī digits) along with the fullwidth digits u'\uff10' through u'\uff19'.

如果 *value* 是一个*tuple* ，它应该有三个组件，一个符号（0 表示正数或 1 表示负数），一个数字的*tuple* 和整数指数。例如，Decimal((0, (1, 4, 1, 4), -3)) 返回 Decimal('1.414')。

如果 *value* 是*float* ，则二进制浮点值无损地转换为其精确的十进制等效值。此转换通常需要 53 位或更多位数的精度。例如，Decimal(float('1.1')) 转换为 “Decimal(‘1.1000000000000000088817841970012523233890533447265625’)“。

context 精度不会影响存储的位数。这完全由 *value* 中的位数决定。例如，Decimal('3.00000') 记录所有五个零，即使上下文精度只有三。

context 参数的目的是确定 *value* 是格式错误的字符串时该怎么做。如果上下文陷阱*InvalidOperation*，则引发异常；否则，构造函数返回一个新的 Decimal，其值为 NaN。

构造完成后，*Decimal* 对象是不可变的。

在 2.6 版更改: leading and trailing whitespace characters are permitted when creating a Decimal instance from a string.

在 2.7 版更改: 现在允许构造函数的参数为*float* 实例。

Decimal floating point objects share many properties with the other built-in numeric types such as *float* and *int*. All of the usual math operations and special methods apply. Likewise, decimal objects can be copied, pickled, printed, used as dictionary keys, used as set elements, compared, sorted, and coerced to another type (such as *float* or *long*).

算术对十进制对象和算术对整数和浮点数有一些小的差别。当余数运算符 `%` 应用于 Decimal 对象时，结果的符号是 被除数的符号，而不是除数的符号:

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

整数除法运算符 `//` 的行为类似，返回真商的整数部分（截断为零）而不是它的向下取整，以便保留通常的标识 `x == (x // y) * y + x % y`:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

`%` 和 `//` 运算符实现了 `remainder` 和 `divide-integer` 操作（分别），如规范中所述。

Decimal objects cannot generally be combined with floats in arithmetic operations: an attempt to add a *Decimal* to a *float*, for example, will raise a *TypeError*. There's one exception to this rule: it's possible to use Python's comparison operators to compare a *float* instance `x` with a *Decimal* instance `y`. Without this exception, comparisons between *Decimal* and *float* instances would follow the general rules for comparing objects of different types described in the expressions section of the reference manual, leading to confusing results.

在 2.7 版更改: A comparison between a *float* instance `x` and a *Decimal* instance `y` now returns a result based on the values of `x` and `y`. In earlier versions `x < y` returned the same (arbitrary) result for any *Decimal* instance `x` and any *float* instance `y`.

除了标准的数字属性，十进制浮点对象还有许多专门的方法:

adjusted()
: 在移出系数最右边的数字之后返回调整后的指数，直到只剩下前导数字: `Decimal('321e+5').adjusted()` 返回 7 。用于确定最高有效位相对于小数点的位置。

as_tuple()
: 返回一个*named tuple* 表示的数字: `DecimalTuple(sign, digits, exponent)`。

  在 2.6 版更改: Use a named tuple.

canonical()
: 返回参数的规范编码。目前，一个*Decimal* 实例的编码始终是规范的，因此该操作返回其参数不变。

  2.6 新版功能.

compare(*other*[, *context*])
: Compare the values of two Decimal instances. This operation behaves in the same way as the usual comparison method `__cmp__()`, except that *compare()* returns a Decimal instance rather than an integer, and if either operand is a NaN then the result is a NaN:

```
a or b is a NaN ==> Decimal('NaN')
a < b           ==> Decimal('-1')
a == b          ==> Decimal('0')
a > b           ==> Decimal('1')
```

compare_signal(*other*[, *context*])
除了所有 NaN 信号之外，此操作与*compare()* 方法相同。也就是说，如果两个操作数都不是信令 NaN，那么任何静默的 NaN 操作数都被视为信令 NaN。

2.6 新版功能.

compare_total(*other*)
使用它们的抽象表示而不是它们的数值来比较两个操作数。类似于*compare()* 方法，但结果给出了一个总排序*Decimal* 实例。两个*Decimal* 实例具有相同的数值但不同的表示形式在此排序中比较不相等：

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

静默和发出信号的 NaN 也包括在总排序中。这个函数的结果是 `Decimal('0')` 如果两个操作数具有相同的表示，或是 `Decimal('-1')` 如果第一个操作数的总顺序低于第二个操作数，或是 `Decimal('1')` 如果第一个操作数在总顺序中高于第二个操作数。有关总排序的详细信息，请参阅规范。

2.6 新版功能.

compare_total_mag(*other*)
比较两个操作数使用它们的抽象表示而不是它们的值，如*compare_total()*，但忽略每个操作数的符号。`x.compare_total_mag(y)` 相当于 `x.copy_abs().compare_total(y.copy_abs())`。

2.6 新版功能.

conjugate()
只返回 self，这种方法只符合 Decimal 规范。

2.6 新版功能.

copy_abs()
返回参数的绝对值。此操作不受上下文影响并且是静默的：没有更改标志且不执行舍入。

2.6 新版功能.

copy_negate()
回到参数的否定。此操作不受上下文影响并且是静默的：没有标志更改且不执行舍入。

2.6 新版功能.

copy_sign(*other*)
返回第一个操作数的副本，其符号设置为与第二个操作数的符号相同。例如：

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

2.6 新版功能.

exp([*context*])
返回给定数字的（自然）指数函数 “e**x“的值。结果使用 ROUND_HALF_EVEN 舍入模式正确舍入。

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

2.6 新版功能.

from_float(*f*)
将浮点数转换为十进制数的类方法。

注意，*Decimal.from_float(0.1)* 与 *Decimal('0.1')* 不同。由于 0.1 在二进制浮点中不能精确表示，因此该值存储为最接近的可表示值，即 *0x1.999999999999ap-4* 。十进制的等效值是 '0.1000000000000000055511151231257827021181583404541015625'。

注解： From Python 2.7 onwards, a `Decimal` instance can also be constructed directly from a `float`.

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

2.7 新版功能.

fma(*other*, *third*[, *context*])
混合乘法加法。返回 self*other+third ， 中间乘积 self*other 没有四舍五入。

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

2.6 新版功能.

is_canonical()
如果参数是规范的，则为返回`True`，否则为`False` 。目前，`Decimal` 实例总是规范的，所以这个操作总是返回`True` 。

2.6 新版功能.

is_finite()
如果参数是一个有限的数，则返回为`True` ； 如果参数为无穷大或 NaN ， 则返回为`False`。

2.6 新版功能.

is_infinite()
如果参数为正负无穷大，则返回为`True` ， 否则为`False` 。

2.6 新版功能.

is_nan()
如果参数为 NaN （无论是否静默），则返回为`True` ， 否则为`False` 。

2.6 新版功能.

is_normal()
Return `True` if the argument is a *normal* finite non-zero number with an adjusted exponent greater than or equal to *Emin*. Return `False` if the argument is zero, subnormal, infinite or a NaN. Note, the term *normal* is used here in a different sense with the `normalize()` method which is used to create canonical values.

2.6 新版功能.

is_qnan()
如果参数为静默 NaN，返回`True`，否则返回`False`。

2.6 新版功能.

is_signed()
如果参数带有负号，则返回为*True*，否则返回*False*。注意，0 和 NaN 都可带有符号。

2.6 新版功能.

is_snan()
如果参数为显式 NaN，则返回*True*，否则返回*False*。

2.6 新版功能.

is_subnormal()
Return *True* if the argument is subnormal, and *False* otherwise. A number is subnormal is if it is nonzero, finite, and has an adjusted exponent less than *Emin*.

2.6 新版功能.

is_zero()
如果参数是 0（正负皆可），则返回*True*，否则返回*False*。

2.6 新版功能.

ln([*context*])
返回操作数的自然对数（以 e 为底）。结果是使用 ROUND_HALF_EVEN 舍入模式正确四舍五入的。

2.6 新版功能.

log10([*context*])
返回操作数的以十为底的对数。结果是使用 ROUND_HALF_EVEN 舍入模式正确四舍五入的。

2.6 新版功能.

logb([*context*])
对于一个非零数，返回其运算数的调整后指数作为一个*Decimal* 实例。如果运算数为零将返回 `Decimal('-Infinity')` 并且产生 the *DivisionByZero* 标志。如果运算数是无限大则返回 `Decimal('Infinity')` 。

2.6 新版功能.

logical_and(*other*[, *context*])
logical_and() 是需要两个 *逻辑运算数*的逻辑运算（参考*逻辑操作数* ）。结果是按位输出的两运算数的“和”。

2.6 新版功能.

logical_invert([*context*])
logical_invert() 是一个逻辑运算。结果是按位的倒转的运算数。

2.6 新版功能.

logical_or(*other*[, *context*])
logical_or() 是需要两个 *logical operands* 的逻辑运算（请参阅*逻辑操作数* ）。结果是两个运算数的按位的 `or` 。

2.6 新版功能.

logical_xor(*other*[, *context*])
logical_xor() 是需要两个 *逻辑运算数*的逻辑运算（参考*逻辑操作数* ）。结果是按位输出的两运算数的异或运算。

2.6 新版功能.

max(*other*[, *context*])
像 `max(self, other)` 一样，除了在返回之前应用上下文舍入规则并且用信号通知或忽略 NaN 值（取决于上下文以及它们是发信号还是安静）。

max_mag(*other*[, *context*])
与*max()* 方法相似，但是操作数使用绝对值进行比较。

2.6 新版功能.

min(*other*[, *context*])
像 min(self, other) 一样，除了在返回之前应用上下文舍入规则并且用信号通知或忽略 NaN 值（取决于上下文以及它们是发信号还是安静）。

min_mag(*other*[, *context*])
与*min()* 方法相似，但是操作数使用绝对值进行比较。

2.6 新版功能.

next_minus([*context*])
返回小于给定操作数的上下文中可表示的最大数字（或者当前线程的上下文中的可表示的最大数字如果没有给定上下文）。

2.6 新版功能.

next_plus([*context*])
返回大于给定操作数的上下文中可表示的最小数字（或者当前线程的上下文中的可表示的最小数字如果没有给定上下文）。

2.6 新版功能.

next_toward(*other*[, *context*])
如果两运算数不相等，返回在第二个操作数的方向上最接近第一个操作数的数。如果两操作数数值上相等，返回将符号设置为与第二个运算数相同的第一个运算数的拷贝。

2.6 新版功能.

normalize([*context*])
通过去除尾随的零并将所有结果等于 Decimal('0') 的转化为 Decimal('0e0') 来标准化数字。用于为等效类的属性生成规范值。比如，Decimal('32.100') 和 Decimal('0.321000e+2') 都被标准化为相同的值 Decimal('32.1')。

number_class([*context*])
返回一个字符串描述运算数的 *class* 。返回值是以下十个字符串中的一个。

- "-Infinity"，指示操作数为负无穷大。
- "-Normal"，指示该操作数是负正常数字。
- "-Subnormal"，指示该操作数是负的次正规数。
- "-Zero"，指示该操作数是负零。
- "-Zero"，指示该操作数是正零。
- "+Subnormal"，指示该操作数是正的次正规数。
- "+Normal"，指示该操作数是正的正规数。
- "+Infinity"，指示该运算数是正无穷。
- "NaN"，指示该运算数是沉寂的 NaN （非数字）。
- "sNaN"，指示该运算数是信号 NaN 。

2.6 新版功能.

quantize(*exp*[, *rounding*[, *context*[, *watchexp*]]])
返回的值等于四舍五入的第一个运算数并且具有第二个操作数的指数。

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

与其他运算不同，如果量化运算后的系数长度大于精度，那么会发出一个*InvalidOperation*信号。这保证了除非有一个错误情况，量化指数恒等于右手运算数的指数。

与其他运算不同，量化永不信号下溢，即使结果不正常且不精确。

如果第二个运算数的指数大于第一个运算数的指数那或许需要四舍五入。在这种情况下，舍入模式由给定 `rounding` 参数决定，其余的由给定 `context` 参数决定；如果参数都未给定，使用当前线程上下文的舍入模式。

If *watchexp* is set (default), then an error is returned whenever the resulting exponent is greater than `Emax` or less than `Etiny`.

radix()

返回 `Decimal(10)`，即*Decimal*类进行所有算术运算所用的数制（基数）。这是为保持与规范描述的兼容性而加入的。

2.6 新版功能.

remainder_near(*other*[, *context*])

返回 *self* 除以 *other* 的余数。这与 `self % other` 的区别在于所选择的余数要使其绝对值最小化。更准确地说，返回值为 `self - n * other` 其中 `n` 是最接近 `self / other` 的实际值的整数，并且如果两个整数与实际值的差相等则会选择其中的偶数。

如果结果为零则其符号将为 *self* 的符号。

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate(*other*[, *context*])

返回对第一个操作数的数码按第二个操作数所指定的数量进行轮转的结果。第二个操作数必须为 -precision 至 precision 精度范围内的整数。第二个操作数的绝对值给出要轮转的位数。如果第二个操作数为正值则向左轮转；否则向右轮转。如有必要第一个操作数的系数会在左侧填充零以达到 precision 所指定的长度。第一个操作数的符号和指数保持不变。

2.6 新版功能.

same_quantum(*other*[, *context*])

检测自身与 other 是否具有相同的指数或是否均为 `NaN`。

scaleb(*other*[, *context*])

返回第一个操作数使用第二个操作数对指数进行调整的结果。等价于返回第一个操作数乘以 `10**other` 的结果。第二个操作数必须为整数。

2.6 新版功能.

shift(*other*[, *context*])

返回第一个操作数的数码按第二个操作数所指定的数量进行移位的结果。第二个操作数必须为 -precision 至 precision 范围内的整数。第二个操作数的绝对值给出要移动的位数。如果第二个操作数为正值则向左移位；否则向右移位。移入系数的数码为零。第一个操作数的符号和指数保持不变。

2.6 新版功能.

sqrt([*context*])
返回参数的平方根精确到完整精度。

to_eng_string([*context*])
转换为字符串，如果需要指数则会使用工程标注法。

工程标注法的指数是 3 的倍数。这会在十进制位的左边保留至多 3 个数码，并可能要求添加一至两个末尾零。

例如，此方法会将 `Decimal('123E+1')` 转换为 `Decimal('1.23E+3')`。

to_integral([*rounding*[, *context*]])
与*to_integral_value()* 方法相同。保留 `to_integral` 名称是为了与旧版本兼容。

to_integral_exact([*rounding*[, *context*]])
舍入到最接近的整数，发出信号*Inexact* 或者如果发生舍入则相应地发出信号*Rounded*。如果给出 `rounding` 形参则由其确定舍入模式，否则由给定的 `context` 来确定。如果没有给定任何形参则会使用当前上下文的舍入模式。

2.6 新版功能.

to_integral_value([*rounding*[, *context*]])
舍入到最接近的整数而不发出*Inexact* 或*Rounded* 信号。如果给出 *rounding* 则会应用其所指定的舍入模式；否则使用所提供的 *context* 或当前上下文的舍入方法。

在 2.6 版更改: renamed from `to_integral` to `to_integral_value`. The old name remains valid for compatibility.

逻辑操作数

`logical_and()`, `logical_invert()`, `logical_or()` 和 `logical_xor()` 方法期望其参数为 *逻辑操作数*。*逻辑操作数*是指数位与符号位均为零的*Decimal* 实例，并且其数字位均为 `0` 或 `1`。

9.4.3 Context 对象

上下文是算术运算所在的环境。它们管理精度、设置舍入规则、确定将哪些信号视为异常，并限制指数的范围。

每个线程都有自己的当前上下文，可使用*getcontext()* 和*setcontext()* 函数来读取或修改：

`decimal`.**getcontext**()
返回活动线程的当前上下文。

`decimal`.**setcontext**(*c*)
将活动线程的当前上下文设为 *c*。

Beginning with Python 2.5, you can also use the `with` statement and the *localcontext()* function to temporarily change the active context.

`decimal`.**localcontext**([*c*])
Return a context manager that will set the current context for the active thread to a copy of *c* on entry to the with-statement and restore the previous context when exiting the with-statement. If no context is specified, a copy of the current context is used.

2.5 新版功能.

例如，以下代码会将当前 decimal 精度设为 42 位，执行一个运算，然后自动恢复之前的上下文：

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42   # Perform a high precision calculation
    s = calculate_something()
s = +s  # Round the final result back to the default precision

with localcontext(BasicContext):      # temporarily use the BasicContext
    print Decimal(1) / Decimal(7)
    print Decimal(355) / Decimal(113)
```

新的上下文也可使用下述的*Context* 构造器来创建。此外，模块还提供了三种预设的上下文:

class decimal.**BasicContext**
这是由通用十进制算术规范描述所定义的标准上下文。精度设为九。舍入设为 ROUND_HALF_UP。清除所有旗标。启用所有陷阱（视为异常），但*Inexact*, *Rounded* 和*Subnormal* 除外。

由于启用了许多陷阱，此上下文适用于进行调试。

class decimal.**ExtendedContext**
这是由通用十进制算术规范描述所定义的标准上下文。精度设为九。舍入设为 ROUND_HALF_EVEN。清除所有旗标。不启用任何陷阱（因此在计算期间不会引发异常）。

由于禁用了陷阱，此上下文适用于希望结果值为 NaN 或 Infinity 而不是引发异常的应用。这允许应用在出现当其他情况下会中止程序的条件时仍能完成运行。

class decimal.**DefaultContext**
此上下文被*Context* 构造器用作新上下文的原型。改变一个字段（例如精度）的效果将是改变*Context* 构造器所创建的新上下文的默认值。

此上下文最适用于多线程环境。在线程开始前改变一个字段具有设置全系统默认值的效果。不推荐在线程开始后改变字段，因为这会要求线程同步避免竞争条件。

在单线程环境中，最好完全不使用此上下文。而是简单地电显式创建上下文，具体如下所述。

The default values are precision=28, rounding=ROUND_HALF_EVEN, and enabled traps for Overflow, InvalidOperation, and DivisionByZero.

在已提供的三种上下文之外，还可以使用*Context* 构造器创建新的上下文。

class decimal.**Context**(*prec=None*, *rounding=None*, *traps=None*, *flags=None*, *Emin=None*, *Emax=None*, *capitals=1*)
创建一个新上下文。如果某个字段未指定或为*None*，则从*DefaultContext* 拷贝默认值。如果 *flags* 字段未指定或为*None*，则清空所有旗标。

The *prec* field is a positive integer that sets the precision for arithmetic operations in the context.

The *rounding* option is one of:

- ROUND_CEILING (towards Infinity),
- ROUND_DOWN (towards zero),
- ROUND_FLOOR (towards -Infinity),
- ROUND_HALF_DOWN (to nearest with ties going towards zero),
- ROUND_HALF_EVEN (to nearest with ties going to nearest even integer),
- ROUND_HALF_UP (to nearest with ties going away from zero), or
- ROUND_UP (away from zero).

- ROUND_05UP (away from zero if last digit after rounding towards zero would have been 0 or 5; otherwise towards zero)

traps 和 *flags* 字段列出要设置的任何信号。通常，新上下文应当只设置 traps 而让 flags 为空。

The *Emin* and *Emax* fields are integers specifying the outer limits allowable for exponents.

capitals 字段为 0 或 1 (默认值)。如果设为 1，指数将附带打印大写的 E；其他情况则将使用小写的 e: Decimal('6.02e+23')。

在 2.6 版更改: The ROUND_05UP rounding mode was added.

The *Context* class defines several general purpose methods as well as a large number of methods for doing arithmetic directly in a given context. In addition, for each of the *Decimal* methods described above (with the exception of the adjusted() and as_tuple() methods) there is a corresponding *Context* method. For example, for a *Context* instance C and *Decimal* instance x, C.exp(x) is equivalent to x.exp(context=C). Each *Context* method accepts a Python integer (an instance of *int* or *long*) anywhere that a Decimal instance is accepted.

clear_flags()
将所有旗标重置为 0。

copy()
返回上下文的一个副本。

copy_decimal(*num*)
返回 Decimal 实例 num 的一个副本。

create_decimal(*num*)
基于 *num* 创建一个新 Decimal 实例但使用 *self* 作为上下文。与*Decimal* 构造器不同，该上下文的精度、舍入方法、旗标和陷阱会被应用于转换过程。

此方法很有用处，因为常量往往被给予高于应用所需的精度。另一个好处在于立即执行舍入可以消除超出当前精度的数位所导致的意外效果。在下面的示例中，使用未舍入的输入意味着在总和中添加零会改变结果：

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

This method implements the to-number operation of the IBM specification. If the argument is a string, no leading or trailing whitespace is permitted.

create_decimal_from_float(*f*)
基于浮点数 *f* 创建一个新的 Decimal 实例，但会使用 *self* 作为上下文来执行舍入。与*Decimal.from_float()* 类方法不同，上下文的精度、舍入方法、旗标和陷阱会应用到转换中。

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
    ...
Inexact: None
```

2.7 新版功能.

Etiny()
返回一个等于 Emin - prec + 1 的值即次正规化结果中的最小指数值。当发生向下溢出时，指数会设为*Etiny*。

Etop()
返回一个等于 Emax - prec + 1 的值。

使用 decimal 的通常方式是创建*Decimal* 实例然后对其应用算术运算, 这些运算发生在活动线程的当前上下文中。一种替代方式则是使用上下文的方法在特定上下文中进行计算。这些方法类似于*Decimal* 类的方法，在此仅简单地重新列出。

abs(*x*)
返回 *x* 的绝对值。

add(*x*, *y*)
返回 *x* 与 *y* 的和。

canonical(*x*)
返回相同的 Decimal 对象 *x*。

compare(*x*, *y*)
对 *x* 与 *y* 进行数值比较。

compare_signal(*x*, *y*)
对两个操作数进行数值比较。

compare_total(*x*, *y*)
对两个操作数使用其抽象表示进行比较。

compare_total_mag(*x*, *y*)
对两个操作数使用其抽象表示进行比较，忽略符号。

copy_abs(*x*)
返回 *x* 的副本，符号设为 0。

copy_negate(*x*)
返回 *x* 的副本，符号取反。

copy_sign(*x*, *y*)
从 *y* 拷贝符号至 *x*。

divide(*x*, *y*)
返回 *x* 除以 *y* 的结果。

divide_int(*x*, *y*)
返回 *x* 除以 *y* 的结果，截短为整数。

divmod(*x*, *y*)
两个数字相除并返回结果的整数部分。

exp(*x*)
返回 *e* ** *x*。

fma(*x*, *y*, *z*)
返回 *x* 乘以 *y* 再加 *z* 的结果。

is_canonical(*x*)
如果 *x* 是规范的则返回 True；否则返回 False。

is_finite(*x*)
如果 *x* 为有限数则返回 “True“；否则返回 False。

is_infinite(*x*)
如果 *x* 是无限的则返回 True；否则返回 False。

is_nan(*x*)
如果 *x* 是 qNaN 或 sNaN 则返回 `True`；否则返回 `False`。

is_normal(*x*)
如果 *x* 是正规数则返回 `True`；否则返回 `False`。

is_qnan(*x*)
如果 *x* 是静默 NaN 则返回 `True`；否则返回 `False`。

is_signed(*x*)
x 是负数则返回 `True`；否则返回 `False`。

is_snan(*x*)
如果 *x* 是显式 NaN 则返回 `True`；否则返回 `False`。

is_subnormal(*x*)
如果 *x* 是次标准数则返回 `True`；否则返回 `False`。

is_zero(*x*)
如果 *x* 为零则返回 `True`；否则返回 `False`。

ln(*x*)
返回 *x* 的自然对数（以 e 为底）。

log10(*x*)
返回 *x* 的以 10 为底的对数。

logb(*x*)
返回操作数的 MSD 等级的指数。

logical_and(*x*, *y*)
在操作数的每个数码间应用逻辑运算 *and*。

logical_invert(*x*)
反转 *x* 中的所有数位。

logical_or(*x*, *y*)
在操作数的每个数位间应用逻辑运算 *or*。

logical_xor(*x*, *y*)
在操作数的每个数位间应用逻辑运算 *xor*。

max(*x*, *y*)
对两个值执行数字比较并返回其中的最大值。

max_mag(*x*, *y*)
对两个值执行忽略正负号的数字比较。

min(*x*, *y*)
对两个值执行数字比较并返回其中的最小值。

min_mag(*x*, *y*)
对两个值执行忽略正负号的数字比较。

minus(*x*)
对应于 Python 中的单目取负运算符执行取负操作。

multiply(*x*, *y*)
返回 *x* 和 *y* 的积。

next_minus(*x*)
返回小于 *x* 的最大数字表示形式。

next_plus(*x*)
返回大于 *x* 的最小数字表示形式。

next_toward(*x*, *y*)
返回 *x* 趋向于 *y* 的最接近的数字。

normalize(*x*)
将 *x* 改写为最简形式。

number_class(*x*)
返回 *x* 的类的表示。

plus(*x*)
对应于 Python 中的单目前缀取正运算符执行取正操作。此操作将应用上下文精度和舍入，因此它不是标识运算。

power(*x*, *y*[, *modulo*])
返回 x 的 y 次方，如果给出了模数 modulo 则取其余数。

With two arguments, compute x**y. If x is negative then y must be integral. The result will be inexact unless y is integral and the result is finite and can be expressed exactly in 'precision' digits. The result should always be correctly rounded, using the rounding mode of the current thread's context.

带有三个参数时，计算 (x**y) % modulo。对于三个参数的形式，参数将会应用以下限制：

- 三个参数必须都是整数
- y 必须是非负数
- x 或 y 至少有一个不为零
- modulo 必须不为零且至多有 'precision' 位

来自 Context.power(x, y, modulo) 的结果值等于使用无限精度计算 (x**y) % modulo 所得到的值，但其计算过程更高效。结果的指数为零，无论 x, y 和 modulo 的指数是多少。结果值总是完全精确的。

在 2.6 版更改: y may now be nonintegral in x**y. Stricter requirements for the three-argument version.

quantize(*x*, *y*)
返回的值等于 *x* (舍入后)，并且指数为 *y*。

radix()
恰好返回 10，因为这是 Decimal 对象:)

remainder(*x*, *y*)
返回整除所得到的余数。

结果的符号，如果不为零，则与原始除数的符号相同。

remainder_near(*x*, *y*)
返回 x - y * n，其中 *n* 为最接近 x / y 实际值的整数（如结果为 0 则其符号将与 *x* 的符号相同）。

rotate(*x*, *y*)
返回 *x* 翻转 *y* 次的副本。

same_quantum(*x*, *y*)
如果两个操作数具有相同的指数则返回 True。

scaleb(*x*, *y*)
返回第一个操作数对第二个值添加其指数后的结果。

shift(*x*, *y*)
返回 *x* 变换 *y* 次的副本。

sqrt(*x*)
非负数基于上下文精度的平方根。

subtract(*x*, *y*)
返回 *x* 和 *y* 的差。

to_eng_string(*x*)
转换为字符串，如果需要指数则会使用工程标注法。

工程标注法的指数是 3 的倍数。这会在十进制位的左边保留至多 3 个数码，并可能要求添加一至两个末尾零。

to_integral_exact(*x*)
舍入到一个整数。

to_sci_string(*x*)
使用科学计数法将一个数字转换为字符串。

9.4.4 信号

信号代表在计算期间引发的条件。每个信号对应于一个上下文旗标和一个上下文陷阱启用器。

上下文旗标将在遇到特定条件时被设定。在完成计算之后，将为了获得信息而检测旗标（例如确定计算是否精确）。在检测旗标后，请确保在开始下一次计算之前清除所有旗标。

如果为信号设定了上下文的陷阱启用器，则条件会导致特定的 Python 异常被引发。举例来说，如果设定了*DivisionByZero* 陷阱，则当遇到此条件时就将引发*DivisionByZero* 异常。

class decimal.**Clamped**
修改一个指数以符合表示限制。

通常，限位将在一个指数超出上下文的 Emin 和 Emax 限制时发生。在可能的情况下，会通过给系数添加零来将指数缩减至符合限制。

class decimal.**DecimalException**
其他信号的基类，并且也是*ArithmeticError* 的一个子类。

class decimal.**DivisionByZero**
非无限数被零除的信号。

可在除法、取余队法或对一个数求负数次幂时发生。如果此信号未被陷阱捕获，则返回 Infinity 或 -Infinity 并且由对计算的输入来确定正负符号。

class decimal.**Inexact**
表明发生了舍入且结果是不精确的。

有非零数位在舍入期间被丢弃的信号。舍入结果将被返回。此信号旗标或陷阱被用于检测结果不精确的情况。

class decimal.**InvalidOperation**
执行了一个无效的操作。

表明请求了一个无意义的操作。如未被陷阱捕获则返回 NaN。可能的原因包括:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
x._rescale( non-integer )
sqrt(-x) and x > 0
```

(下页继续)

(续上页)

```
0 ** 0
x ** (non-integer)
x ** Infinity
```

class decimal.**Overflow**
数值的溢出。

表明在发生舍入之后的指数大于 Emax。如果未被陷阱捕获，则结果将取决于舍入模式，或者向下舍入为最大的可表示有限数，或者向上舍入为 Infinity。无论哪种情况，都将引发*Inexact* 和*Rounded* 信号。

class decimal.**Rounded**
发生了舍入，但或许并没有信息丢失。

一旦舍入丢弃了数位就会发出此信号；即使被丢弃的数位是零 (例如将 5.00 舍入为 5.0)。如果未被陷阱捕获，则不经修改地返回结果。此信号用于检测有效位数的丢弃。

class decimal.**Subnormal**
在舍入之前指数低于 Emin。

当操作结果是次标准数（即指数过小）时就会发出此信号。如果未被陷阱捕获，则不经修改过返回结果。

class decimal.**Underflow**
数字向下溢出导致结果舍入到零。

当一个次标准数结果通过舍入转为零时就会发出此信号。同时还将引发*Inexact* 和*Subnormal* 信号。

以下表格总结了信号的层级结构:

```
exceptions.ArithmeticError(exceptions.StandardError)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
```

9.4.5 浮点数说明

通过提升精度来缓解舍入误差

使用十进制浮点数可以消除十进制表示错误（即能够完全精确地表示 0.1 这样的数）；然而，某些运算在非零数位超出给定的精度时仍然可能导致舍入错误。

舍入错误的影响可能因接近相互抵销的加减运算被放大从而导致损失有效位。Knuth 提供了两个指导性示例，其中出现了精度不足的浮点算术舍入，导致加法的交换律和分配律被打破:

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
```

(下页继续)

(续上页)

```
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

decimal 模块则可以通过充分地扩展精度来避免有效位的丢失:

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

特殊的值

decimal 模块的数字系统提供了一些特殊的值，包括 NaN, sNaN, -Infinity, Infinity 以及两种零值 +0 和 -0。

无穷大可以使用 Decimal('Infinity') 来构建。它们也可以在不捕获*DivisionByZero* 信号捕获时通过除以零来产生。类似地，当不捕获*Overflow* 信号时，也可以通过舍入到超出最大可表示数字限制的方式产生无穷大的结果。

无穷大是有符号的（仿射）并可用于算术运算，它们会被当作极其巨大的不确定数字来处理。例如，无穷大加一个常量结果也将为无穷大。

某些不存在有效结果的运算将会返回 NaN，或者如果捕获了*InvalidOperation* 信号则会引发一个异常。例如，0/0 会返回 NaN 表示结果“不是一个数字”。这样的 NaN 是静默产生的，并且在产生之后参与其它计算时总是会得到 NaN 的结果。这种行为对于偶而缺少输入的各类计算都很有用处—它允许在将特定结果标记为无效的同时让计算继续运行。

另一种变体形式是 sNaN，它在每次运算后会发出信号而不是保持静默。当对于无效结果需要中断计算进行特别处理时，这是一个很有用的返回值。

Python 中比较运算符的行为在涉及 NaN 时可能会令人有点惊讶。相等性检测在操作数中有静默型或信号型 NaN 时总是会返回*False* (即使是执行 Decimal('NaN')==Decimal('NaN'))，而不等性检测总是会返回*True*。当尝试使用 <, <=, > 或 >= 运算符中的任何一个来比较两个 Decimal 值时，如果运算数中有 NaN 则将引发*InvalidOperation* 信号，如果此信号未被捕获则将返回*False*。请注意通用十进制算术规范并未规定直接比较行为；这些涉及 NaN 的比较规则来自于 IEEE 854 标准 (见第 5.7 节表 3)。要确保严格符合标准，请改用 compare() 和 compare-signal() 方法。

有符号零值可以由向下溢出的运算产生。它们保留符号是为了让运算结果能以更高的精度传递。由于它们的大小为零，正零和负零会被视为相等，且它们的符号具有信息。

在这两个不相同但却相等的有符号零之外，还存在几种零的不同表示形式，它们的精度不同但值也都相等。这需要一些时间来逐渐适应。对于习惯了标准浮点表示形式的眼睛来说，以下运算返回等于零的值并不是显

而易见的:

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000000026')
```

9.4.6 使用线程

The *getcontext()* function accesses a different *Context* object for each thread. Having separate thread contexts means that threads may make changes (such as `getcontext.prec=10`) without interfering with other threads.

类似的*setcontext()* 会为当前上下文的目标自动赋值。

如果在调用*setcontext()* 之前调用了*getcontext()*，则*getcontext()* 将自动创建一个新的上下文在当前线程中使用。

新的上下文拷贝自一个名为 *DefaultContext* 的原型上下文。要控制默认值以便每个线程在应用运行期间都使用相同的值，可以直接修改 *DefaultContext* 对象。这应当在任何线程启动 之前完成以使得调用*getcontext()* 的线程之间不会产生竞争条件。例如:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
 . . .
```

9.4.7 例程

以下是一些用作工具函数的例程，它们演示了使用*Decimal* 类的各种方式:

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.

    places:  required number of places after the decimal point
    curr:    optional currency symbol before the sign (may be blank)
    sep:     optional grouping separator (comma, period, space, or blank)
    dp:      decimal point indicator (comma or period)
             only specify as blank when places is zero
    pos:     optional sign for positive numbers: '+', space or blank
    neg:     optional sign for negative numbers: '-', '(', space or blank
    trailneg:optional trailing minus indicator:  '-', ')', space or blank

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
```

(下页继续)

(续上页)

```
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<0.02>'

    """
    q = Decimal(10) ** -places      # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = map(str, digits)
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')
    build(dp)
    if not digits:
        build('0')
    i = 0
    while digits:
        build(next())
        i += 1
        if i == 3 and digits:
            i = 0
            build(sep)
    build(curr)
    build(neg if sign else pos)
    return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print pi()
    3.141592653589793238462643383

    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s               # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x.  Result type matches input type.

    >>> print exp(Decimal(1))
    2.718281828459045235360287471
    >>> print exp(Decimal(2))
    7.389056098930650227230427461
```

(下页继续)

(续上页)

```
    >>> print exp(2.0)
    7.38905609893
    >>> print exp(2+0j)
    (7.38905609893+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    >>> print cos(Decimal('0.5'))
    0.87758256189037271611628158260
    >>> print cos(0.5)
    0.87758256189
    >>> print cos(0.5+0j)
    (0.87758256189+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    >>> print sin(Decimal('0.5'))
    0.47942553860420300027328793521
    >>> print sin(0.5)
    0.479425538604
    >>> print sin(0.5+0j)
    (0.479425538604+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
        i += 2
```

(下页继续)

(续上页)

```
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
```

9.4.8 Decimal FAQ

Q. 总是输入 `decimal.Decimal('1234.5')` 是否过于笨拙。在使用交互解释器时有没有最小化输入量的方式?

A. 有些用户会将构造器简写为一个字母:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. 在带有两个十进制位的定点数应用中，有些输入值具有许多位，需要被舍入。另一些数则不应具有多余位，需要验证有效性。这种情况应该用什么方法?

A. 用 `quantize()` 方法舍入到固定数量的十进制位。如果设置了 *Inexact* 陷阱，它也适用于验证有效性:

```
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal('0.01')
```

```
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
```

```
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
   ...
Inexact: None
```

Q. 当我使用两个有效位的输入时，我要如何在一个应用中保持有效位不变?

A. 某些运算例如与整数相加、相减和相乘将会自动保留固定的小数位数。其他运算，例如相除和非整数相乘则将会改变小数位数，需要再加上 `quantize()` 处理步骤:

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                          # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)     # Must quantize non-integer multiplication
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)     # And quantize division
Decimal('0.03')
```

在开发定点数应用时，更方便的做法是定义处理 quantize() 步骤的函数：

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```
>>> mul(a, b)                           # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q. 表示同一个值有许多方式。数字 200, 200.000, 2E2 和 02E+4 的值都相同但有精度不同。是否有办法将它们转换为一个可识别的规范值?

A. normalize() 方法可将所有相同的值映射为统一表示形式：

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. 有些十进制值总是被打印为指数表示形式。是否有办法得到一个非指数表示形式?

A. 对于某些值来说，指数表示形式是表示系数中有效位的唯一办法。例如，将 5.0E+3 表示为 5000 可以让值保持恒定，但是无法显示原本的两位有效数字。

If an application does not care about tracking significance, it is easy to remove the exponent and trailing zeros, losing significance, but keeping the value unchanged:

```
def remove_exponent(d):
    '''Remove exponent and trailing zeros.

    >>> remove_exponent(Decimal('5E+3'))
    Decimal('5000')

    '''
    return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

Q. 是否有办法将一个普通浮点数转换为*Decimal*?

A. 是的，任何二进制浮点数都可以精确地表示为 Decimal 值，但精确的转换可能需要比直觉设想更高的精度：

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. 在一个复杂的计算中，我怎样才能保证不会得到由精度不足和舍入异常所导致的虚假结果。

A. 使用 decimal 模块可以很容易地检测结果。最好的做法是使用更高的精度和不同的舍入模式重新进行计算。明显不同的结果表明存在精度不足、舍入模式问题、不符合条件的输入或是结果不稳定的算法。

Q. 我发现上下文精度的应用只针对运算结果而不针对输入。在混合使用不同精度的值时有什么需要注意的吗?

A. 是的。原则上所有值都会被视为精确值，在这些值上进行的算术运算也是如此。只有结果会被舍入。对于输入来说其好处是“所输入即所得”。而其缺点则是如果你忘记了输入没有被舍入，结果看起来可能会很奇怪：

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
```

(下页继续)

(续上页)

```
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

解决办法是提高精度或使用单目加法运算对输入执行强制舍入：

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')      # unary plus triggers rounding
Decimal('1.23')
```

此外，还可以使用*Context.create_decimal()* 方法在创建输入时执行舍入：

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

9.5 fractions —分数

2.6 新版功能.

源代码 Lib/fractions.py

---

fractions 模块支持分数运算。

分数实例可以由一对整数，一个分数，或者一个字符串构建而成。

class fractions.**Fraction**(*numerator=0*, *denominator=1*)
class fractions.**Fraction**(*other_fraction*)
class fractions.**Fraction**(*float*)
class fractions.**Fraction**(*decimal*)
class fractions.**Fraction**(*string*)

第一个版本要求 *numerator* 和 *denominator* 是*numbers.Rational* 的实例，并返回一个新的*Fraction* 实例，其值为 numerator/denominator。如果 *denominator* 为 0 将会引发*ZeroDivisionError*。第二个版本要求 *other_fraction* 是*numbers.Rational* 的实例，并返回一个*Fraction* 实例且与传入值相等。下两个版本接受*float* 或*decimal.Decimal* 的实例，并返回一个*Fraction* 实例且与传入值完全相等。请注意由于二进制浮点数通常存在的问题 (参见 tut-fp-issues)，Fraction(1.1) 的参数并不会精确等于 11/10，因此 Fraction(1.1) 也 不会返回用户所期望的 Fraction(11, 10)。(请参阅下文中*limit_denominator()* 方法的文档。) 构造器的最后一个版本接受一个字符串或 unicode 实例。此实例的通常形式为:

```
[sign] numerator ['/' denominator]
```

其中的可选项 sign 可以为‘+’或‘-‘并且 numerator 和 denominator (如果存在) 是十进制数码的字符串。此外，*float* 构造器所接受的任何表示一个有限值的字符串也都为*Fraction* 构造器所接受。不论哪种形式的输入字符串也都可以带有前缀和/或后缀的空格符。这里是一些示例:

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
```

(下页继续)

(续上页)

```
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

The *Fraction* class inherits from the abstract base class *numbers.Rational*, and implements all of the methods and operations from that class. *Fraction* instances are hashable, and should be treated as immutable. In addition, *Fraction* has the following methods:

在 2.7 版更改: *Fraction* 构造器现在接受*float* 和*decimal.Decimal* 实例。

from_float(*flt*)

此类方法可构造一个*Fraction* 来表示 *flt* 的精确值，该参数必须是一个*float*。请注意 `Fraction.from_float(0.3)` 的值并不等于 `Fraction(3, 10)`。

注解： From Python 2.7 onwards, you can also construct a *Fraction* instance directly from a *float*.

---

from_decimal(*dec*)

This class method constructs a *Fraction* representing the exact value of *dec*, which must be a *decimal.Decimal*.

注解： From Python 2.7 onwards, you can also construct a *Fraction* instance directly from a *decimal.Decimal* instance.

---

limit_denominator(*max_denominator=1000000*)

找到并返回一个*Fraction* 使得其值最接近 `self` 并且分母不大于 max_denominator。此方法适用于找出给定浮点数的有理数近似值:

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

或是用来恢复被表示为一个浮点数的有理数:

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

fractions.**gcd**(*a*, *b*)
返回整数 *a* 和 *b* 的最大公约数。如果 *a* 或 *b* 之一非零，则 gcd(a, b) 的绝对值是能同时整除 *a* 和 *b* 的最大整数。若 *b* 非零，则 gcd(a,b) 与 *b* 同号；否则返回值与 *a* 同号。gcd(0, 0) 返回 0。

参见:

***numbers* 模块** 构成数字塔的所有抽象基类。

9.6 random —生成伪随机数

源码: Lib/random.py

该模块实现了各种分布的伪随机数生成器。

For integers, uniform selection from a range. For sequences, uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

在实数轴上，有计算均匀、正态（高斯）、对数正态、负指数、伽马和贝塔分布的函数。为了生成角度分布，可以使用 von Mises 分布。

几乎所有模块函数都依赖于基本函数*random()*，它在半开放区间 [0.0,1.0) 内均匀生成随机浮点数。Python 使用 Mersenne Twister 作为核心生成器。它产生 53 位精度浮点数，周期为 2**19937-1，其在 C 中的底层实现既快又线程安全。Mersenne Twister 是现存最广泛测试的随机数发生器之一。但是，因为完全确定性，它不适用于所有目的，并且完全不适合加密目的。

The functions supplied by this module are actually bound methods of a hidden instance of the random.Random class. You can instantiate your own instances of Random to get generators that don't share state. This is especially useful for multi-threaded programs, creating a different instance of Random for each thread, and using the *jumpahead()* method to make it likely that the generated sequences seen by each thread don't overlap.

Class Random can also be subclassed if you want to use a different basic generator of your own devising: in that case, override the random(), seed(), getstate(), setstate() and jumpahead() methods. Optionally, a new generator can supply a getrandbits() method —this allows *randrange()* to produce selections over an arbitrarily large range.

2.4 新版功能: the *getrandbits()* method.

As an example of subclassing, the *random* module provides the *WichmannHill* class that implements an alternative generator in pure Python. The class provides a backward compatible way to reproduce results from earlier versions of Python, which used the Wichmann-Hill algorithm as the core generator. Note that this Wichmann-Hill generator can no longer be recommended: its period is too short by contemporary standards, and the sequence generated is known to fail some stringent randomness tests. See the references below for a recent variant that repairs these flaws.

在 2.3 版更改: MersenneTwister replaced Wichmann-Hill as the default generator.

random 模块还提供*SystemRandom* 类，它使用系统函数*os.urandom()* 从操作系统提供的源生成随机数。

警告: The pseudo-random generators of this module should not be used for security purposes. Use *os.urandom()* or *SystemRandom* if you require a cryptographically secure pseudo-random number generator.

Bookkeeping functions:

random.**seed**(*a=None*)
Initialize internal state of the random number generator.

None or no argument seeds from current time or from an operating system specific randomness source if available (see the *os.urandom()* function for details on availability).

If *a* is not None or an *int* or a *long*, then hash(a) is used instead. Note that the hash values for some types are nondeterministic when PYTHONHASHSEED is enabled.

在 2.4 版更改: formerly, operating system resources were not used.

random.**getstate**()
返回捕获生成器当前内部状态的对象。这个对象可以传递给*setstate()* 来恢复状态。

2.1 新版功能.

在 2.6 版更改: State values produced in Python 2.6 cannot be loaded into earlier versions.

random.**setstate**(*state*)
state 应该是从之前调用*getstate()* 获得的，并且*setstate()* 将生成器的内部状态恢复到*getstate()* 被调用时的状态。

2.1 新版功能.

random.**jumpahead**(*n*)
Change the internal state to one different from and likely far away from the current state. *n* is a non-negative integer which is used to scramble the current state vector. This is most useful in multi-threaded programs, in conjunction with multiple instances of the Random class: *setstate()* or *seed()* can be used to force all instances into the same internal state, and then *jumpahead()* can be used to force the instances' states far apart.

2.1 新版功能.

在 2.3 版更改: Instead of jumping to a specific state, *n* steps ahead, jumpahead(n) jumps to another state likely to be separated by many steps.

random.**getrandbits**(*k*)
Returns a python *long* int with *k* random bits. This method is supplied with the MersenneTwister generator and some other generators may also provide it as an optional part of the API. When available, *getrandbits()* enables *randrange()* to handle arbitrarily large ranges.

2.4 新版功能.

Functions for integers:

random.**randrange**(*stop*)
random.**randrange**(*start*, *stop*[, *step*])
从 range(start, stop, step) 返回一个随机选择的元素。这相当于 choice(range(start, stop, step))，但实际上并没有构建一个 range 对象。

1.5.2 新版功能.

random.**randint**(*a*, *b*)
Return a random integer *N* such that a <= N <= b.

Functions for sequences:

random.**choice**(*seq*)
从非空序列 *seq* 返回一个随机元素。如果 *seq* 为空，则引发*IndexError*。

random.**shuffle**(*x*[, *random*])
Shuffle the sequence *x* in place. The optional argument *random* is a 0-argument function returning a random float in [0.0, 1.0); by default, this is the function *random()*.

Note that for even rather small len(x), the total number of permutations of *x* is larger than the period of most random number generators; this implies that most permutations of a long sequence can never be generated.

random.**sample**(*population*, *k*)
Return a *k* length list of unique elements chosen from the population sequence. Used for random sampling without replacement.

2.3 新版功能.

返回包含来自总体的元素的新列表，同时保持原始总体不变。结果列表按选择顺序排列，因此所有子切片也将是有效的随机样本。这允许抽奖获奖者（样本）被划分为大奖和第二名获胜者（子切片）。

总体成员不必是*hashable* 或 unique 。如果总体包含重复，则每次出现都是样本中可能的选择。

To choose a sample from a range of integers, use an *xrange()* object as an argument. This is especially fast and space efficient for sampling from a large population: `sample(xrange(10000000), 60)`.

以下函数生成特定的实值分布。如常用数学实践中所使用的那样, 函数参数以分布方程中的相应变量命名; 大多数这些方程都可以在任何统计学教材中找到。

random.**random**()
返回 [0.0, 1.0) 范围内的下一个随机浮点数。

random.**uniform**(*a*, *b*)
返回一个随机浮点数 *N* ，当 `a <= b` 时 `a <= N <= b` ，当 `b < a` 时 `b <= N <= a` 。

取决于等式 `a + (b-a) * random()` 中的浮点舍入，终点 `b` 可以包括或不包括在该范围内。

random.**triangular**(*low*, *high*, *mode*)
返回一个随机浮点数 *N* ，使得 `low <= N <= high` 并在这些边界之间使用指定的 *mode* 。*low* 和 *high* 边界默认为零和一。*mode* 参数默认为边界之间的中点，给出对称分布。

2.6 新版功能.

random.**betavariate**(*alpha*, *beta*)
Beta 分布。参数的条件是 `alpha > 0` 和 `beta > 0`。返回值的范围介于 0 和 1 之间。

random.**expovariate**(*lambd*)
指数分布。*lambd* 是 1.0 除以所需的平均值，它应该是非零的。（该参数本应命名为 “lambda”，但这是 Python 中的保留字。）如果 *lambd* 为正，则返回值的范围为 0 到正无穷大；如果 *lambd* 为负，则返回值从负无穷大到 0。

random.**gammavariate**(*alpha*, *beta*)
Gamma 分布。（不是 gamma 函数!）参数的条件是 `alpha > 0` 和 `beta > 0`。

概率分布函数是:

```
            x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

random.**gauss**(*mu*, *sigma*)
高斯分布。*mu* 是平均值，*sigma* 是标准差。这比下面定义的*normalvariate()* 函数略快。

random.**lognormvariate**(*mu*, *sigma*)
对数正态分布。如果你采用这个分布的自然对数，你将得到一个正态分布，平均值为 *mu* 和标准差为 *sigma* 。*mu* 可以是任何值，*sigma* 必须大于零。

random.**normalvariate**(*mu*, *sigma*)
正态分布。*mu* 是平均值，*sigma* 是标准差。

random.**vonmisesvariate**(*mu*, *kappa*)
冯 · 米塞斯（von Mises）分布。*mu* 是平均角度，以弧度表示，介于 0 和 2*pi* 之间，*kappa* 是浓度参数，必须大于或等于零。如果 *kappa* 等于零，则该分布在 0 到 2*pi* 的范围内减小到均匀的随机角度。

random.**paretovariate**(*alpha*)
帕累托分布。*alpha* 是形状参数。

random.**weibullvariate**(*alpha*, *beta*)
威布尔分布。*alpha* 是比例参数，*beta* 是形状参数。

Alternative Generators:

class random.**WichmannHill**([*seed*])
Class that implements the Wichmann-Hill algorithm as the core generator. Has all of the same methods as Random plus the *whseed()* method described below. Because this class is implemented in pure Python, it is not threadsafe and may require locks between calls. The period of the generator is 6,953,607,871,644 which is small enough to require care that two independent random sequences do not overlap.

random.**whseed**([*x*])
This is obsolete, supplied for bit-level compatibility with versions of Python prior to 2.1. See *seed()* for details. *whseed()* does not guarantee that distinct integer arguments yield distinct internal states, and can yield no more than about 2**24 distinct internal states in all.

class random.**SystemRandom**([*seed*])
Class that uses the *os.urandom()* function for generating random numbers from sources provided by the operating system. Not available on all systems. Does not rely on software state and sequences are not reproducible. Accordingly, the *seed()* and *jumpahead()* methods have no effect and are ignored. The *getstate()* and *setstate()* methods raise *NotImplementedError* if called.

2.4 新版功能.

Examples of basic usage:

```
>>> random.random()         # Random float x, 0.0 <= x < 1.0
0.37444887175646646
>>> random.uniform(1, 10)   # Random float x, 1.0 <= x < 10.0
1.1800146073117523
>>> random.randint(1, 10)   # Integer from 1 to 10, endpoints included
7
>>> random.randrange(0, 101, 2)   # Even integer from 0 to 100
26
>>> random.choice('abcdefghij')   # Choose a random element
'c'

>>> items = [1, 2, 3, 4, 5, 6, 7]
>>> random.shuffle(items)
>>> items
[7, 3, 2, 5, 6, 4, 1]

>>> random.sample([1, 2, 3, 4, 5],  3)   # Choose 3 elements
[4, 1, 5]
```

参见:

M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3–30 1998.

Wichmann, B. A. & Hill, I. D., "Algorithm AS 183: An efficient and portable pseudo-random number generator", Applied Statistics 31 (1982) 188-190.

Complementary-Multiply-with-Carry recipe for a compatible alternative random number generator with a long period and comparatively simple update operations.

9.7 itertools —为高效循环而创建迭代器的函数

2.3 新版功能.

本模块实现一系列*iterator*，这些迭代器受到 APL，Haskell 和 SML 的启发。为了适用于 Python，它们都被重新写过。

本模块标准化了一个快速、高效利用内存的核心工具集，这些工具本身或组合都很有用。它们一起形成了“迭代器代数”，这使得在纯 Python 中有可能创建简洁又高效的专用工具。

For instance, SML provides a tabulation tool: `tabulate(f)` which produces a sequence `f(0), f(1), ...`. The same effect can be achieved in Python by combining *imap()* and *count()* to form `imap(f, count())`.

These tools and their built-in counterparts also work well with the high-speed functions in the *operator* module. For example, the multiplication operator can be mapped across two vectors to form an efficient dot-product: `sum(imap(operator.mul, vector1, vector2))`.

Infinite Iterators:

迭代器	实参	结果	示例
count()	start, [step]	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
cycle()	p	p0, p1, …plast, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
repeat()	elem [,n]	elem, elem, elem, …重复无限次或 n 次	`repeat(10, 3) --> 10 10 10`

根据最短输入序列长度停止的迭代器：

迭代器	实参	结果	示例
chain()	p, q, …	p0, p1, …plast, q0, q1, …	`chain('ABC', 'DEF') --> A B C D E F`
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), …	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`
dropwhile()	pred, seq	seq[n], seq[n+1], …从 pred 首次真值测试失败开始	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`
groupby()	iterable[, keyfunc]	sub-iterators grouped by value of keyfunc(v)	
ifilter()	pred, seq	elements of seq where pred(elem) is true	`ifilter(lambda x: x%2, range(10)) --> 1 3 5 7 9`
ifilterfalse()	pred, seq	seq 中 pred(x) 为假值的元素，x 是 seq 中的元素。	`ifilterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8`
islice()	seq, [start,] stop [, step]	seq[start:stop:step] 中的元素	`islice('ABCDEFG', 2, None) --> C D E F G`
imap()	func, p, q, …	func(p0, q0), func(p1, q1), …	`imap(pow, (2,3,10), (5,2,3)) --> 32 9 1000`
starmap()	func, seq	func(*seq[0]), func(*seq[1]), …	`starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000`
tee()	it, n	it1, it2, …itn 将一个迭代器拆分为 n 个迭代器	
takewhile()	pred, seq	seq[0], seq[1], …, 直到 pred 真值测试失败	`takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4`
izip()	p, q, …	(p[0], q[0]), (p[1], q[1]), …	`izip('ABCD', 'xy') --> Ax By`
izip_longest()	p, q, …	(p[0], q[0]), (p[1], q[1]), …	`izip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-`

Combinatoric generators:

迭代器	实参	结果
product()	p, q, …[repeat=1]	笛卡尔积，相当于嵌套的 for 循环
permutations()	p[, r]	长度 r 元组，所有可能的排列，无重复元素
combinations()	p, r	长度 r 元组，有序，无重复元素
combinations_with_replacement()	p, r	长度 r 元组，有序，元素可重复
`product('ABCD', repeat=2)`		`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`		`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`		`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`		`AA AB AC AD BB BC BD CC CD DD`

9.7.1 Itertool 函数

下列模块函数均创建并返回迭代器。有些迭代器不限制输出流长度，所以它们只应在能截断输出流的函数或循环中使用。

`itertools.`**`chain`**`(*iterables)`

创建一个迭代器，它首先返回第一个可迭代对象中所有元素，接着返回下一个可迭代对象中所有元素，直到耗尽所有可迭代对象中的元素。可将多个序列处理为单个序列。大致相当于：

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

`classmethod` `chain.`**`from_iterable`**`(iterable)`

构建类似*chain()* 迭代器的另一个选择。从一个单独的可迭代参数中得到链式输入，该参数是延迟计算的。大致相当于：

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

2.6 新版功能.

`itertools.`**`combinations`**`(iterable, r)`

返回由输入 *iterable* 中元素组成长度为 *r* 的子序列。

组合按照字典序返回。所以如果输入 *iterable* 是有序的，生成的组合元组也是有序的。

即使元素的值相同，不同位置的元素也被认为是不同的。如果元素各自不同，那么每个组合中没有重复元素。

大致相当于：

```
def combinations(iterable, r):
    # combinations('ABCD', 2) --> AB AC AD BC BD CD
```

(下页继续)

(续上页)

```
    # combinations(range(4), 3) --> 012 013 023 123
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = range(r)
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

combinations() 的代码可被改写为*permutations()* 过滤后的子序列，（相对于元素在输入中的位置）元素不是有序的。

```
def combinations(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in permutations(range(n), r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

当 `0 <= r <= n` 时，返回项的个数是 `n! / r! / (n-r)!`；当 `r > n` 时，返回项个数为 0。

2.6 新版功能.

itertools.**combinations_with_replacement**(*iterable*, *r*)

返回由输入 *iterable* 中元素组成的长度为 *r* 的子序列，允许每个元素可重复出现。

组合按照字典序返回。所以如果输入 *iterable* 是有序的，生成的组合元组也是有序的。

不同位置的元素是不同的，即使它们的值相同。因此如果输入中的元素都是不同的话，返回的组合中元素也都会不同。

大致相当于：

```
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) --> AA AB AC BB BC CC
    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

combinations_with_replacement() 的代码可被改写为 production() 过滤后的子序列，（相对于元素在输入中的位置）元素不是有序的。

```
def combinations_with_replacement(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in product(range(n), repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

当 n > 0 时，返回项个数为 (n+r-1)! / r! / (n-1)!.

2.7 新版功能.

itertools.**compress**(*data*, *selectors*)

创建一个迭代器，它返回 *data* 中经 *selectors* 真值测试为 True 的元素。迭代器在两者较短的长度处停止。大致相当于：

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
    return (d for d, s in izip(data, selectors) if s)
```

2.7 新版功能.

itertools.**count**(*start=0*, *step=1*)

Make an iterator that returns evenly spaced values starting with *n*. Often used as an argument to *imap()* to generate consecutive data points. Also, used with *izip()* to add sequence numbers. Equivalent to:

```
def count(start=0, step=1):
    # count(10) --> 10 11 12 13 14 ...
    # count(2.5, 0.5) -> 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

当对浮点数计数时，替换为乘法代码有时精度会更好，例如：(start + step * i for i in count()) 。

在 2.7 版更改: added *step* argument and allowed non-integer arguments.

itertools.**cycle**(*iterable*)

创建一个迭代器，返回 *iterable* 中所有元素并保存一个副本。当取完 *iterable* 中所有元素，返回副本中的所有元素。无限重复。大致相当于：

```
def cycle(iterable):
    # cycle('ABCD') --> A B C D A B C D A B C D ...
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
              yield element
```

注意，该函数可能需要相当大的辅助空间（取决于 *iterable* 的长度）。

itertools.**dropwhile**(*predicate*, *iterable*)

创建一个迭代器，如果 *predicate* 为 true，迭代器丢弃这些元素，然后返回其他元素。注意，迭代器在 *predicate* 首次为 false 之前不会产生任何输出，所以可能需要一定长度的启动时间。大致相当于：

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

itertools.**groupby**(*iterable*[, *key*])

创建一个迭代器，返回 *iterable* 中连续的键和组。*key* 是一个计算元素键值函数。如果未指定或为 None，*key* 缺省为恒等函数（identity function），返回元素不变。一般来说，*iterable* 需用同一个键值函数预先排序。

groupby() 操作类似于 Unix 中的 uniq。当每次 *key* 函数产生的键值改变时，迭代器会分组或生成一个新组（这就是为什么通常需要使用同一个键值函数先对数据进行排序）。这种行为与 SQL 的 GROUP BY 操作不同，SQL 的操作会忽略输入的顺序将相同键值的元素分在同组中。

返回的组本身也是一个迭代器，它与*groupby()* 共享底层的可迭代对象。因为源是共享的，当*groupby()* 对象向后迭代时，前一个组将消失。因此如果稍后还需要返回结果，可保存为列表：

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # Store group iterator as a list
    uniquekeys.append(k)
```

groupby() 大致相当于：

```
class groupby(object):
    # [k for k, g in groupby('AAAABBBCCDAABBB')] --> A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] --> AAAA BBB CC D
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
        self.it = iter(iterable)
        self.tgtkey = self.currkey = self.currvalue = object()
    def __iter__(self):
        return self
    def next(self):
        while self.currkey == self.tgtkey:
            self.currvalue = next(self.it)    # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
        self.tgtkey = self.currkey
        return (self.currkey, self._grouper(self.tgtkey))
    def _grouper(self, tgtkey):
        while self.currkey == tgtkey:
            yield self.currvalue
            self.currvalue = next(self.it)    # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
```

2.4 新版功能.

itertools.**ifilter**(*predicate*, *iterable*)

Make an iterator that filters elements from iterable returning only those for which the predicate is `True`. If *predicate* is `None`, return the items that are true. Roughly equivalent to:

```
def ifilter(predicate, iterable):
    # ifilter(lambda x: x%2, range(10)) --> 1 3 5 7 9
    if predicate is None:
        predicate = bool
    for x in iterable:
        if predicate(x):
            yield x
```

itertools.**ifilterfalse**(*predicate*, *iterable*)

创建一个迭代器，只返回 *iterable* 中 *predicate* 为 `False` 的元素。如果 *predicate* 是 `None`，返回真值测试为 false 的元素。大致相当于：

```
def ifilterfalse(predicate, iterable):
    # ifilterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.**imap**(*function*, **iterables*)

Make an iterator that computes the function using arguments from each of the iterables. If *function* is set to `None`, then *imap()* returns the arguments as a tuple. Like *map()* but stops when the shortest iterable is exhausted instead of filling in `None` for shorter iterables. The reason for the difference is that infinite iterator arguments are typically an error for *map()* (because the output is fully evaluated) but represent a common and useful way of supplying arguments to *imap()*. Roughly equivalent to:

```
def imap(function, *iterables):
    # imap(pow, (2,3,10), (5,2,3)) --> 32 9 1000
    iterables = map(iter, iterables)
    while True:
        args = [next(it) for it in iterables]
        if function is None:
            yield tuple(args)
        else:
            yield function(*args)
```

itertools.**islice**(*iterable*, *stop*)

itertools.**islice**(*iterable*, *start*, *stop*[, *step*])

创建一个迭代器，返回从 *iterable* 里选中的元素。如果 *start* 不是 0，跳过 *iterable* 中的元素，直到到达 *start* 这个位置。之后迭代器连续返回元素，除非 *step* 设置的值很高导致被跳过。如果 *stop* 为 `None`，迭代器耗光为止；否则，在指定的位置停止。与普通的切片不同，*islice()* 不支持将 *start*，*stop*，或 *step* 设为负值。可用来从内部数据结构被压平的数据中提取相关字段（例如一个多行报告，它的名称字段出现在每三行上）。大致相当于：

```
def islice(iterable, *args):
    # islice('ABCDEFG', 2) --> A B
    # islice('ABCDEFG', 2, 4) --> C D
    # islice('ABCDEFG', 2, None) --> C D E F G
    # islice('ABCDEFG', 0, None, 2) --> A C E G
    s = slice(*args)
    start, stop, step = s.start or 0, s.stop or sys.maxint, s.step or 1
    it = iter(xrange(start, stop, step))
    try:
```

(下页继续)

(续上页)

```
        nexti = next(it)
    except StopIteration:
        # Consume *iterable* up to the *start* position.
        for i, element in izip(xrange(start), iterable):
            pass
        return
    try:
        for i, element in enumerate(iterable):
            if i == nexti:
                yield element
                nexti = next(it)
    except StopIteration:
        # Consume to *stop*.
        for i, element in izip(xrange(i + 1, stop), iterable):
            pass
```

如果 *start* 为 None，迭代从 0 开始。如果 *step* 为 None ，步长缺省为 1。

在 2.5 版更改: accept None values for default *start* and *step*.

itertools.**izip**(**iterables*)

Make an iterator that aggregates elements from each of the iterables. Like *zip()* except that it returns an iterator instead of a list. Used for lock-step iteration over several iterables at a time. Roughly equivalent to:

```
def izip(*iterables):
    # izip('ABCD', 'xy') --> Ax By
    iterators = map(iter, iterables)
    while iterators:
        yield tuple(map(next, iterators))
```

在 2.4 版更改: When no iterables are specified, returns a zero length iterator instead of raising a *TypeError* exception.

The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using izip(*[iter(s)]*n).

izip() should only be used with unequal length inputs when you don't care about trailing, unmatched values from the longer iterables. If those values are important, use *izip_longest()* instead.

itertools.**izip_longest**(**iterables*[, *fillvalue*])

创建一个迭代器，从每个可迭代对象中收集元素。如果可迭代对象的长度未对齐，将根据 *fillvalue* 填充缺失值。迭代持续到耗光最长的可迭代对象。大致相当于:

```
class ZipExhausted(Exception):
    pass

def izip_longest(*args, **kwds):
    # izip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
    fillvalue = kwds.get('fillvalue')
    counter = [len(args) - 1]
    def sentinel():
        if not counter[0]:
            raise ZipExhausted
        counter[0] -= 1
        yield fillvalue
    fillers = repeat(fillvalue)
    iterators = [chain(it, sentinel(), fillers) for it in args]
    try:
```

(下页继续)

(续上页)

```
        while iterators:
            yield tuple(map(next, iterators))
    except ZipExhausted:
        pass
```

If one of the iterables is potentially infinite, then the *izip_longest()* function should be wrapped with something that limits the number of calls (for example *islice()* or *takewhile()*). If not specified, *fillvalue* defaults to None.

2.6 新版功能.

itertools.**permutations**(*iterable*[, *r*])

连续返回由 *iterable* 元素生成长度为 *r* 的排列。

如果 *r* 未指定或为 None，*r* 默认设置为 *iterable* 的长度，这种情况下，生成所有全长排列。

排列依字典序发出。因此，如果 *iterable* 是已排序的，排列元组将有序地产出。

即使元素的值相同，不同位置的元素也被认为是不同的。如果元素值都不同，每个排列中的元素值不会重复。

大致相当于：

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) --> 012 021 102 120 201 210
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = range(n)
    cycles = range(n, n-r, -1)
    yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

permutations() 的代码也可被改写为*product()* 的子序列，只要将含有重复元素（来自输入中同一位置的）的项排除。

```
def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    for indices in product(range(n), repeat=r):
        if len(set(indices)) == r:
            yield tuple(pool[i] for i in indices)
```

当 0 <= r <= n，返回项个数为 n! / (n-r)!；当 r > n，返回项个数为 0。

2.6 新版功能.

itertools.**product**(**iterables*[, *repeat*])

可迭代对象输入的笛卡儿积。

大致相当于生成器表达式中的嵌套循环。例如，product(A, B) 和 ((x,y) for x in A for y in B) 返回结果一样。

嵌套循环像里程表那样循环变动，每次迭代时将最右侧的元素向后迭代。这种模式形成了一种字典序，因此如果输入的可迭代对象是已排序的，笛卡尔积元组依次序发出。

要计算可迭代对象自身的笛卡尔积，将可选参数 *repeat* 设定为要重复的次数。例如，product(A, repeat=4) 和 product(A, A, A, A) 是一样的。

该函数大致相当于下面的代码，只不过实际实现方案不会在内存中创建中间结果。

```
def product(*args, **kwds):
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) --> 000 001 010 011 100 101 110 111
    pools = map(tuple, args) * kwds.get('repeat', 1)
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

2.6 新版功能.

itertools.**repeat**(*object*[, *times*])

Make an iterator that returns *object* over and over again. Runs indefinitely unless the *times* argument is specified. Used as argument to *imap()* for invariant function parameters. Also used with *izip()* to create constant fields in a tuple record. Roughly equivalent to:

```
def repeat(object, times=None):
    # repeat(10, 3) --> 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in xrange(times):
            yield object
```

A common use for *repeat* is to supply a stream of constant values to *imap* or *zip*:

```
>>> list(imap(pow, xrange(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

itertools.**starmap**(*function*, *iterable*)

Make an iterator that computes the function using arguments obtained from the iterable. Used instead of *imap()* when argument parameters are already grouped in tuples from a single iterable (the data has been "pre-zipped"). The difference between *imap()* and *starmap()* parallels the distinction between function(a,b) and function(*c). Roughly equivalent to:

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

在 2.6 版更改: Previously, *starmap()* required the function arguments to be tuples. Now, any iterable is allowed.

itertools.**takewhile**(*predicate*, *iterable*)
创建一个迭代器，只要 predicate 为真就从可迭代对象中返回元素。大致相当于:

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

itertools.**tee**(*iterable*[, *n=2*])
Return *n* independent iterators from a single iterable. Roughly equivalent to:

```
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:             # when the local deque is empty
                newval = next(it)       # fetch a new value and
                for d in deques:        # load it to all the deques
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

一旦*tee()* 实施了一次分裂，原有的 *iterable* 不应再被使用；否则 tee 对象无法得知 *iterable* 可能已向后迭代。

tee 迭代器不是线程安全的。当同时使用由同一个*tee()* 调用所返回的迭代器时可能引发*RuntimeError*，即使原本的 *iterable* 是线程安全的。

该迭代工具可能需要相当大的辅助存储空间（这取决于要保存多少临时数据）。通常，如果一个迭代器在另一个迭代器开始之前就要使用大部份或全部数据，使用 `list()` 会比*tee()* 更快。

2.4 新版功能.

9.7.2 Recipes

本节将展示如何使用现有的 itertools 作为基础构件来创建扩展的工具集。

扩展的工具提供了与底层工具集相同的高性能。保持了超棒的内存利用率，因为一次只处理一个元素，而不是将整个可迭代对象加载到内存。代码量保持得很小，以函数式风格将这些工具连接在一起，有助于消除临时变量。速度依然很快，因为倾向于使用“矢量化”构件来取代解释器开销大的 for 循环和*generator* 。

```
def take(n, iterable):
    "Return first n items of the iterable as a list"
    return list(islice(iterable, n))

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return imap(function, count(start))

def consume(iterator, n=None):
```

(下页继续)

(续上页)

```
    "Advance the iterator n-steps ahead. If n is None, consume entirely."
    # Use functions that consume iterators at C speed.
    if n is None:
        # feed the entire iterator into a zero-length deque
        collections.deque(iterator, maxlen=0)
    else:
        # advance to the empty slice starting at position n
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value"
    return next(islice(iterable, n, None), default)

def all_equal(iterable):
    "Returns True if all the elements are equal to each other"
    g = groupby(iterable)
    return next(g, True) and not next(g, False)

def quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(imap(pred, iterable))

def padnone(iterable):
    """Returns the sequence elements and then returns None indefinitely.

    Useful for emulating the behavior of the built-in map() function.
    """
    return chain(iterable, repeat(None))

def ncycles(iterable, n):
    "Returns the sequence elements n times"
    return chain.from_iterable(repeat(tuple(iterable), n))

def dotproduct(vec1, vec2):
    return sum(imap(operator.mul, vec1, vec2))

def flatten(listOfLists):
    "Flatten one level of nesting"
    return chain.from_iterable(listOfLists)

def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.

    Example:  repeatfunc(random.random)
    """
    if times is None:
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return izip(a, b)

def grouper(iterable, n, fillvalue=None):
```

(下页继续)

(续上页)

```
    "Collect data into fixed-length chunks or blocks"
    # grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx
    args = [iter(iterable)] * n
    return izip_longest(fillvalue=fillvalue, *args)

def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    # Recipe credited to George Sakkis
    pending = len(iterables)
    nexts = cycle(iter(it).next for it in iterables)
    while pending:
        try:
            for next in nexts:
                yield next()
        except StopIteration:
            pending -= 1
            nexts = cycle(islice(nexts, pending))

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def unique_everseen(iterable, key=None):
    "List unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') --> A B C D
    # unique_everseen('ABBCcAD', str.lower) --> A B C D
    seen = set()
    seen_add = seen.add
    if key is None:
        for element in ifilterfalse(seen.__contains__, iterable):
            seen_add(element)
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen_add(k)
                yield element

def unique_justseen(iterable, key=None):
    "List unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') --> A B C D A B
    # unique_justseen('ABBCcAD', str.lower) --> A B C A D
    return imap(next, imap(itemgetter(1), groupby(iterable, key)))

def iter_except(func, exception, first=None):
    """ Call a function repeatedly until an exception is raised.

    Converts a call-until-exception interface to an iterator interface.
    Like __builtin__.iter(func, sentinel) but uses an exception instead
    of a sentinel to end the loop.

    Examples:
        bsddbiter = iter_except(db.next, bsddb.error, db.first)
        heapiter = iter_except(functools.partial(heappop, h), IndexError)
```

(下页继续)

(续上页)

```
        dictiter = iter_except(d.popitem, KeyError)
        dequeiter = iter_except(d.popleft, IndexError)
        queueiter = iter_except(q.get_nowait, Queue.Empty)
        setiter = iter_except(s.pop, KeyError)

    """
    try:
        if first is not None:
            yield first()
        while 1:
            yield func()
    except exception:
        pass

def random_product(*args, **kwds):
    "Random selection from itertools.product(*args, **kwds)"
    pools = map(tuple, args) * kwds.get('repeat', 1)
    return tuple(random.choice(pool) for pool in pools)

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(xrange(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "Random selection from itertools.combinations_with_replacement(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.randrange(n) for i in xrange(r))
    return tuple(pool[i] for i in indices)

def tee_lookahead(t, i):
    """Inspect the i-th upcomping value from a tee object
       while leaving the tee object at its current position.

       Raise an IndexError if the underlying iterator doesn't
       have enough values.

    """
    for value in islice(t.__copy__(), i, None):
        return value
    raise IndexError(i)
```

注意，通过将全局查找替换为局部变量的缺省值，上述配方中有很多可以这样优化。例如，*dotproduct* 配方可以这样写：

```
def dotproduct(vec1, vec2, sum=sum, imap=imap, mul=operator.mul):
    return sum(imap(mul, vec1, vec2))
```

9.8 functools —高阶函数和可调用对象上的操作

2.5 新版功能.

源代码: Lib/functools.py

---

functools 模块应用于高阶函数，即——参数或（和）返回值为其他函数的函数。通常来说，此模块的功能适用于所有可调用对象。

functools 模块定义了以下函数:

functools.**cmp_to_key**(*func*)

Transform an old-style comparison function to a *key function*. Used with tools that accept key functions (such as *sorted()*, *min()*, *max()*, *heapq.nlargest()*, *heapq.nsmallest()*, *itertools.groupby()*). This function is primarily used as a transition tool for programs being converted to Python 3 where comparison functions are no longer supported.

比较函数意为一个可调用对象，该对象接受两个参数并比较它们，结果为小于则返回一个负数，相等则返回零，大于则返回一个正数。key function 则是一个接受一个参数，并返回另一个用以排序的值的可调用对象。

示例:

```
sorted(iterable, key=cmp_to_key(locale.strcoll))  # locale-aware sort order
```

有关排序示例和简要排序教程，请参阅 sortinghowto 。

2.7 新版功能.

functools.**total_ordering**(*cls*)

给定一个声明一个或多个全比较排序方法的类，这个类装饰器实现剩余的方法。这减轻了指定所有可能的全比较操作的工作。

此类必须包含以下方法之一：__lt__() 、__le__()、__gt__() 或 __ge__()。另外，此类必须支持 __eq__() 方法。

例如

```
@total_ordering
class Student:
    def __eq__(self, other):
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

2.7 新版功能.

functools.**reduce**(*function*, *iterable*[, *initializer*])

This is the same function as *reduce()*. It is made available in this module to allow writing code more forward-compatible with Python 3.

2.6 新版功能.

functools.**partial**(*func[,*args][, **keywords]*)

返回一个新的部分对象，当被调用时其行为类似于 *func* 附带位置参数 *args* 和关键字参数 *keywords* 被调用。如果为调用提供了更多的参数，它们会被附加到 *args*。如果提供了额外的关键字参数，它们会扩展并重载 *keywords*。大致等价于:

```
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*(args + fargs), **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

partial() 会被“冻结了”一部分函数参数和/或关键字的部分函数应用所使用，从而得到一个具有简化签名的新对象。例如，*partial()* 可用来创建一个行为类似于*int()* 函数的可调用对象，其中 *base* 参数默认为二：

```
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

functools.**update_wrapper**(*wrapper, wrapped[, assigned][, updated]*)

Update a *wrapper* function to look like the *wrapped* function. The optional arguments are tuples to specify which attributes of the original function are assigned directly to the matching attributes on the wrapper function and which attributes of the wrapper function are updated with the corresponding attributes from the original function. The default values for these arguments are the module level constants *WRAPPER_ASSIGNMENTS* (which assigns to the wrapper function’s *__name__*, *__module__* and *__doc__*, the documentation string) and *WRAPPER_UPDATES* (which updates the wrapper function’s *__dict__*, i.e. the instance dictionary).

此函数的主要目的是在*decorator* 函数中用来包装被装饰的函数并返回包装器。如果包装器函数未被更新，则被返回函数的元数据将反映包装器定义而不是原始函数定义，这通常没有什么用处。

functools.**wraps**(*wrapped[, assigned][, updated]*)

这是一个便捷函数，用于在定义包装器函数时发起调用*update_wrapper()* 作为函数装饰器。它等价于 `partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)`。例如:

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print 'Calling decorated function'
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print 'Called example function'
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

如果不使用这个装饰器工厂函数，则 example 函数的名称将变为 `'wrapper'`，并且 `example()` 原本

的文档字符串将会丢失。

9.8.1 partial 对象

partial 对象是由*partial()* 创建的可调用对象。它们具有三个只读属性：

partial.**func**
一个可调用对象或函数。对*partial* 对象的调用将被转发给*func* 并附带新的参数和关键字。

partial.**args**
最左边的位置参数将放置在提供给*partial* 对象调用的位置参数之前。

partial.**keywords**
当调用*partial* 对象时将要提供的关键字参数。

partial 对象与 function 对象的类似之处在于它们都是可调用、可弱引用的对象并可拥有属性。但两者也存在一些重要的区别。例如前者不会自动创建*__name__* 和 __doc__ 属性。而且，在类中定义的*partial* 对象的行为类似于静态方法，并且不会在实例属性查找期间转换为绑定方法。

9.9 operator —标准运算符替代函数

The *operator* module exports a set of efficient functions corresponding to the intrinsic operators of Python. For example, `operator.add(x, y)` is equivalent to the expression `x+y`. The function names are those used for special class methods; variants without leading and trailing __ are also provided for convenience.

The functions fall into categories that perform object comparisons, logical operations, mathematical operations, sequence operations, and abstract type tests.

对象比较函数适用于所有的对象，函数名根据它们对应的比较运算符命名。

operator.**lt**(*a*, *b*)
operator.**le**(*a*, *b*)
operator.**eq**(*a*, *b*)
operator.**ne**(*a*, *b*)
operator.**ge**(*a*, *b*)
operator.**gt**(*a*, *b*)
operator.**__lt__**(*a*, *b*)
operator.**__le__**(*a*, *b*)
operator.**__eq__**(*a*, *b*)
operator.**__ne__**(*a*, *b*)
operator.**__ge__**(*a*, *b*)
operator.**__gt__**(*a*, *b*)
Perform "rich comparisons" between *a* and *b*. Specifically, `lt(a, b)` is equivalent to `a < b`, `le(a, b)` is equivalent to `a <= b`, `eq(a, b)` is equivalent to `a == b`, `ne(a, b)` is equivalent to `a != b`, `gt(a, b)` is equivalent to `a > b` and `ge(a, b)` is equivalent to `a >= b`. Note that unlike the built-in *cmp()*, these functions can return any value, which may or may not be interpretable as a Boolean value. See comparisons for more information about rich comparisons.

2.2 新版功能.

逻辑运算通常也适用于所有对象，并且支持真值检测、标识检测和布尔运算：

operator.**not_**(*obj*)
operator.**__not__**(*obj*)
Return the outcome of `not` *obj*. (Note that there is no *__not__()* method for object instances; only the interpreter core defines this operation. The result is affected by the __nonzero__() and __len__() methods.)

operator.**truth**(*obj*)
如果 *obj* 为真值则返回*True*，否则返回*False*。这等价于使用*bool* 构造器。

operator.**is_**(*a*, *b*)
返回 `a is b`. 测试对象标识。

2.3 新版功能.

operator.**is_not**(*a*, *b*)
返回 `a is not b`. 测试对象标识。

2.3 新版功能.

数学和按位运算的种类是最多的：

operator.**abs**(*obj*)
operator.**__abs__**(*obj*)
返回 *obj* 的绝对值。

operator.**add**(*a*, *b*)
operator.**__add__**(*a*, *b*)
对于数字 *a* 和 *b*，返回 `a + b`。

operator.**and_**(*a*, *b*)
operator.**__and__**(*a*, *b*)
返回 *x* 和 *y* 按位与

operator.**div**(*a*, *b*)
operator.**__div__**(*a*, *b*)
Return `a / b` when `__future__.division` is not in effect. This is also known as "classic" division.

operator.**floordiv**(*a*, *b*)
operator.**__floordiv__**(*a*, *b*)
返回 `a // b`.

2.2 新版功能.

operator.**index**(*a*)
operator.**__index__**(*a*)
返回 *a* 转换为整数的结果。等价于 `a.__index__()`。

2.5 新版功能.

operator.**inv**(*obj*)
operator.**invert**(*obj*)
operator.**__inv__**(*obj*)
operator.**__invert__**(*obj*)
返回数字 *obj* 按位取反的结果。这等价于 `~obj`。

2.0 新版功能: The names *invert()* and *__invert__()*.

operator.**lshift**(*a*, *b*)
operator.**__lshift__**(*a*, *b*)
返回 *a* 左移 *b* 位的结果。

operator.**mod**(*a*, *b*)
operator.**__mod__**(*a*, *b*)
返回 `a % b`.

operator.**mul**(*a*, *b*)
operator.**__mul__**(*a*, *b*)
对于数字 *a* 和 *b*，返回 `a * b`。

operator.**neg**(*obj*)
operator.**__neg__**(*obj*)
返回 *obj* 的负值 (-obj)。

operator.**or_**(*a*, *b*)
operator.**__or__**(*a*, *b*)
返回 *a* 和 *b* 按位或的结果。

operator.**pos**(*obj*)
operator.**__pos__**(*obj*)
返回 *obj* 取正的结果 (+obj)。

operator.**pow**(*a*, *b*)
operator.**__pow__**(*a*, *b*)
对于数字 *a* 和 *b*，返回 a ** b。

2.3 新版功能.

operator.**rshift**(*a*, *b*)
operator.**__rshift__**(*a*, *b*)
返回 *a* 右移 *b* 位的结果。

operator.**sub**(*a*, *b*)
operator.**__sub__**(*a*, *b*)
返回 a - b.

operator.**truediv**(*a*, *b*)
operator.**__truediv__**(*a*, *b*)
Return a / b when __future__.division is in effect. This is also known as "true" division.

2.2 新版功能.

operator.**xor**(*a*, *b*)
operator.**__xor__**(*a*, *b*)
返回 *a* 和 *b* 按位异或的结果。

适用于序列的操作（其中一些也适用于映射）包括:

operator.**concat**(*a*, *b*)
operator.**__concat__**(*a*, *b*)
对于序列 *a* 和 *b*，返回 a + b。

operator.**contains**(*a*, *b*)
operator.**__contains__**(*a*, *b*)
返回 b in a 检测的结果。请注意操作数是反序的。

2.0 新版功能: The name *__contains__()*.

operator.**countOf**(*a*, *b*)
返回 *b* 在 *a* 中的出现次数。

operator.**delitem**(*a*, *b*)
operator.**__delitem__**(*a*, *b*)
移除索引号 *b* 上的值 *a*。

operator.**delslice**(*a*, *b*, *c*)
operator.**__delslice__**(*a*, *b*, *c*)
Delete the slice of *a* from index *b* to index *c-1*.

2.6 版后已移除: This function is removed in Python 3.x. Use *delitem()* with a slice index.

operator.**getitem**(*a*, *b*)

operator.**__getitem__**(*a*, *b*)
返回索引号 *b* 上的值 *a*。

operator.**getslice**(*a*, *b*, *c*)
operator.**__getslice__**(*a*, *b*, *c*)
Return the slice of *a* from index *b* to index *c-1*.

2.6 版后已移除: This function is removed in Python 3.x. Use *getitem()* with a slice index.

operator.**indexOf**(*a*, *b*)
返回 *b* 在 *a* 中首次出现所在的索引号。

operator.**repeat**(*a*, *b*)
operator.**__repeat__**(*a*, *b*)
2.7 版后已移除: Use *__mul__()* instead.

Return `a * b` where *a* is a sequence and *b* is an integer.

operator.**sequenceIncludes**(...)
2.0 版后已移除: Use *contains()* instead.

Alias for *contains()*.

operator.**setitem**(*a*, *b*, *c*)
operator.**__setitem__**(*a*, *b*, *c*)
将索引号 *b* 上的值 *a* 设为 *c*。

operator.**setslice**(*a*, *b*, *c*, *v*)
operator.**__setslice__**(*a*, *b*, *c*, *v*)
Set the slice of *a* from index *b* to index *c-1* to the sequence *v*.

2.6 版后已移除: This function is removed in Python 3.x. Use *setitem()* with a slice index.

Example use of operator functions:

```
>>> # Elementwise multiplication
>>> map(mul, [0, 1, 2, 3], [10, 20, 30, 40])
[0, 20, 60, 120]

>>> # Dot product
>>> sum(map(mul, [0, 1, 2, 3], [10, 20, 30, 40]))
200
```

Many operations have an "in-place" version. The following functions provide a more primitive access to in-place operators than the usual syntax does; for example, the *statement* `x += y` is equivalent to `x = operator.iadd(x, y)`. Another way to put it is to say that `z = operator.iadd(x, y)` is equivalent to the compound statement `z = x; z += y`.

operator.**iadd**(*a*, *b*)
operator.**__iadd__**(*a*, *b*)
`a = iadd(a, b)` 等价于 `a += b`。

2.5 新版功能.

operator.**iand**(*a*, *b*)
operator.**__iand__**(*a*, *b*)
`a = iand(a, b)` 等价于 `a &= b`。

2.5 新版功能.

operator.**iconcat**(*a*, *b*)

operator.**__iconcat__**(*a*, *b*)
a = iconcat(a, b) 等价于 a += b 其中 *a* 和 *b* 为序列。

2.5 新版功能.

operator.**idiv**(*a*, *b*)
operator.**__idiv__**(*a*, *b*)
a = idiv(a, b) is equivalent to a /= b when __future__.division is not in effect.

2.5 新版功能.

operator.**ifloordiv**(*a*, *b*)
operator.**__ifloordiv__**(*a*, *b*)
a = ifloordiv(a, b) 等价于 a //= b.

2.5 新版功能.

operator.**ilshift**(*a*, *b*)
operator.**__ilshift__**(*a*, *b*)
a = ilshift(a, b) 等价于 a <<= b。

2.5 新版功能.

operator.**imod**(*a*, *b*)
operator.**__imod__**(*a*, *b*)
a = imod(a, b) 等价于 a %= b。

2.5 新版功能.

operator.**imul**(*a*, *b*)
operator.**__imul__**(*a*, *b*)
a = imul(a, b) 等价于 a *= b。

2.5 新版功能.

operator.**ior**(*a*, *b*)
operator.**__ior__**(*a*, *b*)
a = ior(a, b) 等价于 a |= b。

2.5 新版功能.

operator.**ipow**(*a*, *b*)
operator.**__ipow__**(*a*, *b*)
a = ipow(a, b) 等价于 a **= b。

2.5 新版功能.

operator.**irepeat**(*a*, *b*)
operator.**__irepeat__**(*a*, *b*)
2.7 版后已移除: Use *__imul__()* instead.

a = irepeat(a, b) is equivalent to a *= b where *a* is a sequence and *b* is an integer.

2.5 新版功能.

operator.**irshift**(*a*, *b*)
operator.**__irshift__**(*a*, *b*)
a = irshift(a, b) 等价于 a >>= b。

2.5 新版功能.

operator.**isub**(*a*, *b*)
operator.**__isub__**(*a*, *b*)
a = isub(a, b) 等价于 a -= b。

2.5 新版功能.

operator.**itruediv**(*a*, *b*)
operator.**__itruediv__**(*a*, *b*)
`a = itruediv(a, b)` is equivalent to `a /= b` when `__future__.division` is in effect.

2.5 新版功能.

operator.**ixor**(*a*, *b*)
operator.**__ixor__**(*a*, *b*)
`a = ixor(a, b)` 等价于 `a ^= b`。

2.5 新版功能.

The *operator* module also defines a few predicates to test the type of objects; however, these are not all reliable. It is preferable to test abstract base classes instead (see *collections* and *numbers* for details).

operator.**isCallable**(*obj*)
2.0 版后已移除: Use `isinstance(x, collections.Callable)` instead.

Returns true if the object *obj* can be called like a function, otherwise it returns false. True is returned for functions, bound and unbound methods, class objects, and instance objects which support the `__call__()` method.

operator.**isMappingType**(*obj*)
2.7 版后已移除: Use `isinstance(x, collections.Mapping)` instead.

Returns true if the object *obj* supports the mapping interface. This is true for dictionaries and all instance objects defining *__getitem__()*.

operator.**isNumberType**(*obj*)
2.7 版后已移除: Use `isinstance(x, numbers.Number)` instead.

Returns true if the object *obj* represents a number. This is true for all numeric types implemented in C.

operator.**isSequenceType**(*obj*)
2.7 版后已移除: Use `isinstance(x, collections.Sequence)` instead.

Returns true if the object *obj* supports the sequence protocol. This returns true for all objects which define sequence methods in C, and for all instance objects defining *__getitem__()*.

operator 模块还定义了一些用于常规属性和条目查找的工具。这些工具适合用来编写快速字段提取器作为*map()*, *sorted()*, *itertools.groupby()* 或其他需要相应函数参数的函数的参数。

operator.**attrgetter**(*attr*)
operator.**attrgetter**(**attrs*)
返回一个可从操作数中获取 *attr* 的可调用对象。如果请求了一个以上的属性，则返回一个属性元组。属性名称还可包含点号。例如:

- 在 `f = attrgetter('name')` 之后，调用 `f(b)` 将返回 `b.name`。
- 在 `f = attrgetter('name', 'date')` 之后，调用 `f(b)` 将返回 `(b.name, b.date)`。
- 在 `f = attrgetter('name.first', 'name.last')` 之后，调用 `f(b)` 将返回 `(b.name.first, b.name.last)`。

等价于:

```
def attrgetter(*items):
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
```

(下页继续)

(续上页)

```
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g

def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

2.4 新版功能.

在 2.5 版更改: Added support for multiple attributes.

在 2.6 版更改: Added support for dotted attributes.

operator.**itemgetter**(*item*)
operator.**itemgetter**(**items*)

返回一个使用操作数的*__getitem__()* 方法从操作数中获取 *item* 的可调用对象。如果指定了多个条目，则返回一个查找值的元组。例如:

- 在 `f = itemgetter(2)` 之后，调用 `f(r)` 将返回 `r[2]`。
- 在 `g = itemgetter(2, 5, 3)` 之后，调用 `g(r)` 将返回 `(r[2], r[5], r[3])`。

等价于:

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

传入的条目可以为操作数的*__getitem__()* 所接受的任何类型。字典接受任意可哈希的值。列表、元组和字符串接受 index 或 slice 对象:

```
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1,3,5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2,None))('ABCDEFG')
'CDEFG'
```

2.4 新版功能.

在 2.5 版更改: Added support for multiple item extraction.

使用*itemgetter()* 从元组的记录中提取特定字段的例子:

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> map(getcount, inventory)
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator.**methodcaller**(*name*[, *args...*])

返回一个在操作数上调用 *name* 方法的可调用对象。如果给出额外的参数和/或关键字参数，它们也将被传给该方法。例如:

- 在 `f = methodcaller('name')` 之后，调用 `f(b)` 将返回 `b.name()`。
- 在 `f = methodcaller('name', 'foo', bar=1)` 之后，调用 `f(b)` 将返回 `b.name('foo', bar=1)`。

等价于:

```
def methodcaller(name, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

2.6 新版功能.

9.9.1 将运算符映射到函数

以下表格显示了抽象运算是如何对应于 Python 语法中的运算符和*operator* 模块中的函数的。

运算	语法	函数
加法	`a + b`	`add(a, b)`
字符串拼接	`seq1 + seq2`	`concat(seq1, seq2)`
包含测试	`obj in seq`	`contains(seq, obj)`
除法	`a / b`	`div(a, b)` (without `__future__.division`)
除法	`a / b`	`truediv(a, b)` (with `__future__.division`)
除法	`a // b`	`floordiv(a, b)`
按位与	`a & b`	`and_(a, b)`
按位异或	`a ^ b`	`xor(a, b)`
按位取反	`~ a`	`invert(a)`
按位或	`a | b`	`or_(a, b)`
取幂	`a ** b`	`pow(a, b)`
一致	`a is b`	`is_(a, b)`
一致	`a is not b`	`is_not(a, b)`
索引赋值	`obj[k] = v`	`setitem(obj, k, v)`
索引删除	`del obj[k]`	`delitem(obj, k)`
索引取值	`obj[k]`	`getitem(obj, k)`
左移	`a << b`	`lshift(a, b)`
取模	`a % b`	`mod(a, b)`
乘法	`a * b`	`mul(a, b)`
否定（算术）	`- a`	`neg(a)`
否定（逻辑）	`not a`	`not_(a)`
正数	`+ a`	`pos(a)`
右移	`a >> b`	`rshift(a, b)`
Sequence Repetition	`seq * i`	`repeat(seq, i)`
切片赋值	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
切片删除	`del seq[i:j]`	`delitem(seq, slice(i, j))`
切片取值	`seq[i:j]`	`getitem(seq, slice(i, j))`
字符串格式化	`s % obj`	`mod(s, obj)`
减法	`a - b`	`sub(a, b)`
真值测试	`obj`	`truth(obj)`

下页继续

表 1 – 续上页

运算	语法	函数
比较	`a < b`	`lt(a, b)`
比较	`a <= b`	`le(a, b)`
相等	`a == b`	`eq(a, b)`
不等	`a != b`	`ne(a, b)`
比较	`a >= b`	`ge(a, b)`
比较	`a > b`	`gt(a, b)`

CHAPTER 10

文件和目录访问

本章中描述的模块处理磁盘文件和目录。例如，有一些模块用于读取文件的属性，以可移植的方式操作路径以及创建临时文件。本章的完整模块列表如下：

10.1 os.path —常见路径操作

This module implements some useful functions on pathnames. To read or write files see *open()*, and for accessing the filesystem see the *os* module.

注解： On Windows, many of these functions do not properly support UNC pathnames. *splitunc()* and *ismount()* do handle them correctly.

与 unix shell 不同，Python 不执行任何 自动路径扩展。当应用程序需要类似 shell 的路径扩展时，可以显式调用诸如*expanduser()* 和*expandvars()* 之类的函数。（另请参见*glob* 模块。）

注解： 由于不同的操作系统具有不同的路径名称约定，因此标准库中有此模块的几个版本。*os.path* 模块始终是适合 Python 运行的操作系统的路径模块，因此可用于本地路径。但是，如果操作的路径 总是以一种不同的格式显示，那么也可以分别导入和使用各个模块。它们都具有相同的界面：

- `posixpath` 用于 Unix 样式的路径
- `ntpath` 用于 Windows 路径
- *macpath* 用于旧 MacOS 样式的路径
- `os2emxpath` for OS/2 EMX paths

`os.path.`**`abspath`**(*path*)
返回路径 *path* 的绝对路径（标准化的）。在大多数平台上，这等同于用 `normpath(join(os.getcwd(), path))` 的方式调用*normpath()* 函数。

1.5.2 新版功能.

os.path.**basename**(*path*)
返回路径 *path* 的基本名称。这是将 *path* 传入函数*split()* 之后，返回的一对值中的第二个元素。请注意，此函数的结果与 Unix **basename** 程序不同。**basename** 在 '/foo/bar/' 上返回 'bar'，而*basename()* 函数返回一个空字符串 ('')。

os.path.**commonprefix**(*list*)
Return the longest path prefix (taken character-by-character) that is a prefix of all paths in *list*. If *list* is empty, return the empty string (''). Note that this may return invalid paths because it works a character at a time.

os.path.**dirname**(*path*)
返回路径 *path* 的目录名称。这是将 *path* 传入函数*split()* 之后，返回的一对值中的第一个元素。

os.path.**exists**(*path*)
Return True if *path* refers to an existing path. Returns False for broken symbolic links. On some platforms, this function may return False if permission is not granted to execute *os.stat()* on the requested file, even if the *path* physically exists.

os.path.**lexists**(*path*)
如果 *path* 指向一个已存在的路径，返回 True。对于失效的符号链接，返回 False。在缺失*os.lstat()* 的平台上等同于*exists()*。

2.4 新版功能.

os.path.**expanduser**(*path*)
在 Unix 和 Windows 上，将参数中开头部分的 ~ 或 ~user 替换为当前 用户 的家目录并返回。

在 Unix 上，开头的 ~ 会被环境变量 HOME 代替，如果变量未设置，则通过内置模块*pwd* 在 password 目录中查找当前用户的主目录。以 ~user 开头则直接在 password 目录中查找。

在 Windows 上，如果设置了 HOME 和 USERPROFILE 则将使用它们，否则将使用 HOMEPATH 和 HOMEDRIVE 的组合。原本的 ~user 处理方式为从上述方法所生成的用户路径中截去最后一级目录。

如果展开路径失败，或者路径不是以波浪号开头，则路径将保持不变。

os.path.**expandvars**(*path*)
输入带有环境变量的路径作为参数，返回展开变量以后的路径。$name 或 ${name} 形式的子字符串被环境变量 *name* 的值替换。格式错误的变量名称和对不存在变量的引用保持不变。

在 Windows 上，除了 $name 和 ${name} 外，还可以展开 %name%。

os.path.**getatime**(*path*)
Return the time of last access of *path*. The return value is a number giving the number of seconds since the epoch (see the *time* module). Raise *os.error* if the file does not exist or is inaccessible.

1.5.2 新版功能.

在 2.3 版更改: If *os.stat_float_times()* returns True, the result is a floating point number.

os.path.**getmtime**(*path*)
Return the time of last modification of *path*. The return value is a number giving the number of seconds since the epoch (see the *time* module). Raise *os.error* if the file does not exist or is inaccessible.

1.5.2 新版功能.

在 2.3 版更改: If *os.stat_float_times()* returns True, the result is a floating point number.

os.path.**getctime**(*path*)
Return the system's ctime which, on some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time for *path*. The return value is a number giving the number of seconds since the epoch (see the *time* module). Raise *os.error* if the file does not exist or is inaccessible.

2.3 新版功能.

os.path.**getsize**(*path*)
Return the size, in bytes, of *path*. Raise *os.error* if the file does not exist or is inaccessible.

1.5.2 新版功能.

os.path.**isabs**(*path*)
如果 *path* 是一个绝对路径，则返回 True。在 Unix 上，它就是以斜杠开头，而在 Windows 上，它可以是去掉驱动器号后以斜杠（或反斜杠）开头。

os.path.**isfile**(*path*)
Return True if *path* is an existing regular file. This follows symbolic links, so both *islink()* and *isfile()* can be true for the same path.

os.path.**isdir**(*path*)
Return True if *path* is an existing directory. This follows symbolic links, so both *islink()* and *isdir()* can be true for the same path.

os.path.**islink**(*path*)
Return True if *path* refers to a directory entry that is a symbolic link. Always False if symbolic links are not supported by the Python runtime.

os.path.**ismount**(*path*)
Return True if pathname *path* is a *mount point*: a point in a file system where a different file system has been mounted. The function checks whether *path*'s parent, path/.., is on a different device than *path*, or whether path/.. and *path* point to the same i-node on the same device —this should detect mount points for all Unix and POSIX variants.

os.path.**join**(*path*, **paths*)
合理地拼接一个或多个路径部分。返回值是 *path* 和 **paths* 所有值的连接，每个非空部分后面都紧跟一个目录分隔符 (os.sep)，除了最后一部分。这意味着如果最后一部分为空，则结果将以分隔符结尾。如果参数中某个部分是绝对路径，则绝对路径前的路径都将被丢弃，并从绝对路径部分开始连接。

在 Windows 上，遇到绝对路径部分（例如 r'\foo'）时，不会重置盘符。如果某部分路径包含盘符，则会丢弃所有先前的部分，并重置盘符。请注意，由于每个驱动器都有一个“当前目录”，所以 os.path.join("c:", "foo") 表示驱动器 C: 上当前目录的相对路径 (c:foo)，而不是 c:\foo。

os.path.**normcase**(*path*)
Normalize the case of a pathname. On Unix and Mac OS X, this returns the path unchanged; on case-insensitive filesystems, it converts the path to lowercase. On Windows, it also converts forward slashes to backward slashes.

os.path.**normpath**(*path*)
通过折叠多余的分隔符和对上级目录的引用来标准化路径名，所以 A//B、A/B/、A/./B 和 A/foo/../B 都会转换成 A/B。这个字符串操作可能会改变带有符号链接的路径的含义。在 Windows 上，本方法将正斜杠转换为反斜杠。要规范大小写，请使用*normcase()*。

os.path.**realpath**(*path*)
返回指定文件的规范路径，消除路径中存在的任何符号链接（如果操作系统支持）。

2.2 新版功能.

os.path.**relpath**(*path*[, *start*])
返回从当前目录或 *start* 目录（可选）到达 *path* 之间要经过的相对路径。这仅仅是对路径的计算，不会访问文件系统来确认 *path* 或 *start* 的存在性或属性。

start 默认为 *os.curdir*

Availability: Windows, Unix.

2.6 新版功能.

os.path.**samefile**(*path1*, *path2*)
Return True if both pathname arguments refer to the same file or directory (as indicated by device number and i-node number). Raise an exception if an *os.stat()* call on either pathname fails.

Availability: Unix.

os.path.**sameopenfile**(*fp1*, *fp2*)
如果文件描述符 *fp1* 和 *fp2* 指向相同文件，则返回 True。

Availability: Unix.

os.path.**samestat**(*stat1*, *stat2*)
如果 stat 元组 *stat1* 和 *stat2* 指向相同文件，则返回 True。这些 stat 元组可能是由*os.fstat()*、*os.lstat()* 或*os.stat()* 返回的。本函数实现了*samefile()* 和*sameopenfile()* 底层所使用的比较过程。

Availability: Unix.

os.path.**split**(*path*)
将路径 *path* 拆分为一对，即 (head, tail)，其中，*tail* 是路径的最后一部分，而 *head* 里是除最后部分外的所有内容。*tail* 部分不会包含斜杠，如果 *path* 以斜杠结尾，则 *tail* 将为空。如果 *path* 中没有斜杠，*head* 将为空。如果 *path* 为空，则 *head* 和 *tail* 均为空。*head* 末尾的斜杠会被去掉，除非它是根目录（即它仅包含一个或多个斜杠）。在所有情况下，join(head, tail) 指向的位置都与 *path* 相同（但字符串可能不同）。另请参见函数*dirname()* 和*basename()*。

os.path.**splitdrive**(*path*)
Split the pathname *path* into a pair (drive, tail) where *drive* is either a drive specification or the empty string. On systems which do not use drive specifications, *drive* will always be the empty string. In all cases, drive + tail will be the same as *path*.

1.3 新版功能.

os.path.**splitext**(*path*)
将路径 *path* 拆分为一对，即 (root, ext)，使 root + ext == path，其中 *ext* 为空或以英文句点开头，且最多包含一个句点。路径前的句点将被忽略，例如 splitext('.cshrc') 返回 ('.cshrc', '')。

在 2.6 版更改: Earlier versions could produce an empty root when the only period was the first character.

os.path.**splitunc**(*path*)
Split the pathname *path* into a pair (unc, rest) so that *unc* is the UNC mount point (such as r'\\host\mount'), if present, and *rest* the rest of the path (such as r'\path\file.ext'). For paths containing drive letters, *unc* will always be the empty string.

Availability: Windows.

os.path.**walk**(*path*, *visit*, *arg*)
Calls the function *visit* with arguments (arg, dirname, names) for each directory in the directory tree rooted at *path* (including *path* itself, if it is a directory). The argument *dirname* specifies the visited directory, the argument *names* lists the files in the directory (gotten from os.listdir(dirname)). The *visit* function may modify *names* to influence the set of directories visited below *dirname*, e.g. to avoid visiting certain parts of the tree. (The object referred to by *names* must be modified in place, using del or slice assignment.)

注解： Symbolic links to directories are not treated as subdirectories, and that *walk()* therefore will not visit them. To visit linked directories you must identify them with os.path.islink(file) and os.path.isdir(file), and invoke *walk()* as necessary.

注解: This function is deprecated and has been removed in Python 3 in favor of *os.walk()*.

os.path.**supports_unicode_filenames**
如果（在文件系统限制下）允许将任意 Unicode 字符串用作文件名，则为 True。

2.3 新版功能.

10.2 fileinput —迭代来自多个输入流的行

源代码: Lib/fileinput.py

此模块实现了一个辅助类和一些函数用来快速编写访问标准输入或文件列表的循环。如果你只想要读写一个文件请参阅*open()*。

典型用法为:

```
import fileinput
for line in fileinput.input():
    process(line)
```

This iterates over the lines of all files listed in sys.argv[1:], defaulting to sys.stdin if the list is empty. If a filename is '-', it is also replaced by sys.stdin. To specify an alternative list of filenames, pass it as the first argument to *input()*. A single file name is also allowed.

All files are opened in text mode by default, but you can override this by specifying the *mode* parameter in the call to *input()* or *FileInput()*. If an I/O error occurs during opening or reading a file, *IOError* is raised.

如果 sys.stdin 被使用超过一次，则第二次之后的使用将不返回任何行，除非是被交互式的使用，或都是被显式地重置 (例如使用 sys.stdin.seek(0))。

空文件打开后将立即被关闭；它们在文件列表中会被注意到的唯一情况只有当最后打开的文件为空的时候。

反回的行不会对换行符做任何处理，这意味着文件中的最后一行可能不带换行符。

想要控制文件的打开方式，你可以通过将 *openhook* 形参传给*fileinput.input()* 或*FileInput()* 来提供一个打开钩子。此钩子必须为一个函数，它接受两个参数，*filename* 和 *mode*，并返回一个以相应模式打开的文件类对象。此模块已经提供了两个有用的钩子。

以下函数是此模块的初始接口：

fileinput.**input**([*files*[, *inplace*[, *backup*[, *bufsize*[, *mode*[, *openhook*]]]]]])
创建一个*FileInput* 类的实例。该实例将被用作此模块中函数的全局状态，并且还将在迭代期间被返回使用。此函数的形参将被继续传递给*FileInput* 类的构造器。

在 2.5 版更改: Added the *mode* and *openhook* parameters.

在 2.7.12 版更改: The *bufsize* parameter is no longer used.

下列函数会使用*fileinput.input()* 所创建的全局状态；如果没有活动的状态，则会引发*RuntimeError*。

fileinput.**filename**()
返回当前被读取的文件名。在第一行被读取之前，返回 None。

fileinput.**fileno**()
返回以整数表示的当前文件“文件描述符”。当未打开文件时（处在第一行和文件之间），返回 -1。

2.5 新版功能.

fileinput.**lineno**()
返回已被读取的累计行号。在第一行被读取之前，返回 0。在最后一个文件的最后一行被读取之后，返回该行的行号。

fileinput.**filelineno**()
返回当前文件中的行号。在第一行被读取之前，返回 0。在最后一个文件的最后一行被读取之后，返回此文件中该行的行号。

fileinput.**isfirstline**()
Returns true if the line just read is the first line of its file, otherwise returns false.

fileinput.**isstdin**()
Returns true if the last line was read from sys.stdin, otherwise returns false.

fileinput.**nextfile**()
关闭当前文件以使下次迭代将从下一个文件（如果存在）读取第一行；不是从该文件读取的行将不会被计入累计行数。直到下一个文件的第一行被读取之后文件名才会改变。在第一行被读取之前，此函数将不会生效；它不能被用来跳过第一个文件。在最后一个文件的最后一行被读取之后，此函数将不再生效。

fileinput.**close**()
关闭序列。

此模块所提供的实现了序列行为的类同样也可用于子类化：

class fileinput.**FileInput**([*files*[, *inplace*[, *backup*[, *bufsize*[, *mode*[, *openhook*]]]]]])
Class *FileInput* is the implementation; its methods *filename()*, *fileno()*, *lineno()*, *filelineno()*, *isfirstline()*, *isstdin()*, *nextfile()* and *close()* correspond to the functions of the same name in the module. In addition it has a *readline()* method which returns the next input line, and a __getitem__() method which implements the sequence behavior. The sequence must be accessed in strictly sequential order; random access and *readline()* cannot be mixed.

通过 *mode* 你可以指定要传给*open()* 的文件模式。它必须为 'r', 'rU', 'U' 和 'rb' 中的一个。

openhook 如果给出则必须为一个函数，它接受两个参数 *filename* 和 *mode*，并相应地返回一个打开的文件类对象。你不能同时使用 *inplace* 和 *openhook*。

在 2.5 版更改: Added the *mode* and *openhook* parameters.

在 2.7.12 版更改: The *bufsize* parameter is no longer used.

Optional in-place filtering: if the keyword argument inplace=1 is passed to *fileinput.input()* or to the *FileInput* constructor, the file is moved to a backup file and standard output is directed to the input file (if a file of the same name as the backup file already exists, it will be replaced silently). This makes it possible to write a filter that rewrites its input file in place. If the *backup* parameter is given (typically as backup='.<some extension>'), it specifies the extension for the backup file, and the backup file remains around; by default, the extension is '.bak' and it is deleted when the output file is closed. In-place filtering is disabled when standard input is read.

注解： The current implementation does not work for MS-DOS 8+3 filesystems.

此模块提供了以下两种打开文件钩子：

fileinput.**hook_compressed**(*filename*, *mode*)
使用*gzip* 和*bz2* 模块透明地打开 gzip 和 bzip2 压缩的文件（通过扩展名 '.gz' 和 '.bz2' 来识别）。如果文件扩展名不是 '.gz' 或 '.bz2'，文件会以正常方式打开（即使用*open()* 并且不带任何解压操作）。

使用示例: fi = fileinput.FileInput(openhook=fileinput.hook_compressed)

2.5 新版功能.

fileinput.**hook_encoded**(*encoding*)
Returns a hook which opens each file with *io.open()*, using the given *encoding* to read the file.

Usage example: `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("iso-8859-1"))`

注解: With this hook, *FileInput* might return Unicode strings depending on the specified *encoding*.

2.5 新版功能.

10.3 stat —解析 stat() 结果

源代码: Lib/stat.py

---

stat 模块定义了一些用于解析*os.stat()*, *os.fstat()* 和*os.lstat()* (如果它们存在) 输出结果的常量和函数。有关 stat(), fstat() 和 lstat() 调用的完整细节，请参阅你的系统文档。

stat 模块定义了以下函数来检测特定文件类型:

stat.**S_ISDIR**(*mode*)
如果 mode 来自一个目录则返回非零值。

stat.**S_ISCHR**(*mode*)
如果 mode 来自一个字符专属的设备文件则返回非零值。

stat.**S_ISBLK**(*mode*)
如果 mode 来自一个块特殊设备文件则返回非零值。

stat.**S_ISREG**(*mode*)
如果 mode 来自一个常规文件则返回非零值。

stat.**S_ISFIFO**(*mode*)
如果 mode 来自一个 FIFO (命名管道) 则返回非零值。

stat.**S_ISLNK**(*mode*)
如果 mode 来自一个符号链接则返回非零值。

stat.**S_ISSOCK**(*mode*)
如果 mode 来自一个套接字则返回非零值。

定义了两个附加函数用于对文件模式进行更一般化的操作:

stat.**S_IMODE**(*mode*)
返回文件模式中可由*os.chmod()* 进行设置的部分—即文件的 permission 位, 加上 sticky 位、set-group-id 以及 set-user-id 位（在支持这些部分的系统上）。

stat.**S_IFMT**(*mode*)
返回文件模式中描述文件类型的部分（供上面的 S_IS*() 函数使用）。

通常，你应当使用 os.path.is*() 函数来检测文件的类型；这里提供的函数则适用于当你要对同一文件执行多项检测并且希望避免每项检测的 stat() 系统调用开销的情况。这些函数也适用于检测有关未被*os.path* 处理的信息，例如检测块和字符设备等。

示例:

```
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top,
       calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.stat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print 'Skipping %s' % pathname

def visitfile(file):
    print 'visiting', file

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

以下所有变量是一些简单的符号索引，用于访问*os.stat()*, *os.fstat()* 或*os.lstat()* 所返回的 10 条目元组。

stat.**ST_MODE**
inode 保护模式。

stat.**ST_INO**
Inode 号

stat.**ST_DEV**
Inode 所在的设备。

stat.**ST_NLINK**
Inode 拥有的链接数量。

stat.**ST_UID**
所有者的用户 ID。

stat.**ST_GID**
所有者的用户组 ID。

stat.**ST_SIZE**
以字节为单位的普通文件大小；对于某些特殊文件的预期数据量。

stat.**ST_ATIME**
上次访问的时间。

stat.**ST_MTIME**
上次修改的时间。

stat.**ST_CTIME**
操作系统所报告的“ctime”。在某些系统上（例如 Unix）是元数据的最后修改时间，而在其他系统上（例如 Windows）则是创建时间（请参阅系统平台的文档了解相关细节）。

对于“文件大小”的解析可因文件类型的不同而变化。对于普通文件就是文件的字节数。对于大部分种类的 Unix（特别包括 Linux）的 FIFO 和套接字来说，“大小”则是指在调用*os.stat()*，*os.fstat()* 或*os.lstat()* 时等待读取的字节数；这在某些时候很有用处，特别是在一个非阻塞的打开后轮询这些特殊文件中的一个时。其他字符和块设备的文件大小字段的含义还会有更多变化，具体取决于底层系统调用的实现方式。

以下变量定义了在*ST_MODE* 字段中使用的旗标。

使用上面的函数会比使用第一组旗标更容易移植：

stat.**S_IFSOCK**
套接字

stat.**S_IFLNK**
符号链接。

stat.**S_IFREG**
普通文件。

stat.**S_IFBLK**
块设备

stat.**S_IFDIR**
目录

stat.**S_IFCHR**
字符设备。

stat.**S_IFIFO**
先进先出

以下旗标还可以*os.chmod()* 的在 *mode* 参数中使用：

stat.**S_ISUID**
设置 UID 位。

stat.**S_ISGID**
设置分组 ID 位。这个位有几种特殊用途。对于目录它表示该目录将使用 BSD 语义：在其中创建的文件将从目录继承其分组 ID，而不是从创建进程的有效分组 ID 继承，并且在其中创建的目录也将设置*S_ISGID* 位。对于没有设置分组执行位 (*S_IXGRP*) 的文件，设置分组 ID 位表示强制性文件/记录锁定 (另请参见*S_ENFMT*)。

stat.**S_ISVTX**
固定位。当对目录设置该位时则意味着此目录中的文件只能由文件所有者、目录所有者或特权进程来重命名或删除。

stat.**S_IRWXU**
文件所有者权限的掩码。

stat.**S_IRUSR**
所有者具有读取权限。

stat.**S_IWUSR**
所有者具有写入权限。

stat.**S_IXUSR**
所有者具有执行权限。

stat.**S_IRWXG**
组权限的掩码。

stat.**S_IRGRP**
组具有读取权限。

stat.**S_IWGRP**
组具有写入权限。

stat.**S_IXGRP**
组具有执行权限。

stat.**S_IRWXO**
其他人（不在组中）的权限掩码。

stat.**S_IROTH**
其他人具有读取权限。

stat.**S_IWOTH**
其他人具有写入权限。

stat.**S_IXOTH**
其他人具有执行权限。

stat.**S_ENFMT**
System V 执行文件锁定。此旗标是与*S_ISGID* 共享的：文件/记录锁定会针对未设置分组执行位 (*S_IXGRP*) 的文件强制执行。

stat.**S_IREAD**
Unix V7 中*S_IRUSR* 的同义词。

stat.**S_IWRITE**
Unix V7 中*S_IWUSR* 的同义词。

stat.**S_IEXEC**
Unix V7 中*S_IXUSR* 的同义词。

以下旗标可以在*os.chflags()* 的 *flags* 参数中使用：

stat.**UF_NODUMP**
不要转储文件。

stat.**UF_IMMUTABLE**
文件不能更改。

stat.**UF_APPEND**
文件只能附加到。

stat.**UF_OPAQUE**
当通过联合堆栈查看时，目录是不透明的。

stat.**UF_NOUNLINK**
文件不能重命名或删除。

stat.**UF_COMPRESSED**
文件是压缩存储的（Mac OS X 10.6+）。

stat.**UF_HIDDEN**
文件不能显示在 GUI 中（Mac OS X 10.5+）。

stat.**SF_ARCHIVED**
文件可能已存档。

stat.**SF_IMMUTABLE**
文件不能更改。

stat.**SF_APPEND**
文件只能附加到。

stat.**SF_NOUNLINK**
文件不能重命名或删除。

stat.**SF_SNAPSHOT**
文件有一个快照文件

请参阅 *BSD 或 Mac OS 系统的指南页 *chflags(2)* 了解详情。

10.4 statvfs —Constants used with os.statvfs()

2.6 版后已移除: The *statvfs* module has been removed in Python 3.

The *statvfs* module defines constants so interpreting the result if *os.statvfs()*, which returns a tuple, can be made without remembering "magic numbers." Each of the constants defined in this module is the *index* of the entry in the tuple returned by *os.statvfs()* that contains the specified information.

statvfs.**F_BSIZE**
Preferred file system block size.

statvfs.**F_FRSIZE**
Fundamental file system block size.

statvfs.**F_BLOCKS**
Total number of blocks in the filesystem.

statvfs.**F_BFREE**
Total number of free blocks.

statvfs.**F_BAVAIL**
Free blocks available to non-super user.

statvfs.**F_FILES**
Total number of file nodes.

statvfs.**F_FFREE**
Total number of free file nodes.

statvfs.**F_FAVAIL**
Free nodes available to non-super user.

statvfs.**F_FLAG**
Flags. System dependent: see `statvfs()` man page.

statvfs.**F_NAMEMAX**
Maximum file name length.

10.5 filecmp —文件及目录的比较

源代码: Lib/filecmp.py

---

filecmp 模块定义了用于比较文件及目录的函数，并且可以选取多种关于时间和准确性的折衷方案。对于文件的比较，另见*difflib* 模块。

filecmp 模块定义了如下函数:

filecmp.**cmp**(*f1*, *f2*[, *shallow*])
比较名为 *f1* 和 *f2* 的文件，如果它们似乎相等则返回 `True` ，否则返回 `False` 。

Unless *shallow* is given and is false, files with identical *os.stat()* signatures are taken to be equal.

Files that were compared using this function will not be compared again unless their *os.stat()* signature changes.

需要注意，没有外部程序被该函数调用，这赋予了该函数可移植性与效率。

filecmp.**cmpfiles**(*dir1*, *dir2*, *common*[, *shallow*])
比较在两个目录 *dir1* 和 *dir2* 中，由 *common* 所确定名称的文件。

返回三组文件名列表：*match*, *mismatch*, *errors* 。*match* 含有相匹配的文件，*mismatch* 含有那些不匹配的，然后 *errors* 列出那些未被比较文件的名称。如果文件不存在于两目录中的任一个，或者用户缺少读取它们的权限，又或者因为其他的一些原因而无法比较，那么这些文件将会被列在 *errors* 中。

参数 *shallow* 具有同*filecmp.cmp()* 一致的含义与默认值。

例如，`cmpfiles('a', 'b', ['c', 'd/e'])` 将会比较 `a/c` 与 `b/c` 以及 `a/d/e` 与 `b/d/e` 。`'c'` 和 `'d/e'` 将会各自出现在返回的三个列表里的某一个列表中。

Example:

```
>>> import filecmp
>>> filecmp.cmp('undoc.rst', 'undoc.rst')
True
>>> filecmp.cmp('undoc.rst', 'index.rst')
False
```

10.5.1 dircmp 类

dircmp instances are built using this constructor:

class filecmp.**dircmp**(*a*, *b*[, *ignore*[, *hide*]])
Construct a new directory comparison object, to compare the directories *a* and *b*. *ignore* is a list of names to ignore, and defaults to `['RCS', 'CVS', 'tags']`. *hide* is a list of names to hide, and defaults to `[os.curdir, os.pardir]`.

dircmp 类如*filecmp.cmp()* 中所描述的那样对文件进行 *shallow* 比较。

dircmp 类提供以下方法：

report()
Print (to `sys.stdout`) a comparison between *a* and *b*.

report_partial_closure()
打印 *a* 与 *b* 及共同直接子目录的比较结果。

report_full_closure()
打印 *a* 与 *b* 及共同子目录比较结果（递归地）。

dircmp 类提供了一些有趣的属性，用以得到关于参与比较的目录树的各种信息。

需要注意，通过 `__getattr__()` 钩子，所有的属性将会惰性求值，因此如果只使用那些计算简便的属性，将不会有速度损失。

left
目录 *a* 。

right
目录 *b* 。

left_list
经 *hide* 和 *ignore* 过滤，目录 *a* 中的文件与子目录。

right_list
经 *hide* 和 *ignore* 过滤，目录 *b* 中的文件与子目录。

common
同时存在于目录 *a* 和 *b* 中的文件和子目录。

left_only
仅在目录 *a* 中的文件和子目录。

right_only
仅在目录 *b* 中的文件和子目录。

common_dirs
同时存在于目录 *a* 和 *b* 中的子目录。

common_files
Files in both *a* and *b*

common_funny
在目录 *a* 和 *b* 中类型不同的名字，或者那些*os.stat()* 报告错误的名字。

same_files
在目录 *a* 和 *b* 中，使用类的文件比较操作符判定相等的文件。

diff_files
在目录 *a* 和 *b* 中，根据类的文件比较操作符判定内容不等的文件。

funny_files
在目录 *a* 和 *b* 中无法比较的文件。

subdirs
一个将*common_dirs* 中名称映射为*dircmp* 对象的字典。

下面是一个简单的例子，使用 subdirs 属性递归搜索两个目录以显示公共差异文件：

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print "diff_file %s found in %s and %s" % (name, dcmp.left,
...               dcmp.right)
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```

10.6 tempfile —生成临时文件和目录

源代码： Lib/tempfile.py

This module generates temporary files and directories. It works on all supported platforms.

In version 2.3 of Python, this module was overhauled for enhanced security. It now provides three new functions, *NamedTemporaryFile()*, *mkstemp()*, and *mkdtemp()*, which should eliminate all remaining need to use the

insecure *mktemp()* function. Temporary file names created by this module no longer contain the process ID; instead a string of six random characters is used.

Also, all the user-callable functions now take additional arguments which allow direct control over the location and name of temporary files. It is no longer necessary to use the global *tempdir* and *template* variables. To maintain backward compatibility, the argument order is somewhat odd; it is recommended to use keyword arguments for clarity.

The module defines the following user-callable functions:

tempfile.**TemporaryFile**([*mode='w+b'*[, *bufsize=-1*[, *suffix=''*[, *prefix='tmp'*[, *dir=None*]]]]])

Return a file-like object that can be used as a temporary storage area. The file is created using *mkstemp()*. It will be destroyed as soon as it is closed (including an implicit close when the object is garbage collected). Under Unix, the directory entry for the file is removed immediately after the file is created. Other platforms do not support this; your code should not rely on a temporary file created using this function having or not having a visible name in the file system.

The *mode* parameter defaults to `'w+b'` so that the file created can be read and written without being closed. Binary mode is used so that it behaves consistently on all platforms without regard for the data that is stored. *bufsize* defaults to `-1`, meaning that the operating system default is used.

The *dir*, *prefix* and *suffix* parameters are passed to *mkstemp()*.

The returned object is a true file object on POSIX platforms. On other platforms, it is a file-like object whose `file` attribute is the underlying true file object. This file-like object can be used in a `with` statement, just like a normal file.

tempfile.**NamedTemporaryFile**([*mode='w+b'*[, *bufsize=-1*[, *suffix=''*[, *prefix='tmp'*[, *dir=None*[, *delete=True*]]]]]])

This function operates exactly as *TemporaryFile()* does, except that the file is guaranteed to have a visible name in the file system (on Unix, the directory entry is not unlinked). That name can be retrieved from the `name` attribute of the returned file-like object. Whether the name can be used to open the file a second time, while the named temporary file is still open, varies across platforms (it can be so used on Unix; it cannot on Windows NT or later). If *delete* is true (the default), the file is deleted as soon as it is closed.

The returned object is always a file-like object whose `file` attribute is the underlying true file object. This file-like object can be used in a `with` statement, just like a normal file.

2.3 新版功能.

2.6 新版功能: The *delete* parameter.

tempfile.**SpooledTemporaryFile**([*max_size=0*[, *mode='w+b'*[, *bufsize=-1*[, *suffix=''*[, *prefix='tmp'*[, *dir=None*]]]]]])

This function operates exactly as *TemporaryFile()* does, except that data is spooled in memory until the file size exceeds *max_size*, or until the file's `fileno()` method is called, at which point the contents are written to disk and operation proceeds as with *TemporaryFile()*. Also, it's `truncate` method does not accept a `size` argument.

此函数生成的文件对象有一个额外的方法——`rollover()`，可以忽略文件大小，让文件立即写入磁盘。

The returned object is a file-like object whose `_file` attribute is either a *StringIO* object or a true file object, depending on whether `rollover()` has been called. This file-like object can be used in a `with` statement, just like a normal file.

2.6 新版功能.

tempfile.**mkstemp**([*suffix=''*[, *prefix='tmp'*[, *dir=None*[, *text=False*]]]])

以最安全的方式创建一个临时文件。假设所在平台正确实现了*os.open()* 的*os.O_EXCL* 标志，则创建文件时不会有竞争的情况。该文件只能由创建者读写，如果所在平台用权限位来标记文件是否可执行，那么没有人有执行权。文件描述符不会过继给子进程。

与*TemporaryFile()* 不同，*mkstemp()* 用户用完临时文件后需要自行将其删除。

If *suffix* is specified, the file name will end with that suffix, otherwise there will be no suffix. *mkstemp()* does not put a dot between the file name and the suffix; if you need one, put it at the beginning of *suffix*.

If *prefix* is specified, the file name will begin with that prefix; otherwise, a default prefix is used.

If *dir* is specified, the file will be created in that directory; otherwise, a default directory is used. The default directory is chosen from a platform-dependent list, but the user of the application can control the directory location by setting the *TMPDIR*, *TEMP* or *TMP* environment variables. There is thus no guarantee that the generated filename will have any nice properties, such as not requiring quoting when passed to external commands via `os.popen()`.

如果指定了 *text* 参数，它表示的是以二进制模式（默认）还是文本模式打开文件。在某些平台上，两种模式没有区别。

mkstemp() 返回一个元组，元组中第一个元素是句柄，它是一个系统级句柄，指向一个打开的文件（等同于*os.open()* 的返回值），第二元素是该文件的绝对路径。

2.3 新版功能.

tempfile.**mkdtemp**([*suffix=''*[, *prefix='tmp'*[, *dir=None*]]])
以最安全的方式创建一个临时目录，创建该目录时不会有竞争的情况。该目录只能由创建者读取、写入和搜索。

mkdtemp() 用户用完临时目录后需要自行将其删除。

prefix、*suffix* 和 *dir* 的含义与它们在*mkstemp()* 中的相同。

mkdtemp() 返回新目录的绝对路径名。

2.3 新版功能.

tempfile.**mktemp**([*suffix=''*[, *prefix='tmp'*[, *dir=None*]]])
2.3 版后已移除: 使用*mkstemp()* 来代替。

Return an absolute pathname of a file that did not exist at the time the call is made. The *prefix*, *suffix*, and *dir* arguments are the same as for *mkstemp()*.

警告: 使用此功能可能会在程序中引入安全漏洞。当你开始使用本方法返回的文件执行任何操作时，可能有人已经捷足先登了。*mktemp()* 的功能可以很轻松地用*NamedTemporaryFile()* 代替，当然需要传递 `delete=False` 参数:

```
>>> f = NamedTemporaryFile(delete=False)
>>> f
<open file '<fdopen>', mode 'w+b' at 0x384698>
>>> f.name
'/var/folders/5q/5qTPn6xq2RaWqk+1Ytw3-U+++TI/-Tmp-/tmpG7V1Y0'
>>> f.write("Hello World!\n")
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

The module uses a global variable that tell it how to construct a temporary name. They are initialized at the first call to any of the functions above. The caller may change them, but this is discouraged; use the appropriate function arguments, instead.

tempfile.**tempdir**
When set to a value other than `None`, this variable defines the default value for the *dir* argument to all the functions defined in this module.

If `tempdir` is unset or `None` at any call to any of the above functions, Python searches a standard list of directories and sets *tempdir* to the first one which the calling user can create files in. The list is:

1. `TMPDIR` 环境变量指向的目录。
2. `TEMP` 环境变量指向的目录。
3. `TMP` 环境变量指向的目录。
4. 与平台相关的位置：
   - On RiscOS, the directory named by the `Wimp$ScrapDir` environment variable.
   - 在 Windows 上，依次为 `C:\TEMP`、`C:\TMP`、`\TEMP` 和 `\TMP`。
   - 在所有其他平台上，依次为 `/tmp`、`/var/tmp` 和 `/usr/tmp`。
5. 不得已时，使用当前工作目录。

`tempfile.`**`gettempdir`**`()`
Return the directory currently selected to create temporary files in. If *tempdir* is not `None`, this simply returns its contents; otherwise, the search described above is performed, and the result returned.

2.3 新版功能.

`tempfile.`**`template`**
2.0 版后已移除: Use *gettempprefix()* instead.

When set to a value other than `None`, this variable defines the prefix of the final component of the filenames returned by *mktemp()*. A string of six random letters and digits is appended to the prefix to make the filename unique. The default prefix is `tmp`.

Older versions of this module used to require that `template` be set to `None` after a call to *os.fork()*; this has not been necessary since version 1.5.2.

`tempfile.`**`gettempprefix`**`()`
Return the filename prefix used to create temporary files. This does not contain the directory component. Using this function is preferred over reading the *template* variable directly.

1.5.2 新版功能.

10.7 `glob` —Unix 风格路径名模式扩展

源代码: Lib/glob.py

The *glob* module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order. No tilde expansion is done, but `*`, `?`, and character ranges expressed with `[]` will be correctly matched. This is done by using the *os.listdir()* and *fnmatch.fnmatch()* functions in concert, and not by actually invoking a subshell. Note that unlike *fnmatch.fnmatch()*, *glob* treats filenames beginning with a dot (`.`) as special cases. (For tilde and shell variable expansion, use *os.path.expanduser()* and *os.path.expandvars()*.)

对于字面值匹配，请将原字符用方括号括起来。例如，`'[?]'` 将匹配字符 `'?'`。

`glob.`**`glob`**`(`*pathname*`)`
返回匹配 *pathname* 的可能为空的路径名列表，路径名必须为包含一个路径描述的字符串。*pathname* 可以是绝对路径 (如 `/usr/src/Python-1.5/Makefile`) 或相对路径 (如 `../../Tools/*/*.gif`)，并且可包含 shell 风格的通配符。无效的符号链接可以包含在结果中 (与在 shell 中一样)。

glob.**iglob**(*pathname*)
返回一个*iterator*，它会产生与*glob()* 相同的结果，但不会实际地同时保存它们。

2.5 新版功能.

For example, consider a directory containing only the following files: `1.gif`, `2.txt`, and `card.gif`. *glob()* will produce the following results. Notice how any leading components of the path are preserved.

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
```

如果目录包含以 `.` 打头的文件，它们默认将不会被匹配。例如，考虑一个包含 `card.gif` 和 `.card.gif` 的目录:

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

参见:

模块*fnmatch* Shell 风格文件名（而非路径）扩展

10.8 fnmatch —Unix 文件名模式匹配

源代码: Lib/fnmatch.py

此模块提供了 Unix shell 风格的通配符，它们 并不等同于正则表达式（关于后者的文档参见*re* 模块）。shell 风格通配符所使用的特殊字符如下:

模式	含义
`*`	匹配所有
`?`	匹配任何单个字符
`[seq]`	匹配 *seq* 中的任何字符
`[!seq]`	匹配任何不在 *seq* 中的字符

对于字面值匹配，请将原字符用方括号括起来。例如，`'[?]'` 将匹配字符 `'?'`。

注意文件名分隔符 (Unix 上为 `'/'`) 不是此模块所特有的。请参见*glob* 模块了解文件名扩展 (*glob* 使用*filter()* 来匹配文件名的各个部分)。类似地，以一个句点打头的文件名也不是此模块所特有的，可以通过 `*` 和 `?` 模式来匹配。

fnmatch.**fnmatch**(*filename*, *pattern*)
检测 *filename* 字符串是否匹配 *pattern* 字符串，返回*True* 或*False*。两个形参都会使用*os.path.normcase()* 进行大小写正规化。*fnmatchcase()* 可被用于执行大小写敏感的比较，无论这是否为所在操作系统的标准。

这个例子将打印当前目录下带有扩展名 `.txt` 的所有文件名:

```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print file
```

fnmatch.**fnmatchcase**(*filename*, *pattern*)
检测 *filename* 是否匹配 *pattern*，返回*True*或*False*；此比较是大小写敏感的，并且不会应用*os.path.normcase()*。

fnmatch.**filter**(*names*, *pattern*)
返回 *names* 列表中匹配 *pattern* 的子集。它等价于 `[n for n in names if fnmatch(n, pattern)]`，但其实现更为高效。

2.2 新版功能.

fnmatch.**translate**(*pattern*)
返回 shell 风格 *pattern* 转换成的正则表达式以便用于*re.match()*。

示例:

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'.*\\.txt\\Z(?ms)'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<_sre.SRE_Match object at 0x...>
```

参见:

模块*glob* Unix shell 风格路径扩展。

10.9 linecache —随机读写文本行

源代码: Lib/linecache.py

The *linecache* module allows one to get any line from any file, while attempting to optimize internally, using a cache, the common case where many lines are read from a single file. This is used by the *traceback* module to retrieve source lines for inclusion in the formatted traceback.

linecache 模块定义了下列函数:

linecache.**getline**(*filename*, *lineno*[, *module_globals*])
从名为 *filename* 的文件中获取 *lineno* 行，此函数绝不会引发异常—出现错误时它将返回 `''` (所有找到的行都将包含换行符作为结束)。

如果名为 *filename* 的文件未找到，该函数将在模块搜索路径 `sys.path` 中查找它，在此之前会先在 *module_globals* 中检查 **PEP 302** `__loader__`，以涵盖模块是从 zip 文件或其他非文件系统导入源导入的情况。

2.5 新版功能: The *module_globals* parameter was added.

linecache.**clearcache**()
清空缓存。如果你不再需要之前使用*getline()* 从文件读取的行即可使用此函数。

linecache.**checkcache**([*filename*])
检查缓存有效性。如果缓存中的文件在磁盘上发生了改变，而你需要更新后的版本即可使用此函数。如果省略了 *filename*，它会检查缓存中的所有条目。

示例:

```
>>> import linecache
>>> linecache.getline('/etc/passwd', 4)
'sys:x:3:3:sys:/dev:/bin/sh\n'
```

10.10 shutil —高阶文件操作

源代码： Lib/shutil.py

---

shutil 模块提供了一系列对文件和文件集合的高阶操作。特别是提供了一些支持文件拷贝和删除的函数。对于单个文件的操作，请参阅*os* 模块。

> **警告：** Even the higher-level file copying functions (*shutil.copy()*, *shutil.copy2()*) can't copy all file metadata.
>
> 在 POSIX 平台上，这意味着将丢失文件所有者和组以及 ACL 数据。在 Mac OS 上，资源钩子和其他元数据不被使用。这意味着将丢失这些资源并且文件类型和创建者代码将不正确。在 Windows 上，将不会拷贝文件所有者、ACL 和替代数据流。

10.10.1 目录和文件操作

shutil.**copyfileobj**(*fsrc*, *fdst*[, *length*])
将文件类对象 *fsrc* 的内容拷贝到文件类对象 *fdst*。整数值 *length* 如果给出则为缓冲区大小。特别地，*length* 为负值表示拷贝数据时不对源数据进行分块循环处理；默认情况下会分块读取数据以避免不受控制的内存消耗。请注意如果 *fsrc* 对象的当前文件位置不为 0，则只有从当前文件位置到文件末尾的内容会被拷贝。

shutil.**copyfile**(*src*, *dst*)
Copy the contents (no metadata) of the file named *src* to a file named *dst*. *dst* must be the complete target file name; look at *shutil.copy()* for a copy that accepts a target directory path. If *src* and *dst* are the same files, *Error* is raised. The destination location must be writable; otherwise, an *IOError* exception will be raised. If *dst* already exists, it will be replaced. Special files such as character or block devices and pipes cannot be copied with this function. *src* and *dst* are path names given as strings.

shutil.**copymode**(*src*, *dst*)
Copy the permission bits from *src* to *dst*. The file contents, owner, and group are unaffected. *src* and *dst* are path names given as strings.

shutil.**copystat**(*src*, *dst*)
Copy the permission bits, last access time, last modification time, and flags from *src* to *dst*. The file contents, owner, and group are unaffected. *src* and *dst* are path names given as strings.

shutil.**copy**(*src*, *dst*)
Copy the file *src* to the file or directory *dst*. If *dst* is a directory, a file with the same basename as *src* is created (or overwritten) in the directory specified. Permission bits are copied. *src* and *dst* are path names given as strings.

shutil.**copy2**(*src*, *dst*)
类似于*copy()*，区别在于*copy2()* 还会尝试保留文件的元数据。

copy2() uses *copystat()* to copy the file metadata. Please see *copystat()* for more information.

shutil.**ignore_patterns**(**patterns*)
这个工厂函数会创建一个函数，它可被用作*copytree()* 的 *ignore* 可调用对象参数，以忽略那些匹配所提供的 glob 风格的 *patterns* 之一的文件和目录。参见以下示例。

2.6 新版功能.

shutil.**copytree**(*src*, *dst*, *symlinks=False*, *ignore=None*)
Recursively copy an entire directory tree rooted at *src*. The destination directory, named by *dst*, must not already exist; it will be created as well as missing parent directories. Permissions and times of directories are copied with *copystat()*, individual files are copied using *shutil.copy2()*.

If *symlinks* is true, symbolic links in the source tree are represented as symbolic links in the new tree, but the metadata of the original links is NOT copied; if false or omitted, the contents and metadata of the linked files are copied to the new tree.

如果给出了 *ignore*，它必须是一个可调用对象，该对象将接受*copytree()* 所访问的目录以及*os.listdir()* 所返回的目录内容列表作为其参数。由于*copytree()* 是递归地被调用的，*ignore* 可调用对象对于每个被拷贝目录都将被调用一次。该可调用对象必须返回一个相对于当前目录的目录和文件名序列（即其第二个参数的子集）；随后这些名称将在拷贝进程中被忽略。*ignore_patterns()* 可被用于创建这种基于 glob 风格模式来忽略特定名称的可调用对象。

如果发生了异常，将引发一个附带原因列表的*Error*。

The source code for this should be considered an example rather than the ultimate tool.

在 2.3 版更改: *Error* is raised if any exceptions occur during copying, rather than printing a message.

在 2.5 版更改: Create intermediate directories needed to create *dst*, rather than raising an error. Copy permissions and times of directories using *copystat()*.

在 2.6 版更改: Added the *ignore* argument to be able to influence what is being copied.

shutil.**rmtree**(*path*[, *ignore_errors*[, *onerror*]])
删除一个完整的目录树；*path* 必须指向一个目录（但不能是一个目录的符号链接）。如果 *ignore_errors* 为真值，删除失败导致的错误将被忽略；如果为假值或是省略，此类错误将通过调用由 *onerror* 所指定的处理程序来处理，或者如果此参数被省略则将引发一个异常。

If *onerror* is provided, it must be a callable that accepts three parameters: *function*, *path*, and *excinfo*. The first parameter, *function*, is the function which raised the exception; it will be *os.path.islink()*, *os.listdir()*, *os.remove()* or *os.rmdir()*. The second parameter, *path*, will be the path name passed to *function*. The third parameter, *excinfo*, will be the exception information return by *sys.exc_info()*. Exceptions raised by *onerror* will not be caught.

在 2.6 版更改: Explicitly check for *path* being a symbolic link and raise *OSError* in that case.

shutil.**move**(*src*, *dst*)
Recursively move a file or directory (*src*) to another location (*dst*).

如果目标是已存在的目录，则 *src* 会被移至该目录下。如果目标已存在但不是目录，它可能会被覆盖，具体取决于*os.rename()* 的语义。

If the destination is on the current filesystem, then *os.rename()* is used. Otherwise, *src* is copied (using *shutil.copy2()*) to *dst* and then removed.

2.3 新版功能.

exception shutil.**Error**
此异常会收集在多文件操作期间所引发的异常。对于*copytree()*，此异常参数将是一个由三元组 (*srcname*, *dstname*, *exception*) 构成的列表。

2.3 新版功能.

copytree 示例

这个示例就是上面所描述的*copytree()* 函数的实现，其中省略了文档字符串。它还展示了此模块所提供的许多其他函数。

```
def copytree(src, dst, symlinks=False, ignore=None):
    names = os.listdir(src)
    if ignore is not None:
        ignored_names = ignore(src, names)
    else:
        ignored_names = set()

    os.makedirs(dst)
    errors = []
    for name in names:
        if name in ignored_names:
            continue
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks, ignore)
            else:
                copy2(srcname, dstname)
            # XXX What about devices, sockets etc.?
        except (IOError, os.error) as why:
            errors.append((srcname, dstname, str(why)))
        # catch the Error from the recursive copytree so that we can
        # continue with other files
        except Error as err:
            errors.extend(err.args[0])
    try:
        copystat(src, dst)
    except WindowsError:
        # can't copy file access times on Windows
        pass
    except OSError as why:
        errors.extend((src, dst, str(why)))
    if errors:
        raise Error(errors)
```

另一个使用*ignore_patterns()* 辅助函数的例子:

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

这将会拷贝除 .pyc 文件和以 tmp 打头的文件或目录以外的所有条目.

另一个使用 *ignore* 参数来添加记录调用的例子:

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s' % path)
    return []   # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

10.10.2 归档操作

本模块也提供了用于创建和读取压缩和归档文件的高层级工具。它们依赖于*zipfile* 和*tarfile* 模块。

shutil.**make_archive**(*base_name*, *format*[, *root_dir*[, *base_dir*[, *verbose*[, *dry_run*[, *owner*[, *group*[, *logger*]]]]]]])

Create an archive file (eg. zip or tar) and returns its name.

base_name is the name of the file to create, including the path, minus any format-specific extension. *format* is the archive format: one of "zip" (if the *zlib* module or external `zip` executable is available), "tar", "gztar" (if the *zlib* module is available), or "bztar" (if the *bz2* module is available).

root_dir is a directory that will be the root directory of the archive; ie. we typically chdir into *root_dir* before creating the archive.

base_dir is the directory where we start archiving from; ie. *base_dir* will be the common prefix of all files and directories in the archive.

root_dir 和 *base_dir* 默认均为当前目录。

owner 和 *group* 将在创建 tar 归档文件时被使用。默认会使用当前的所有者和分组。

logger 必须是一个兼容 **PEP 282** 的对象，通常为*logging.Logger* 的实例。

2.7 新版功能.

shutil.**get_archive_formats**()

返回支持的归档格式列表。所返回序列中的每个元素为一个元组 `(name, description)`。

默认情况下*shutil* 提供以下格式:

- *zip*: ZIP file (if the *zlib* module or external `zip` executable is available).
- *tar*: 未压缩的 tar 文件。
- *gztar*: gzip 压缩的 tar 文件（如果*zlib* 模块可用）。
- *bztar*: bzip2 压缩的 tar 文件（如果*bz2* 模块可用）。

你可以通过使用*register_archive_format()* 注册新的格式或为任何现有格式提供你自己的归档程序。

2.7 新版功能.

shutil.**register_archive_format**(*name*, *function*[, *extra_args*[, *description*]])

Register an archiver for the format *name*. *function* is a callable that will be used to invoke the archiver.

If given, *extra_args* is a sequence of `(name, value)` that will be used as extra keywords arguments when the archiver callable is used.

description is used by *get_archive_formats()* which returns the list of archivers. Defaults to an empty list.

2.7 新版功能.

shutil.**unregister_archive_format**(*name*)
从支持的格式中移除归档格式 *name*。

2.7 新版功能.

归档程序示例

在这个示例中，我们创建了一个 gzip 压缩的 tar 归档文件，其中包含用户的 .ssh 目录下的所有文件:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

结果归档文件中包含有:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx------ tarek/staff       0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff     609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff      65 2008-06-09 13:26:54 ./config
-rwx------ tarek/staff     668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff     609 2008-06-09 13:26:54 ./id_dsa.pub
-rw------- tarek/staff    1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff     397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff   37192 2010-02-06 18:23:10 ./known_hosts
```

10.11 dircache —Cached directory listings

2.6 版后已移除: The *dircache* module has been removed in Python 3.

The *dircache* module defines a function for reading directory listing using a cache, and cache invalidation using the *mtime* of the directory. Additionally, it defines a function to annotate directories by appending a slash.

The *dircache* module defines the following functions:

dircache.**reset**()
Resets the directory cache.

dircache.**listdir**(*path*)
Return a directory listing of *path*, as gotten from *os.listdir()*. Note that unless *path* changes, further call to *listdir()* will not re-read the directory structure.

Note that the list returned should be regarded as read-only. (Perhaps a future version should change it to return a tuple?)

dircache.**opendir**(*path*)
Same as *listdir()*. Defined for backwards compatibility.

dircache.**annotate**(*head*, *list*)
Assume *list* is a list of paths relative to *head*, and append, in place, a '/' to each path which points to a directory.

```
>>> import dircache
>>> a = dircache.listdir('/')
>>> a = a[:] # Copy the return value so we can change 'a'
>>> a
['bin', 'boot', 'cdrom', 'dev', 'etc', 'floppy', 'home', 'initrd', 'lib', 'lost+
found', 'mnt', 'proc', 'root', 'sbin', 'tmp', 'usr', 'var', 'vmlinuz']
>>> dircache.annotate('/', a)
>>> a
['bin/', 'boot/', 'cdrom/', 'dev/', 'etc/', 'floppy/', 'home/', 'initrd/', 'lib/
', 'lost+found/', 'mnt/', 'proc/', 'root/', 'sbin/', 'tmp/', 'usr/', 'var/', 'vm
linuz']
```

10.12 macpath —Mac OS 9 路径操作函数

该模块是*os.path*模块的 Mac OS 9（及更早版本）实现。它可用于在 Mac OS X（或任何其他平台）上操作旧式 Macintosh 路径名。

该模块中提供了以下函数：normcase()、normpath()、isabs()、join()、split()、isdir()、isfile()、walk()、exists()。对于其他可用的函数*os.path*虚拟对应可用。

参见：

Section ***File Objects*** A description of Python’s built-in file objects.

模块*os* Operating system interfaces, including functions to work with files at a lower level than the built-in file object.

CHAPTER 11

数据持久化

The modules described in this chapter support storing Python data in a persistent form on disk. The *pickle* and *marshal* modules can turn many Python data types into a stream of bytes and then recreate the objects from the bytes. The various DBM-related modules support a family of hash-based file formats that store a mapping of strings to other strings. The *bsddb* module also provides such disk-based string-to-string mappings based on hashing, and also supports B-Tree and record-based formats.

本章中描述的模块列表是:

11.1 pickle ——Python 对象序列化

The *pickle* module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as "serialization", "marshalling,"[1] or "flattening", however, to avoid confusion, the terms used here are "pickling" and "unpickling".

This documentation describes both the *pickle* module and the *cPickle* module.

警告: *pickle* 模块在接受被错误地构造或者被恶意地构造的数据时不安全。永远不要 unpickle 来自于不受信任的或者未经验证的来源的数据。

[1] 不要把它与*marshal* 模块混淆。

11.1.1 与其他 Python 模块间的关系

The *pickle* module has an optimized cousin called the *cPickle* module. As its name implies, *cPickle* is written in C, so it can be up to 1000 times faster than *pickle*. However it does not support subclassing of the *Pickler()* and *Unpickler()* classes, because in *cPickle* these are functions, not classes. Most applications have no need for this functionality, and can benefit from the improved performance of *cPickle*. Other than that, the interfaces of the two modules are nearly identical; the common interface is described in this manual and differences are pointed out where necessary. In the following discussions, we use the term "pickle" to collectively describe the *pickle* and *cPickle* modules.

The data streams the two modules produce are guaranteed to be interchangeable.

Python 有一个更原始的序列化模块称为*marshal*，但一般地*pickle* 应该是序列化 Python 对象时的首选。*marshal* 存在主要是为了支持 Python 的 `.pyc` 文件.

pickle 模块与*marshal* 在如下几方面显著地不同：

- *pickle* 模块会跟踪已被序列化的对象，所以该对象之后再次被引用时不会再次被序列化。*marshal* 不会这么做。

  这隐含了递归对象和共享对象。递归对象指包含对自己的引用的对象。这种对象并不会被 marshal 接受，并且实际上尝试 marshal 递归对象会让你的 Python 解释器崩溃。对象共享发生在对象层级中存在多处引用同一对象时。*pickle* 只会存储这些对象一次，并确保其他的引用指向同一个主副本。共享对象将保持共享，这可能对可变对象非常重要。

- *marshal* 不能被用于序列化用户定义类及其实例。*pickle* 能够透明地存储并保存类实例，然而此时类定义必须能够从与被存储时相同的模块被引入。

- The *marshal* serialization format is not guaranteed to be portable across Python versions. Because its primary job in life is to support `.pyc` files, the Python implementers reserve the right to change the serialization format in non-backwards compatible ways should the need arise. The *pickle* serialization format is guaranteed to be backwards compatible across Python releases.

Note that serialization is a more primitive notion than persistence; although *pickle* reads and writes file objects, it does not handle the issue of naming persistent objects, nor the (even more complicated) issue of concurrent access to persistent objects. The *pickle* module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure. Perhaps the most obvious thing to do with these byte streams is to write them onto a file, but it is also conceivable to send them across a network or store them in a database. The module *shelve* provides a simple interface to pickle and unpickle objects on DBM-style database files.

11.1.2 数据流格式

The data format used by *pickle* is Python-specific. This has the advantage that there are no restrictions imposed by external standards such as XDR (which can't represent pointer sharing); however it means that non-Python programs may not be able to reconstruct pickled Python objects.

By default, the *pickle* data format uses a printable ASCII representation. This is slightly more voluminous than a binary representation. The big advantage of using printable ASCII (and of some other characteristics of *pickle*'s representation) is that for debugging or recovery purposes it is possible for a human to read the pickled file with a standard text editor.

There are currently 3 different protocols which can be used for pickling.

- Protocol version 0 is the original ASCII protocol and is backwards compatible with earlier versions of Python.
- Protocol version 1 is the old binary format which is also compatible with earlier versions of Python.
- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of *new-style class*es.

Refer to **PEP 307** for more information.

If a *protocol* is not specified, protocol 0 is used. If *protocol* is specified as a negative value or *HIGHEST_PROTOCOL*, the highest protocol version available will be used.

在 2.3 版更改: Introduced the *protocol* parameter.

A binary format, which is slightly more efficient, can be chosen by specifying a *protocol* version >= 1.

11.1.3 Usage

To serialize an object hierarchy, you first create a pickler, then you call the pickler's *dump()* method. To de-serialize a data stream, you first create an unpickler, then you call the unpickler's *load()* method. The *pickle* module provides the following constant:

pickle.**HIGHEST_PROTOCOL**
The highest protocol version available. This value can be passed as a *protocol* value.

2.3 新版功能.

注解: Be sure to always open pickle files created with protocols >= 1 in binary mode. For the old ASCII-based pickle protocol 0 you can use either text mode or binary mode as long as you stay consistent.

A pickle file written with protocol 0 in binary mode will contain lone linefeeds as line terminators and therefore will look "funny" when viewed in Notepad or other editors which do not support this format.

---

pickle 模块提供了以下方法，让打包过程更加方便。

pickle.**dump**(*obj*, *file*[, *protocol*])
Write a pickled representation of *obj* to the open file object *file*. This is equivalent to `Pickler(file, protocol).dump(obj)`.

If the *protocol* parameter is omitted, protocol 0 is used. If *protocol* is specified as a negative value or *HIGHEST_PROTOCOL*, the highest protocol version will be used.

在 2.3 版更改: Introduced the *protocol* parameter.

file must have a `write()` method that accepts a single string argument. It can thus be a file object opened for writing, a *StringIO* object, or any other custom object that meets this interface.

pickle.**load**(*file*)
Read a string from the open file object *file* and interpret it as a pickle data stream, reconstructing and returning the original object hierarchy. This is equivalent to `Unpickler(file).load()`.

file must have two methods, a `read()` method that takes an integer argument, and a *readline()* method that requires no arguments. Both methods should return a string. Thus *file* can be a file object opened for reading, a *StringIO* object, or any other custom object that meets this interface.

This function automatically determines whether the data stream was written in binary mode or not.

pickle.**dumps**(*obj*[, *protocol*])
Return the pickled representation of the object as a string, instead of writing it to a file.

If the *protocol* parameter is omitted, protocol 0 is used. If *protocol* is specified as a negative value or *HIGHEST_PROTOCOL*, the highest protocol version will be used.

在 2.3 版更改: The *protocol* parameter was added.

pickle.**loads**(*string*)
Read a pickled object hierarchy from a string. Characters in the string past the pickled object's representation are ignored.

The *pickle* module also defines three exceptions:

exception pickle.**PickleError**
A common base class for the other exceptions defined below. This inherits from *Exception*.

exception pickle.**PicklingError**
This exception is raised when an unpicklable object is passed to the *dump()* method.

exception pickle.**UnpicklingError**
This exception is raised when there is a problem unpickling an object. Note that other exceptions may also be raised during unpickling, including (but not necessarily limited to) *AttributeError*, *EOFError*, *ImportError*, and *IndexError*.

The *pickle* module also exports two callables[2], *Pickler* and *Unpickler*:

class pickle.**Pickler**(*file*[, *protocol*])
This takes a file-like object to which it will write a pickle data stream.

If the *protocol* parameter is omitted, protocol 0 is used. If *protocol* is specified as a negative value or *HIGHEST_PROTOCOL*, the highest protocol version will be used.

在 2.3 版更改: Introduced the *protocol* parameter.

file must have a `write()` method that accepts a single string argument. It can thus be an open file object, a *StringIO* object, or any other custom object that meets this interface.

Pickler objects define one (or two) public methods:

dump(*obj*)
Write a pickled representation of *obj* to the open file object given in the constructor. Either the binary or ASCII format will be used, depending on the value of the *protocol* argument passed to the constructor.

clear_memo()
Clears the pickler's "memo". The memo is the data structure that remembers which objects the pickler has already seen, so that shared or recursive objects pickled by reference and not by value. This method is useful when re-using picklers.

注解: Prior to Python 2.3, *clear_memo()* was only available on the picklers created by *cPickle*. In the *pickle* module, picklers have an instance variable called `memo` which is a Python dictionary. So to clear the memo for a *pickle* module pickler, you could do the following:

```
mypickler.memo.clear()
```

Code that does not need to support older versions of Python should simply use *clear_memo()*.

It is possible to make multiple calls to the *dump()* method of the same *Pickler* instance. These must then be matched to the same number of calls to the *load()* method of the corresponding *Unpickler* instance. If the same object is pickled by multiple *dump()* calls, the *load()* will all yield references to the same object.[3]

Unpickler objects are defined as:

[2] In the *pickle* module these callables are classes, which you could subclass to customize the behavior. However, in the *cPickle* module these callables are factory functions and so cannot be subclassed. One common reason to subclass is to control what objects can actually be unpickled. See section *Subclassing Unpicklers* for more details.

[3] *Warning*: this is intended for pickling multiple objects without intervening modifications to the objects or their parts. If you modify an object and then pickle it again using the same `Pickler` instance, the object is not pickled again —a reference to it is pickled and the `Unpickler` will return the old value, not the modified one. There are two problems here: (1) detecting changes, and (2) marshalling a minimal set of changes. Garbage Collection may also become a problem here.

class pickle.**Unpickler**(*file*)

This takes a file-like object from which it will read a pickle data stream. This class automatically determines whether the data stream was written in binary mode or not, so it does not need a flag as in the *Pickler* factory.

file must have two methods, a `read()` method that takes an integer argument, and a *readline()* method that requires no arguments. Both methods should return a string. Thus *file* can be a file object opened for reading, a *StringIO* object, or any other custom object that meets this interface.

Unpickler objects have one (or two) public methods:

load()

Read a pickled object representation from the open file object given in the constructor, and return the reconstituted object hierarchy specified therein.

This method automatically determines whether the data stream was written in binary mode or not.

noload()

This is just like *load()* except that it doesn't actually create any objects. This is useful primarily for finding what's called "persistent ids" that may be referenced in a pickle data stream. See section *The pickle protocol* below for more details.

Note: the *noload()* method is currently only available on *Unpickler* objects created with the *cPickle* module. *pickle* module *Unpickler*s do not have the *noload()* method.

11.1.4 可以被打包/解包的对象

下列类型可以被打包：

- `None`、`True` 和 `False`
- integers, long integers, floating point numbers, complex numbers
- normal and Unicode strings
- 只包含可打包对象的集合，包括 tuple、list、set 和 dict
- functions defined at the top level of a module
- 定义在模块顶层的内置函数
- 定义在模块顶层的类
- instances of such classes whose *__dict__* or the result of calling *__getstate__()* is picklable (see section *The pickle protocol* for details).

Attempts to pickle unpicklable objects will raise the *PicklingError* exception; when this happens, an unspecified number of bytes may have already been written to the underlying file. Trying to pickle a highly recursive data structure may exceed the maximum recursion depth, a *RuntimeError* will be raised in this case. You can carefully raise this limit with *sys.setrecursionlimit()*.

Note that functions (built-in and user-defined) are pickled by "fully qualified" name reference, not by value. This means that only the function name is pickled, along with the name of the module the function is defined in. Neither the function's code, nor any of its function attributes are pickled. Thus the defining module must be importable in the unpickling environment, and the module must contain the named object, otherwise an exception will be raised.[4]

同样的，类也只打包名称，所以在解包环境中也有和函数相同的限制。注意，类体及其数据不会被打包，所以在下面的例子中类属性 `attr` 不会存在于解包后的环境中：

[4] 抛出的异常有可能是*ImportError*或*AttributeError*，也可能是其他异常。

```
class Foo:
    attr = 'a class attr'

picklestring = pickle.dumps(Foo)
```

这些限制决定了为什么必须在一个模块的顶层定义可打包的函数和类。

类似的，在打包类的实例时，其类体和类数据不会跟着实例一起被打包，只有实例数据会被打包。这样设计是有目的的，在将来修复类中的错误、给类增加方法之后，仍然可以载入原来版本类实例的打包数据来还原该实例。如果你准备长期使用一个对象，可能会同时存在较多版本的类体，可以为对象添加版本号，这样就可以通过类的*__setstate__()* 方法将老版本转换成新版本。

11.1.5 The pickle protocol

This section describes the "pickling protocol" that defines the interface between the pickler/unpickler and the objects that are being serialized. This protocol provides a standard way for you to define, customize, and control how your objects are serialized and de-serialized. The description in this section doesn't cover specific customizations that you can employ to make the unpickling environment slightly safer from untrusted pickle data streams; see section *Subclassing Unpicklers* for more details.

Pickling and unpickling normal class instances

`object.`**`__getinitargs__`**`()`

When a pickled class instance is unpickled, its `__init__()` method is normally *not* invoked. If it is desirable that the `__init__()` method be called on unpickling, an old-style class can define a method *__getinitargs__()*, which should return a *tuple* of positional arguments to be passed to the class constructor (`__init__()` for example). Keyword arguments are not supported. The *__getinitargs__()* method is called at pickle time; the tuple it returns is incorporated in the pickle for the instance.

`object.`**`__getnewargs__`**`()`

New-style types can provide a *__getnewargs__()* method that is used for protocol 2. Implementing this method is needed if the type establishes some internal invariants when the instance is created, or if the memory allocation is affected by the values passed to the `__new__()` method for the type (as it is for tuples and strings). Instances of a *new-style class* `C` are created using

```
obj = C.__new__(C, *args)
```

where *args* is the result of calling *__getnewargs__()* on the original object; if there is no *__getnewargs__()*, an empty tuple is assumed.

`object.`**`__getstate__`**`()`

Classes can further influence how their instances are pickled; if the class defines the method *__getstate__()*, it is called and the return state is pickled as the contents for the instance, instead of the contents of the instance's dictionary. If there is no *__getstate__()* method, the instance's *__dict__* is pickled.

`object.`**`__setstate__`**`(`*state*`)`

Upon unpickling, if the class also defines the method *__setstate__()*, it is called with the unpickled state.[5] If there is no *__setstate__()* method, the pickled state must be a dictionary and its items are assigned to the new instance's dictionary. If a class defines both *__getstate__()* and *__setstate__()*, the state object needn't be a dictionary and these methods can do what they want.[6]

[5] These methods can also be used to implement copying class instances.

[6] This protocol is also used by the shallow and deep copying operations defined in the *copy* module.

注解： For *new-style class*es, if *__getstate__()* returns a false value, the *__setstate__()* method will not be called.

注解： At unpickling time, some methods like `__getattr__()`, `__getattribute__()`, or `__setattr__()` may be called upon the instance. In case those methods rely on some internal invariant being true, the type should implement either *__getinitargs__()* or *__getnewargs__()* to establish such an invariant; otherwise, neither `__new__()` nor `__init__()` will be called.

Pickling and unpickling extension types

`object.`**`__reduce__`**`()`

When the `Pickler` encounters an object of a type it knows nothing about —such as an extension type —it looks in two places for a hint of how to pickle it. One alternative is for the object to implement a *__reduce__()* method. If provided, at pickling time *__reduce__()* will be called with no arguments, and it must return either a string or a tuple.

If a string is returned, it names a global variable whose contents are pickled as normal. The string returned by *__reduce__()* should be the object's local name relative to its module; the pickle module searches the module namespace to determine the object's module.

When a tuple is returned, it must be between two and five elements long. Optional elements can either be omitted, or `None` can be provided as their value. The contents of this tuple are pickled as normal and used to reconstruct the object at unpickling time. The semantics of each element are:

- A callable object that will be called to create the initial version of the object. The next element of the tuple will provide arguments for this callable, and later elements provide additional state information that will subsequently be used to fully reconstruct the pickled data.

  In the unpickling environment this object must be either a class, a callable registered as a "safe constructor" (see below), or it must have an attribute `__safe_for_unpickling__` with a true value. Otherwise, an `UnpicklingError` will be raised in the unpickling environment. Note that as usual, the callable itself is pickled by name.

- A tuple of arguments for the callable object.

  在 2.5 版更改: Formerly, this argument could also be `None`.

- Optionally, the object's state, which will be passed to the object's *__setstate__()* method as described in section *Pickling and unpickling normal class instances*. If the object has no *__setstate__()* method, then, as above, the value must be a dictionary and it will be added to the object's *__dict__*.

- Optionally, an iterator (and not a sequence) yielding successive list items. These list items will be pickled, and appended to the object using either `obj.append(item)` or `obj.extend(list_of_items)`. This is primarily used for list subclasses, but may be used by other classes as long as they have `append()` and `extend()` methods with the appropriate signature. (Whether `append()` or `extend()` is used depends on which pickle protocol version is used as well as the number of items to append, so both must be supported.)

- Optionally, an iterator (not a sequence) yielding successive dictionary items, which should be tuples of the form `(key, value)`. These items will be pickled and stored to the object using `obj[key] = value`. This is primarily used for dictionary subclasses, but may be used by other classes as long as they implement `__setitem__()`.

`object.`**`__reduce_ex__`**`(`*protocol*`)`

It is sometimes useful to know the protocol version when implementing *__reduce__()*. This can be done by implementing a method named *__reduce_ex__()* instead of *__reduce__()*. *__reduce_ex__()*, when

it exists, is called in preference over *__reduce__()* (you may still provide *__reduce__()* for backwards compatibility). The *__reduce_ex__()* method will be called with a single integer argument, the protocol version.

The *object* class implements both *__reduce__()* and *__reduce_ex__()*; however, if a subclass overrides *__reduce__()* but not *__reduce_ex__()*, the *__reduce_ex__()* implementation detects this and calls *__reduce__()*.

An alternative to implementing a *__reduce__()* method on the object to be pickled, is to register the callable with the *copy_reg* module. This module provides a way for programs to register "reduction functions" and constructors for user-defined types. Reduction functions have the same semantics and interface as the *__reduce__()* method described above, except that they are called with a single argument, the object to be pickled.

The registered constructor is deemed a "safe constructor" for purposes of unpickling as described above.

Pickling and unpickling external objects

For the benefit of object persistence, the *pickle* module supports the notion of a reference to an object outside the pickled data stream. Such objects are referenced by a "persistent id", which is just an arbitrary string of printable ASCII characters. The resolution of such names is not defined by the *pickle* module; it will delegate this resolution to user defined functions on the pickler and unpickler.[7]

To define external persistent id resolution, you need to set the `persistent_id` attribute of the pickler object and the `persistent_load` attribute of the unpickler object.

To pickle objects that have an external persistent id, the pickler must have a custom `persistent_id()` method that takes an object as an argument and returns either `None` or the persistent id for that object. When `None` is returned, the pickler simply pickles the object as normal. When a persistent id string is returned, the pickler will pickle that string, along with a marker so that the unpickler will recognize the string as a persistent id.

To unpickle external objects, the unpickler must have a custom `persistent_load()` function that takes a persistent id string and returns the referenced object.

Here's a silly example that *might* shed more light:

```
import pickle
from cStringIO import StringIO

src = StringIO()
p = pickle.Pickler(src)

def persistent_id(obj):
    if hasattr(obj, 'x'):
        return 'the value %d' % obj.x
    else:
        return None

p.persistent_id = persistent_id

class Integer:
    def __init__(self, x):
        self.x = x
    def __str__(self):
        return 'My name is integer %d' % self.x
```

(下页继续)

[7] The actual mechanism for associating these user defined functions is slightly different for *pickle* and *cPickle*. The description given here works the same for both implementations. Users of the *pickle* module could also use subclassing to effect the same results, overriding the `persistent_id()` and `persistent_load()` methods in the derived classes.

(续上页)

```
i = Integer(7)
print i
p.dump(i)

datastream = src.getvalue()
print repr(datastream)
dst = StringIO(datastream)

up = pickle.Unpickler(dst)

class FancyInteger(Integer):
    def __str__(self):
        return 'I am the integer %d' % self.x

def persistent_load(persid):
    if persid.startswith('the value '):
        value = int(persid.split()[2])
        return FancyInteger(value)
    else:
        raise pickle.UnpicklingError, 'Invalid persistent id'

up.persistent_load = persistent_load

j = up.load()
print j
```

In the *cPickle* module, the unpickler's `persistent_load` attribute can also be set to a Python list, in which case, when the unpickler reaches a persistent id, the persistent id string will simply be appended to this list. This functionality exists so that a pickle data stream can be "sniffed" for object references without actually instantiating all the objects in a pickle.[8] Setting `persistent_load` to a list is usually used in conjunction with the `noload()` method on the Unpickler.

11.1.6 Subclassing Unpicklers

By default, unpickling will import any class that it finds in the pickle data. You can control exactly what gets unpickled and what gets called by customizing your unpickler. Unfortunately, exactly how you do this is different depending on whether you're using *pickle* or *cPickle*.[9]

In the *pickle* module, you need to derive a subclass from `Unpickler`, overriding the `load_global()` method. `load_global()` should read two lines from the pickle data stream where the first line will the name of the module containing the class and the second line will be the name of the instance's class. It then looks up the class, possibly importing the module and digging out the attribute, then it appends what it finds to the unpickler's stack. Later on, this class will be assigned to the `__class__` attribute of an empty class, as a way of magically creating an instance without calling its class's `__init__()`. Your job (should you choose to accept it), would be to have `load_global()` push onto the unpickler's stack, a known safe version of any class you deem safe to unpickle. It is up to you to produce such a class. Or you could raise an error if you want to disallow all unpickling of instances. If this sounds like a hack, you're right. Refer to the source code to make this work.

Things are a little cleaner with *cPickle*, but not by much. To control what gets unpickled, you can set the unpickler's `find_global` attribute to a function or `None`. If it is `None` then any attempts to unpickle instances will raise an `UnpicklingError`. If it is a function, then it should accept a module name and a class name, and return the

[8] We'll leave you with the image of Guido and Jim sitting around sniffing pickles in their living rooms.

[9] A word of caution: the mechanisms described here use internal attributes and methods, which are subject to change in future versions of Python. We intend to someday provide a common interface for controlling this behavior, which will work in either *pickle* or *cPickle*.

corresponding class object. It is responsible for looking up the class and performing any necessary imports, and it may raise an error to prevent instances of the class from being unpickled.

The moral of the story is that you should be really careful about the source of the strings your application unpickles.

11.1.7 Example

For the simplest code, use the `dump()` and `load()` functions. Note that a self-referencing list is pickled and restored correctly.

```
import pickle

data1 = {'a': [1, 2.0, 3, 4+6j],
         'b': ('string', u'Unicode string'),
         'c': None}

selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.
pickle.dump(data1, output)

# Pickle the list using the highest protocol available.
pickle.dump(selfref_list, output, -1)

output.close()
```

The following example reads the resulting pickled data. When reading a pickle-containing file, you should open the file in binary mode because you can't be sure if the ASCII or binary format was used.

```
import pprint, pickle

pkl_file = open('data.pkl', 'rb')

data1 = pickle.load(pkl_file)
pprint.pprint(data1)

data2 = pickle.load(pkl_file)
pprint.pprint(data2)

pkl_file.close()
```

Here's a larger example that shows how to modify pickling behavior for a class. The `TextReader` class opens a text file, and returns the line number and line contents each time its `readline()` method is called. If a `TextReader` instance is pickled, all attributes *except* the file object member are saved. When the instance is unpickled, the file is reopened, and reading resumes from the last location. The *__setstate__()* and *__getstate__()* methods are used to implement this behavior.

```
#!/usr/local/bin/python

class TextReader:
    """Print and number lines in a text file."""
    def __init__(self, file):
        self.file = file
```

(下页继续)

(续上页)

```
        self.fh = open(file)
        self.lineno = 0

    def readline(self):
        self.lineno = self.lineno + 1
        line = self.fh.readline()
        if not line:
            return None
        if line.endswith("\n"):
            line = line[:-1]
        return "%d: %s" % (self.lineno, line)

    def __getstate__(self):
        odict = self.__dict__.copy() # copy the dict since we change it
        del odict['fh']              # remove filehandle entry
        return odict

    def __setstate__(self, dict):
        fh = open(dict['file'])      # reopen file
        count = dict['lineno']       # read from file...
        while count:                 # until line count is restored
            fh.readline()
            count = count - 1
        self.__dict__.update(dict)   # update attributes
        self.fh = fh                 # save the file object
```

使用方法如下所示：

```
>>> import TextReader
>>> obj = TextReader.TextReader("TextReader.py")
>>> obj.readline()
'1: #!/usr/local/bin/python'
>>> obj.readline()
'2: '
>>> obj.readline()
'3: class TextReader:'
>>> import pickle
>>> pickle.dump(obj, open('save.p', 'wb'))
```

If you want to see that *pickle* works across Python processes, start another Python session, before continuing. What follows can happen from either the same process or a new process.

```
>>> import pickle
>>> reader = pickle.load(open('save.p', 'rb'))
>>> reader.readline()
'4:     """Print and number lines in a text file."""'
```

参见：

Module *copy_reg* 为扩展类型提供 pickle 接口所需的构造函数。

模块*shelve* 带索引的数据库，用于存放对象，使用了*pickle*模块。

模块*copy* 浅层 (shallow) 和深层 (deep) 复制对象操作

模块*marshal* 高效地序列化内置类型的数据。

11.2 cPickle —A faster pickle

The *cPickle* module supports serialization and de-serialization of Python objects, providing an interface and functionality nearly identical to the *pickle* module. There are several differences, the most important being performance and subclassability.

First, *cPickle* can be up to 1000 times faster than *pickle* because the former is implemented in C. Second, in the *cPickle* module the callables `Pickler()` and `Unpickler()` are functions, not classes. This means that you cannot use them to derive custom pickling and unpickling subclasses. Most applications have no need for this functionality and should benefit from the greatly improved performance of the *cPickle* module.

The pickle data stream produced by *pickle* and *cPickle* are identical, so it is possible to use *pickle* and *cPickle* interchangeably with existing pickles.[10]

There are additional minor differences in API between *cPickle* and *pickle*, however for most applications, they are interchangeable. More documentation is provided in the *pickle* module documentation, which includes a list of the documented differences.

备注

11.3 copy_reg —Register pickle support functions

注解: The *copy_reg* module has been renamed to `copyreg` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

The *copy_reg* module offers a way to define functions used while pickling specific objects. The *pickle*, *cPickle*, and *copy* modules use those functions when pickling/copying those objects. The module provides configuration information about object constructors which are not classes. Such constructors may be factory functions or class instances.

`copy_reg.`**`constructor`**`(`*object*`)`
Declares *object* to be a valid constructor. If *object* is not callable (and hence not valid as a constructor), raises *TypeError*.

`copy_reg.`**`pickle`**`(`*type*, *function*[, *constructor*]`)`
Declares that *function* should be used as a "reduction" function for objects of type *type*; *type* must not be a "classic" class object. (Classic classes are handled differently; see the documentation for the *pickle* module for details.) *function* should return either a string or a tuple containing two or three elements.

The optional *constructor* parameter, if provided, is a callable object which can be used to reconstruct the object when called with the tuple of arguments returned by *function* at pickling time. *TypeError* will be raised if *object* is a class or *constructor* is not callable.

See the *pickle* module for more details on the interface expected of *function* and *constructor*.

[10] Since the pickle data format is actually a tiny stack-oriented programming language, and some freedom is taken in the encodings of certain objects, it is possible that the two modules produce different data streams for the same input objects. However it is guaranteed that they will always be able to read each other's data streams.

11.3.1 Example

The example below would like to show how to register a pickle function and how it will be used:

```
>>> import copy_reg, copy, pickle
>>> class C(object):
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copy_reg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```

11.4 shelve —Python 对象持久化

源代码: Lib/shelve.py

“shelf” 是一种持久化的类似字典的对象。与 “dbm” 数据库的区别在于 shelf 中的值（不是键！）实际上可以为任意 Python 对象—即*pickle*模块能够处理的任何东西。这包括大部分类实例、递归数据类型，以及包含大量共享子对象的对象。键则为普通的字符串。

shelve.**open**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

Open a persistent dictionary. The filename specified is the base filename for the underlying database. As a side-effect, an extension may be added to the filename and more than one file may be created. By default, the underlying database file is opened for reading and writing. The optional *flag* parameter has the same interpretation as the *flag* parameter of *anydbm.open()*.

By default, version 0 pickles are used to serialize values. The version of the pickle protocol can be specified with the *protocol* parameter.

在 2.3 版更改: The *protocol* parameter was added.

由于 Python 语义的限制，shelf 无法确定一个可变的持久化字典条目在何时被修改。默认情况下 只有在被修改对象再赋值给 shelf 时才会写入该对象 (参见示例)。如果可选的 *writeback* 形参设为 `True`，则所有被访问的条目都将在内存中被缓存，并会在*sync()* 和*close()* 时被写入；这可以使得对持久化字典中可变条目的修改更方便，但是如果访问的条目很多，这会消耗大量内存作为缓存，并会使得关闭操作变得非常缓慢，因为所有被访问的条目都需要写回到字典（无法确定被访问的条目中哪个是可变的，也无法确定哪个被实际修改了）。

Like file objects, shelve objects should be closed explicitly to ensure that the persistent data is flushed to disk.

警告: 由于*shelve*模块需要*pickle*的支持，因此从不可靠的来源载入 shelf 是不安全的。与 pickle 一样，载入 shelf 时可以执行任意代码。

Shelf objects support most of the methods supported by dictionaries. This eases the transition from dictionary based scripts to those requiring persistent storage.

Note, the Python 3 transition methods (*viewkeys()*, *viewvalues()*, and *viewitems()*) are not supported.

额外支持的两个方法：

Shelf.**sync**()
如果 shelf 打开时将 *writeback* 设为*True* 则写回缓存中的所有条目。如果可行还会清空缓存并将持久化字典同步到磁盘。此方法会在使用*close()* 关闭 shelf 时自动被调用。

Shelf.**close**()
同步并关闭持久化 *dict* 对象。对已关闭 shelf 的操作将失败并引发*ValueError*。

参见:

持久化字典方案，使用了广泛支持的存储格式并具有原生字典的速度。

11.4.1 限制

- The choice of which database package will be used (such as *dbm*, *gdbm* or *bsddb*) depends on which interface is available. Therefore it is not safe to open the database directly using *dbm*. The database is also (unfortunately) subject to the limitations of *dbm*, if it is used —this means that (the pickled representation of) the objects stored in the database should be fairly small, and in rare cases key collisions may cause the database to refuse updates.
- *shelve* 模块不支持对 shelve 对象的 并发读/写访问。（多个同时读取访问则是安全的。）当一个程序打开一个 shelve 对象来写入时，不应再有其他程序同时打开它来读取或写入。Unix 文件锁定可被用来解决此问题，但这在不同 Unix 版本上会存在差异，并且需要有关所用数据库实现的细节知识。

class shelve.**Shelf**(*dict*, *protocol=None*, *writeback=False*)
A subclass of *UserDict.DictMixin* which stores pickled values in the *dict* object.

By default, version 0 pickles are used to serialize values. The version of the pickle protocol can be specified with the *protocol* parameter. See the *pickle* documentation for a discussion of the pickle protocols.

在 2.3 版更改: The *protocol* parameter was added.

如果 *writeback* 形参为 True，对象将为所有访问过的条目保留缓存并在同步和关闭时将它们写回到 *dict*。这允许对可变的条目执行自然操作，但是会消耗更多内存并让同步和关闭花费更长时间。

class shelve.**BsdDbShelf**(*dict*, *protocol=None*, *writeback=False*)
A subclass of *Shelf* which exposes first(), next(), previous(), last() and set_location() which are available in the *bsddb* module but not in other database modules. The *dict* object passed to the constructor must support those methods. This is generally accomplished by calling one of *bsddb.hashopen()*, *bsddb.btopen()* or *bsddb.rnopen()*. The optional *protocol* and *writeback* parameters have the same interpretation as for the *Shelf* class.

class shelve.**DbfilenameShelf**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)
A subclass of *Shelf* which accepts a *filename* instead of a dict-like object. The underlying file will be opened using *anydbm.open()*. By default, the file will be created and opened for both read and write. The optional *flag* parameter has the same interpretation as for the *open()* function. The optional *protocol* and *writeback* parameters have the same interpretation as for the *Shelf* class.

11.4.2 示例

对接口的总结如下 (key 为字符串，data 为任意对象):

```
import shelve

d = shelve.open(filename) # open -- file may get suffix added by low-level
                          # library

d[key] = data   # store data at key (overwrites old data if
                # using an existing key)
data = d[key]   # retrieve a COPY of data at key (raise KeyError if no
                # such key)
del d[key]      # delete data stored at key (raises KeyError
                # if no such key)
flag = d.has_key(key)   # true if the key exists
klist = d.keys() # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = range(4)  # this works as expected, but...
d['xx'].append(5)   # *this doesn't!* -- d['xx'] is STILL range(4)!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']      # extracts the copy
temp.append(5)      # mutates the copy
d['xx'] = temp      # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()       # close it
```

参见:

Module *anydbm* Generic interface to dbm-style databases.

Module *bsddb* BSD db database interface.

Module *dbhash* Thin layer around the *bsddb* which provides an *open()* function like the other database modules.

模块*dbm* Standard Unix database interface.

Module *dumbdbm* Portable implementation of the dbm interface.

Module *gdbm* GNU database interface, based on the dbm interface.

模块*pickle* *shelve* 所使用的对象序列化。

Module *cPickle* High-performance version of *pickle*.

11.5 marshal —内部 Python 对象序列化

此模块包含一此能以二进制格式来读写 Python 值的函数。这种格式是 Python 专属的，但是独立于特定的机器架构（即你可以在一台 PC 上写入某个 Python 值，将文件传到一台 Sun 上并在那里读取它）。这种格式的细节有意不带文档说明；它可能在不同 Python 版本中发生改变（但这种情况极少发生）。[1]

这不是一个通用的“持久化”模块。对于通用的持久化以及通过 RPC 调用传递 Python 对象，请参阅*pickle* 和*shelve* 等模块。*marshal* 模块主要是为了支持读写 `.pyc` 文件形式“伪编译”代码的 Python 模块。因此，Python 维护者保留在必要时以不向下兼容的方式修改 marshal 格式的权利。如果你要序列化和反序列化 Python 对象，请改用*pickle* 模块–其执行效率相当，版本独立性有保证，并且 pickle 还支持比 marshal 更多样的对象类型。

警告： *marshal* 模块对于错误或恶意构建的数据来说是不安全的。永远不要 unmarshal 来自不受信任的或未经验证的来源的数据。

Not all Python object types are supported; in general, only objects whose value is independent from a particular invocation of Python can be written and read by this module. The following types are supported: booleans, integers, long integers, floating point numbers, complex numbers, strings, Unicode objects, tuples, lists, sets, frozensets, dictionaries, and code objects, where it should be understood that tuples, lists, sets, frozensets and dictionaries are only supported as long as the values contained therein are themselves supported; and recursive lists, sets and dictionaries should not be written (they will cause infinite loops). The singletons *None*, *Ellipsis* and *StopIteration* can also be marshalled and unmarshalled.

警告： On machines where C’s `long int` type has more than 32 bits (such as the DEC Alpha), it is possible to create plain Python integers that are longer than 32 bits. If such an integer is marshaled and read back in on a machine where C’s `long int` type has only 32 bits, a Python long integer object is returned instead. While of a different type, the numeric value is the same. (This behavior is new in Python 2.2. In earlier versions, all but the least-significant 32 bits of the value were lost, and a warning message was printed.)

There are functions that read/write files as well as functions operating on strings.

这个模块定义了以下函数：

`marshal.`**`dump`**(*value*, *file*[, *version*])

Write the value on the open file. The value must be a supported type. The file must be an open file object such as `sys.stdout` or returned by *open()* or *os.popen()*. It may not be a wrapper such as TemporaryFile on Windows. It must be opened in binary mode (`'wb'` or `'w+b'`).

如果值具有（或所包含的对象具有）不受支持的类型，则会引发*ValueError* —但是将向文件写入垃圾数据。对象也将不能正确地通过*load()* 重新读取。

2.4 新版功能: *version* 参数指明 `dump` 应当使用的数据格式（见下文）。

`marshal.`**`load`**(*file*)

Read one value from the open file and return it. If no valid value is read (e.g. because the data has a different Python version’s incompatible marshal format), raise *EOFError*, *ValueError* or *TypeError*. The file must be an open file object opened in binary mode (`'rb'` or `'r+b'`).

注解： 如果通过*dump()* marshal 了一个包含不受支持类型的对象，*load()* 将为不可 marshal 的类型替换 `None`。

[1] 此模块的名称来源于 Modula-3 (及其他语言) 的设计者所使用的术语，他们使用术语“marshal”来表示以自包含的形式传输数据。严格地说，将数据从内部形式转换为外部形式 (例如用于 RPC 缓冲区) 称为“marshal”而其逆过程则称为“unmarshal”。

marshal.**dumps**(*value*[, *version*])
Return the string that would be written to a file by dump(value, file). The value must be a supported type. Raise a *ValueError* exception if value has (or contains an object that has) an unsupported type.

2.4 新版功能: *version* 参数指明 dumps 应当使用的数据类型（见下文）。

marshal.**loads**(*string*)
Convert the string to a value. If no valid value is found, raise *EOFError*, *ValueError* or *TypeError*. Extra characters in the string are ignored.

此外，还定义了以下常量:

marshal.**version**
Indicates the format that the module uses. Version 0 is the historical format, version 1 (added in Python 2.4) shares interned strings and version 2 (added in Python 2.5) uses a binary format for floating point numbers. The current version is 2.

2.4 新版功能.

备注

11.6 anydbm —Generic access to DBM-style databases

注解: The *anydbm* module has been renamed to *dbm* in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

anydbm is a generic interface to variants of the DBM database —*dbhash* (requires *bsddb*), *gdbm*, or *dbm*. If none of these modules is installed, the slow-but-simple implementation in module *dumbdbm* will be used.

anydbm.**open**(*filename*[, *flag*[, *mode*]])
Open the database file *filename* and return a corresponding object.

If the database file already exists, the *whichdb* module is used to determine its type and the appropriate module is used; if it does not exist, the first module listed above that can be imported is used.

The optional *flag* argument must be one of these values:

Value	Meaning
'r'	Open existing database for reading only (default)
'w'	Open existing database for reading and writing
'c'	Open database for reading and writing, creating it if it doesn't exist
'n'	Always create a new, empty database, open for reading and writing

If not specified, the default value is 'r'.

The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0666 (and will be modified by the prevailing umask).

exception anydbm.**error**
A tuple containing the exceptions that can be raised by each of the supported modules, with a unique exception also named *anydbm.error* as the first item —the latter is used when *anydbm.error* is raised.

The object returned by *open()* supports most of the same functionality as dictionaries; keys and their corresponding values can be stored, retrieved, and deleted, and the has_key() and keys() methods are available. Keys and values must always be strings.

The following example records some hostnames and a corresponding title, and then prints out the contents of the database:

```
import anydbm

# Open database, creating it if necessary.
db = anydbm.open('cache', 'c')

# Record some values
db['www.python.org'] = 'Python Website'
db['www.cnn.com'] = 'Cable News Network'

# Loop through contents.  Other dictionary methods
# such as .keys(), .values() also work.
for k, v in db.iteritems():
    print k, '\t', v

# Storing a non-string key or value will raise an exception (most
# likely a TypeError).
db['www.yahoo.com'] = 4

# Close when done.
db.close()
```

In addition to the dictionary-like methods, `anydbm` objects provide the following method:

`anydbm.`**`close`**`()`
Close the `anydbm` database.

参见:

Module ***dbhash*** BSD `db` database interface.

Module ***dbm*** Standard Unix database interface.

Module ***dumbdbm*** Portable implementation of the `dbm` interface.

Module ***gdbm*** GNU database interface, based on the `dbm` interface.

Module ***shelve*** General object persistence built on top of the Python `dbm` interface.

Module ***whichdb*** Utility module used to determine the type of an existing database.

11.7 whichdb —Guess which DBM module created a database

注解: The *whichdb* module's only function has been put into the *dbm* module in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

The single function in this module attempts to guess which of the several simple database modules available—*dbm*, *gdbm*, or *dbhash*—should be used to open a given file.

`whichdb.`**`whichdb`**`(`*filename*`)`
Returns one of the following values: `None` if the file can't be opened because it's unreadable or doesn't exist; the empty string (`''`) if the file's format can't be guessed; or a string containing the required module name, such as `'dbm'` or `'gdbm'`.

11.8 dbm —Simple “database” interface

注解： The *dbm* module has been renamed to dbm.ndbm in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

The *dbm* module provides an interface to the Unix “(n)dbm” library. Dbm objects behave like mappings (dictionaries), except that keys and values are always strings. Printing a dbm object doesn’t print the keys and values, and the items() and values() methods are not supported.

This module can be used with the “classic” ndbm interface, the BSD DB compatibility interface, or the GNU GDBM compatibility interface. On Unix, the **configure** script will attempt to locate the appropriate header file to simplify building this module.

该模块定义以下内容：

exception dbm.**error**
Raised on dbm-specific errors, such as I/O errors. *KeyError* is raised for general mapping errors like specifying an incorrect key.

dbm.**library**
所使用的 ndbm 实现库的名称。

dbm.**open**(*filename*[, *flag*[, *mode*]])
Open a dbm database and return a dbm object. The *filename* argument is the name of the database file (without the .dir or .pag extensions; note that the BSD DB implementation of the interface will append the extension .db and only create one file).

可选的 *flag* 参数必须是下列值之一：

值	含义
'r'	以只读方式打开现有数据库（默认）
'w'	以读写方式打开现有数据库
'c'	以读写方式打开数据库，如果不存在则创建它
'n'	始终创建一个新的空数据库，以读写方式打开

The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0666 (and will be modified by the prevailing umask).

In addition to the dictionary-like methods, dbm objects provide the following method:

dbm.**close**()
Close the dbm database.

参见：

Module *anydbm* Generic interface to dbm-style databases.

Module *gdbm* Similar interface to the GNU GDBM library.

Module *whichdb* Utility module used to determine the type of an existing database.

11.9 gdbm —GNU's reinterpretation of dbm

注解: The *gdbm* module has been renamed to `dbm.gnu` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

This module is quite similar to the *dbm* module, but uses `gdbm` instead to provide some additional functionality. Please note that the file formats created by `gdbm` and `dbm` are incompatible.

The *gdbm* module provides an interface to the GNU DBM library. `gdbm` objects behave like mappings (dictionaries), except that keys and values are always strings. Printing a `gdbm` object doesn't print the keys and values, and the `items()` and `values()` methods are not supported.

The module defines the following constant and functions:

exception gdbm.**error**
Raised on `gdbm`-specific errors, such as I/O errors. *KeyError* is raised for general mapping errors like specifying an incorrect key.

gdbm.**open**(*filename*[, *flag*[, *mode*]])
Open a `gdbm` database and return a `gdbm` object. The *filename* argument is the name of the database file.

The optional *flag* argument can be:

Value	Meaning
`'r'`	Open existing database for reading only (default)
`'w'`	Open existing database for reading and writing
`'c'`	Open database for reading and writing, creating it if it doesn't exist
`'n'`	Always create a new, empty database, open for reading and writing

The following additional characters may be appended to the flag to control how the database is opened:

Value	Meaning
`'f'`	Open the database in fast mode. Writes to the database will not be synchronized.
`'s'`	Synchronized mode. This will cause changes to the database to be immediately written to the file.
`'u'`	Do not lock database.

Not all flags are valid for all versions of `gdbm`. The module constant `open_flags` is a string of supported flag characters. The exception *error* is raised if an invalid flag is specified.

The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal `0666`.

In addition to the dictionary-like methods, `gdbm` objects have the following methods:

gdbm.**firstkey**()
It's possible to loop over every key in the database using this method and the *nextkey()* method. The traversal is ordered by `gdbm`'s internal hash values, and won't be sorted by the key values. This method returns the starting key.

gdbm.**nextkey**(*key*)
Returns the key that follows *key* in the traversal. The following code prints every key in the database `db`, without having to create a list in memory that contains them all:

```
k = db.firstkey()
while k != None:
    print k
    k = db.nextkey(k)
```

gdbm.**reorganize**()
If you have carried out a lot of deletions and would like to shrink the space used by the gdbm file, this routine will reorganize the database. gdbm will not shorten the length of a database file except by using this reorganization; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

gdbm.**sync**()
When the database has been opened in fast mode, this method forces any unwritten data to be written to the disk.

gdbm.**close**()
Close the gdbm database.

参见:

Module *anydbm* Generic interface to dbm-style databases.

Module *whichdb* Utility module used to determine the type of an existing database.

11.10 dbhash —DBM-style interface to the BSD database library

2.6 版后已移除: The *dbhash* module has been removed in Python 3.

The *dbhash* module provides a function to open databases using the BSD db library. This module mirrors the interface of the other Python database modules that provide access to DBM-style databases. The *bsddb* module is required to use *dbhash*.

This module provides an exception and a function:

exception dbhash.**error**
Exception raised on database errors other than *KeyError*. It is a synonym for bsddb.error.

dbhash.**open**(*path*[, *flag*[, *mode*]])
Open a db database and return the database object. The *path* argument is the name of the database file.

The *flag* argument can be:

Value	Meaning
'r'	Open existing database for reading only (default)
'w'	Open existing database for reading and writing
'c'	Open database for reading and writing, creating it if it doesn't exist
'n'	Always create a new, empty database, open for reading and writing

For platforms on which the BSD db library supports locking, an 'l' can be appended to indicate that locking should be used.

The optional *mode* parameter is used to indicate the Unix permission bits that should be set if a new database must be created; this will be masked by the current umask value for the process.

参见:

Module *anydbm* Generic interface to dbm-style databases.

Module *bsddb* Lower-level interface to the BSD db library.

Module *whichdb* Utility module used to determine the type of an existing database.

11.10.1 Database Objects

The database objects returned by *open()* provide the methods common to all the DBM-style databases and mapping objects. The following methods are available in addition to the standard methods.

dbhash.**first**()
: It's possible to loop over every key/value pair in the database using this method and the `next()` method. The traversal is ordered by the databases internal hash values, and won't be sorted by the key values. This method returns the starting key.

dbhash.**last**()
: Return the last key/value pair in a database traversal. This may be used to begin a reverse-order traversal; see *previous()*.

dbhash.**next**()
: Returns the key next key/value pair in a database traversal. The following code prints every key in the database `db`, without having to create a list in memory that contains them all:

```
print db.first()
for i in xrange(1, len(db)):
    print db.next()
```

dbhash.**previous**()
: Returns the previous key/value pair in a forward-traversal of the database. In conjunction with *last()*, this may be used to implement a reverse-order traversal.

dbhash.**sync**()
: This method forces any unwritten data to be written to the disk.

11.11 bsddb —Interface to Berkeley DB library

2.6 版后已移除: The *bsddb* module has been removed in Python 3.

The *bsddb* module provides an interface to the Berkeley DB library. Users can create hash, btree or record based library files using the appropriate open call. Bsddb objects behave generally like dictionaries. Keys and values must be strings, however, so to use other objects as keys or to store other kinds of objects the user must serialize them somehow, typically using *marshal.dumps()* or *pickle.dumps()*.

The *bsddb* module requires a Berkeley DB library version from 4.0 thru 4.7.

参见:

http://www.jcea.es/programacion/pybsddb.htm The website with documentation for the `bsddb.db` Python Berkeley DB interface that closely mirrors the object oriented interface provided in Berkeley DB 4.x itself.

http://www.oracle.com/database/berkeley-db/ The Berkeley DB library.

A more modern DB, DBEnv and DBSequence object interface is available in the `bsddb.db` module which closely matches the Berkeley DB C API documented at the above URLs. Additional features provided by the `bsddb.db` API include fine tuning, transactions, logging, and multiprocess concurrent database access.

The following is a description of the legacy *bsddb* interface compatible with the old Python bsddb module. Starting in Python 2.5 this interface should be safe for multithreaded access. The `bsddb.db` API is recommended for threading users as it provides better control.

The *bsddb* module defines the following functions that create objects that access the appropriate type of Berkeley DB file. The first two arguments of each function are the same. For ease of portability, only the first two arguments should be used in most instances.

bsddb.**hashopen**(*filename*[, *flag*[, *mode*[, *pgsize*[, *ffactor*[, *nelem*[, *cachesize*[, *lorder*[, *hflags*]]]]]]]])
Open the hash format file named *filename*. Files never intended to be preserved on disk may be created by passing `None` as the *filename*. The optional *flag* identifies the mode used to open the file. It may be `'r'` (read only), `'w'` (read-write), `'c'` (read-write - create if necessary; the default) or `'n'` (read-write - truncate to zero length). The other arguments are rarely used and are just passed to the low-level `dbopen()` function. Consult the Berkeley DB documentation for their use and interpretation.

bsddb.**btopen**(*filename*[, *flag*[, *mode*[, *btflags*[, *cachesize*[, *maxkeypage*[, *minkeypage*[, *pgsize*[, *lorder*]]]]]]]])
Open the btree format file named *filename*. Files never intended to be preserved on disk may be created by passing `None` as the *filename*. The optional *flag* identifies the mode used to open the file. It may be `'r'` (read only), `'w'` (read-write), `'c'` (read-write - create if necessary; the default) or `'n'` (read-write - truncate to zero length). The other arguments are rarely used and are just passed to the low-level dbopen function. Consult the Berkeley DB documentation for their use and interpretation.

bsddb.**rnopen**(*filename*[, *flag*[, *mode*[, *rnflags*[, *cachesize*[, *pgsize*[, *lorder*[, *rlen*[, *delim*[, *source*[, *pad*]]]]]]]]])
Open a DB record format file named *filename*. Files never intended to be preserved on disk may be created by passing `None` as the *filename*. The optional *flag* identifies the mode used to open the file. It may be `'r'` (read only), `'w'` (read-write), `'c'` (read-write - create if necessary; the default) or `'n'` (read-write - truncate to zero length). The other arguments are rarely used and are just passed to the low-level dbopen function. Consult the Berkeley DB documentation for their use and interpretation.

注解: Beginning in 2.3 some Unix versions of Python may have a `bsddb185` module. This is present *only* to allow backwards compatibility with systems which ship with the old Berkeley DB 1.85 database library. The `bsddb185` module should never be used directly in new code. The module has been removed in Python 3. If you find you still need it look in PyPI.

参见:

Module ***dbhash*** DBM-style interface to the *bsddb*

11.11.1 Hash, BTree and Record Objects

Once instantiated, hash, btree and record objects support the same methods as dictionaries. In addition, they support the methods listed below.

在 2.3.1 版更改: Added dictionary methods.

bsddbobject.**close**()
Close the underlying file. The object can no longer be accessed. Since there is no open *open()* method for these objects, to open the file again a new *bsddb* module open function must be called.

bsddbobject.**keys**()
Return the list of keys contained in the DB file. The order of the list is unspecified and should not be relied on. In particular, the order of the list returned is different for different file formats.

bsddbobject.**has_key**(*key*)
Return `1` if the DB file contains the argument as a key.

bsddbobject.**set_location**(*key*)
Set the cursor to the item indicated by *key* and return a tuple containing the key and its value. For binary tree databases (opened using *btopen()*), if *key* does not actually exist in the database, the cursor will point to the next item in sorted order and return that key and value. For other databases, *KeyError* will be raised if *key* is not found in the database.

bsddbobject.**first**()
Set the cursor to the first item in the DB file and return it. The order of keys in the file is unspecified, except in the case of B-Tree databases. This method raises `bsddb.error` if the database is empty.

bsddbobject.**next**()
Set the cursor to the next item in the DB file and return it. The order of keys in the file is unspecified, except in the case of B-Tree databases.

bsddbobject.**previous**()
Set the cursor to the previous item in the DB file and return it. The order of keys in the file is unspecified, except in the case of B-Tree databases. This is not supported on hashtable databases (those opened with *hashopen()*).

bsddbobject.**last**()
Set the cursor to the last item in the DB file and return it. The order of keys in the file is unspecified. This is not supported on hashtable databases (those opened with *hashopen()*). This method raises `bsddb.error` if the database is empty.

bsddbobject.**sync**()
Synchronize the database on disk.

Example:

```
>>> import bsddb
>>> db = bsddb.btopen('spam.db', 'c')
>>> for i in range(10): db['%d'%i] = '%d'% (i*i)
...
>>> db['3']
'9'
>>> db.keys()
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> db.first()
('0', '0')
>>> db.next()
('1', '1')
>>> db.last()
('9', '81')
>>> db.set_location('2')
('2', '4')
>>> db.previous()
('1', '1')
>>> for k, v in db.iteritems():
...     print k, v
0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
>>> '8' in db
True
>>> db.sync()
0
```

11.12 dumbdbm —Portable DBM implementation

注解: The *dumbdbm* module has been renamed to dbm.dumb in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

注解: The *dumbdbm* module is intended as a last resort fallback for the *anydbm* module when no more robust module is available. The *dumbdbm* module is not written for speed and is not nearly as heavily used as the other database modules.

The *dumbdbm* module provides a persistent dictionary-like interface which is written entirely in Python. Unlike other modules such as *gdbm* and *bsddb*, no external library is required. As with other persistent mappings, the keys and values must always be strings.

The module defines the following:

exception dumbdbm.**error**
Raised on dumbdbm-specific errors, such as I/O errors. *KeyError* is raised for general mapping errors like specifying an incorrect key.

dumbdbm.**open**(*filename*[, *flag*[, *mode*]])
Open a dumbdbm database and return a dumbdbm object. The *filename* argument is the basename of the database file (without any specific extensions). When a dumbdbm database is created, files with .dat and .dir extensions are created.

The optional *flag* argument is currently ignored; the database is always opened for update, and will be created if it does not exist.

The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0666 (and will be modified by the prevailing umask).

在 2.2 版更改: The *mode* argument was ignored in earlier versions.

In addition to the dictionary-like methods, dumbdm objects provide the following method:

dumbdbm.**close**()
Close the dumbdm database.

参见:

Module *anydbm* Generic interface to dbm-style databases.

Module *dbm* Similar interface to the DBM/NDBM library.

Module *gdbm* Similar interface to the GNU GDBM library.

Module *shelve* Persistence module which stores non-string data.

Module *whichdb* Utility module used to determine the type of an existing database.

11.12.1 Dumbdbm Objects

In addition to the methods provided by the *UserDict.DictMixin* class, dumbdbm objects provide the following methods.

dumbdbm.**sync**()
Synchronize the on-disk directory and data files. This method is called by the *sync()* method of Shelve objects.

11.13 sqlite3 —SQLite 数据库 DB-API 2.0 接口模块

2.5 新版功能.

SQLite 是一个 C 语言库，它可以提供一种轻量级的基于磁盘的数据库，这种数据库不需要独立的服务器进程，也允许需要使用一种非标准的 SQL 查询语言来访问它。一些应用程序可以使用 SQLite 作为内部数据存储。可以用它来创建一个应用程序原型，然后再迁移到更大的数据库，比如 PostgreSQL 或 Oracle。

sqlite3 模块由 Gerhard Häring 编写。它提供了符合 DB-API 2.0 规范的接口，这个规范是 **PEP 249**。

要使用这个模块，必须先创建一个*Connection*对象，它代表数据库。下面例子中，数据将存储在 example.db 文件中：

```
import sqlite3
conn = sqlite3.connect('example.db')
```

你也可以使用 :memory: 来创建一个内存中的数据库

当有了*Connection* 对象后，你可以创建一个*Cursor* 游标对象，然后调用它的*execute()* 方法来执行 SQL 语句：

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
             (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

这些数据被持久化保存了，而且可以在之后的会话中使用它们：

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
```

通常你的 SQL 操作需要使用一些 Python 变量的值。你不应该使用 Python 的字符串操作来创建你的查询语句，因为那样做不安全；它会使你的程序容易受到 SQL 注入攻击（在 https://xkcd.com/327/ 上有一个搞笑的例子，看看有什么后果）

推荐另外一种方法：使用 DB-API 的参数替换。在你的 SQL 语句中，使用 ? 占位符来代替值，然后把对应的值组成的元组做为*execute()* 方法的第二个参数。（其他数据库可能会使用不同的占位符，比如 %s 或者

:1）例如：

```
# Never do this -- insecure!
symbol = 'RHAT'
c.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)

# Do this instead
t = ('RHAT',)
c.execute('SELECT * FROM stocks WHERE symbol=?', t)
print c.fetchone()

# Larger example that inserts many records at a time
purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
             ('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),
             ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
            ]
c.executemany('INSERT INTO stocks VALUES (?,?,?,?,?)', purchases)
```

要在执行 SELECT 语句后获取数据，你可以把游标作为*iterator*，然后调用它的`fetchone()`方法来获取一条匹配的行，也可以调用`fetchall()`来得到包含多个匹配行的列表。

下面是一个使用迭代器形式的例子：

```
>>> for row in c.execute('SELECT * FROM stocks ORDER BY price'):
        print row

(u'2006-01-05', u'BUY', u'RHAT', 100, 35.14)
(u'2006-03-28', u'BUY', u'IBM', 1000, 45.0)
(u'2006-04-06', u'SELL', u'IBM', 500, 53.0)
(u'2006-04-05', u'BUY', u'MSFT', 1000, 72.0)
```

参见：

https://github.com/ghaering/pysqlite pysqlite 的主页–sqlite3 在外部使用“pysqlite”名字进行开发。

https://www.sqlite.org SQLite 的主页；它的文档详细描述了它所支持的 SQL 方言的语法和可用的数据类型。

http://www.w3schools.com/sql/ 学习 SQL 语法的教程、参考和例子。

PEP 249 - DB-API 2.0 规范 Marc-André Lemburg 写的 PEP。

11.13.1 模块函数和常量

`sqlite3.`**`version`**
这个模块的版本号，是一个字符串。不是 SQLite 库的版本号。

`sqlite3.`**`version_info`**
这个模块的版本号，是一个由整数组成的元组。不是 SQLite 库的版本号。

`sqlite3.`**`sqlite_version`**
使用中的 SQLite 库的版本号，是一个字符串。

`sqlite3.`**`sqlite_version_info`**
使用中的 SQLite 库的版本号，是一个整数组成的元组。

`sqlite3.`**`PARSE_DECLTYPES`**
这个常量可以作为`connect()`函数的 *detect_types* 参数。

设置这个参数后，*sqlite3* 模块将解析它返回的每一列申明的类型。它会申明的类型的第一个单词，比如“integer primary key”，它会解析出“integer”，再比如“number(10)”，它会解析出“number”。然后，它会在转换器字典里查找那个类型注册的转换器函数，并调用它。

sqlite3.**PARSE_COLNAMES**
这个常量可以作为*connect()* 函数的 *detect_types* 参数。

Setting this makes the SQLite interface parse the column name for each column it returns. It will look for a string formed [mytype] in there, and then decide that ‘mytype’ is the type of the column. It will try to find an entry of ‘mytype’ in the converters dictionary and then use the converter function found there to return the value. The column name found in *Cursor.description* is only the first word of the column name, i. e. if you use something like `'as "x [datetime]"'` in your SQL, then we will parse out everything until the first blank for the column name: the column name would simply be “x”.

sqlite3.**connect**(*database*[, *timeout*, *detect_types*, *isolation_level*, *check_same_thread*, *factory*, *cached_statements*])
Opens a connection to the SQLite database file *database*. You can use `":memory:"` to open a database connection to a database that resides in RAM instead of on disk.

当一个数据库被多个连接访问的时候，如果其中一个进程修改这个数据库，在这个事务提交之前，这个 SQLite 数据库将会被一直锁定。*timeout* 参数指定了这个连接等待锁释放的超时时间，超时之后会引发一个异常。这个超时时间默认是 5.0（5 秒）。

For the *isolation_level* parameter, please see the *Connection.isolation_level* property of *Connection* objects.

SQLite 原生只支持 5 种类型：TEXT，INTEGER，REAL，BLOB 和 NULL。如果你想用其它类型，你必须自己添加相应的支持。使用 *detect_types* 参数和模块级别的*register_converter()* 函数注册 ** 转换器 ** 可以简单的实现。

detect_types 默认为 0（即关闭，没有类型检测）。你也可以组合*PARSE_DECLTYPES* 和*PARSE_COLNAMES* 来开启类型检测。

默认情况下，当调用 connect 方法的时候，*sqlite3* 模块使用了它的*Connection* 类。当然，你也可以创建*Connection* 类的子类，然后创建提供了 *factory* 参数的*connect()* 方法。

详情请查阅当前手册的*SQLite 与 Python 类型* 部分。

sqlite3 模块在内部使用语句缓存来避免 SQL 解析开销。如果要显式设置当前连接可以缓存的语句数，可以设置 *cached_statements* 参数。当前实现的默认值是缓存 100 条语句。

sqlite3.**register_converter**(*typename*, *callable*)
注册一个回调对象 *callable*，用来转换数据库中的字节串为自定的 Python 类型。所有类型为 *typename* 的数据库的值在转换时，都会调用这个回调对象。通过指定*connect()* 函数的 *detect-types* 参数来设置类型检测的方式。注意，*typename* 与查询语句中的类型名进行匹配时不区分大小写。

sqlite3.**register_adapter**(*type*, *callable*)
Registers a callable to convert the custom Python type *type* into one of SQLite’s supported types. The callable *callable* accepts as single parameter the Python value, and must return a value of the following types: int, long, float, str (UTF-8 encoded), unicode or buffer.

sqlite3.**complete_statement**(*sql*)
如果字符串 *sql* 包含一个或多个完整的 SQL 语句（以分号结束）则返回*True*。它不会验证 SQL 语法是否正确，仅会验证字符串字面上是否完整，以及是否以分号结束。

它可以用来构建一个 SQLite shell，下面是一个例子：

```
# A minimal SQLite shell for experiments

import sqlite3

```

(下页继续)

(续上页)

```
con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print "Enter your SQL commands to execute in sqlite3."
print "Enter a blank line to exit."

while True:
    line = raw_input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
            cur.execute(buffer)

            if buffer.lstrip().upper().startswith("SELECT"):
                print cur.fetchall()
        except sqlite3.Error as e:
            print "An error occurred:", e.args[0]
        buffer = ""

con.close()
```

sqlite3.**enable_callback_tracebacks**(*flag*)
默认情况下，您不会获得任何用户定义函数中的回溯消息，比如聚合，转换器，授权器回调等。如果要调试它们，可以设置 *flag* 参数为 `True` 并调用此函数。之后，回调中的回溯信息将会输出到 `sys.stderr`。再次使用*`False`* 来禁用该功能。

11.13.2 连接对象（Connection）

class sqlite3.**Connection**
SQLite 数据库连接对象有如下的属性和方法：

isolation_level
Get or set the current isolation level. *`None`* for autocommit mode or one of "DEFERRED", "IMMEDIATE" or "EXCLUSIVE". See section 控制事务 for a more detailed explanation.

cursor(*factory=Cursor*)
这个方法接受一个可选参数 *factory*，如果要指定这个参数，它必须是一个可调用对象，而且必须返回*`Cursor`* 类的一个实例或者子类。

commit()
这个方法提交当前事务。如果没有调用这个方法，那么从上一次提交 `commit()` 以来所有的变化在其他数据库连接上都是不可见的。如果你往数据库里写了数据，但是又查询不到，请检查是否忘记了调用这个方法。

rollback()
这个方法回滚从上一次调用*`commit()`* 以来所有数据库的改变。

close()
关闭数据库连接。注意，它不会自动调用*`commit()`* 方法。如果在关闭数据库连接之前没有调用*`commit()`*，那么你的修改将会丢失！

execute(*sql*[, *parameters*])
This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's `execute` method with the parameters given.

executemany(*sql*[, *parameters*])
This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's `executemany` method with the parameters given.

executescript(*sql_script*)
This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's `executescript` method with the parameters given.

create_function(*name*, *num_params*, *func*)
Creates a user-defined function that you can later use from within SQL statements under the function name *name*. *num_params* is the number of parameters the function accepts, and *func* is a Python callable that is called as the SQL function.

The function can return any of the types supported by SQLite: unicode, str, int, long, float, buffer and `None`.

示例:

```
import sqlite3
import md5

def md5sum(t):
    return md5.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", ("foo",))
print cur.fetchone()[0]
```

create_aggregate(*name*, *num_params*, *aggregate_class*)
创建一个自定义的聚合函数。

The aggregate class must implement a `step` method, which accepts the number of parameters *num_params*, and a `finalize` method which will return the final result of the aggregate.

The `finalize` method can return any of the types supported by SQLite: unicode, str, int, long, float, buffer and `None`.

示例:

```
import sqlite3

class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.cursor()
cur.execute("create table test(i)")
```

(下页继续)

(续上页)

```
cur.execute("insert into test(i) values (1)")
cur.execute("insert into test(i) values (2)")
cur.execute("select mysum(i) from test")
print cur.fetchone()[0]
```

create_collation(*name*, *callable*)

使用 *name* 和 *callable* 创建排序规则。这个 *callable* 接受两个字符串对象，如果第一个小于第二个则返回 -1，如果两个相等则返回 0，如果第一个大于第二个则返回 1。注意，这是用来控制排序的（SQL 中的 ORDER BY），所以它不会影响其它的 SQL 操作。

注意，这个 *callable* 可调用对象会把它的参数作为 Python 字节串，通常会以 UTF-8 编码格式对它进行编码。

以下示例显示了使用“错误方式”进行排序的自定义排序规则：

```
import sqlite3

def collate_reverse(string1, string2):
    return -cmp(string1, string2)

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.cursor()
cur.execute("create table test(x)")
cur.executemany("insert into test(x) values (?)", [("a",), ("b",)])
cur.execute("select x from test order by x collate reverse")
for row in cur:
    print row
con.close()
```

要移除一个排序规则，需要调用 `create_collation` 并设置 callable 参数为 `None`。

```
con.create_collation("reverse", None)
```

interrupt()

可以从不同的线程调用这个方法来终止所有查询操作，这些查询操作可能正在连接上执行。此方法调用之后，查询将会终止，而且查询的调用者会获得一个异常。

set_authorizer(*authorizer_callback*)

此方法注册一个授权回调对象。每次在访问数据库中某个表的某一列的时候，这个回调对象将会被调用。如果要允许访问，则返回 `SQLITE_OK`，如果要终止整个 SQL 语句，则返回 `SQLITE_DENY`，如果这一列需要当做 NULL 值处理，则返回 `SQLITE_IGNORE`。这些常量可以在*sqlite3* 模块中找到。

回调的第一个参数表示要授权的操作类型。第二个和第三个参数将是参数或*None*，具体取决于第一个参数的值。第 4 个参数是数据库的名称（“main”，“temp” 等），如果需要的话。第 5 个参数是负责访问尝试的最内层触发器或视图的名称，或者如果此访问尝试直接来自输入 SQL 代码，则为*None*。

请参阅 SQLite 文档，了解第一个参数的可能值以及第二个和第三个参数的含义，具体取决于第一个参数。所有必需的常量都可以在*sqlite3* 模块中找到。

set_progress_handler(*handler*, *n*)

此例程注册回调。对 SQLite 虚拟机的每个多指令调用回调。如果要在长时间运行的操作期间从 SQLite 调用（例如更新用户界面），这非常有用。

如果要清除以前安装的任何进度处理程序，调用该方法时请将 *handler* 参数设置为*None*。

2.6 新版功能.

enable_load_extension(*enabled*)
此例程允许/禁止 SQLite 引擎从共享库加载 SQLite 扩展。SQLite 扩展可以定义新功能，聚合或全新的虚拟表实现。一个众所周知的扩展是与 SQLite 一起分发的全文搜索扩展。

默认情况下禁用可加载扩展。见[1].

2.7 新版功能.

```
import sqlite3

con = sqlite3.connect(":memory:")

# enable extension loading
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")

# disable extension loading again
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("create virtual table recipe using fts3(name, ingredients)")
con.executescript("""
    insert into recipe (name, ingredients) values ('broccoli stew', 'broccoli␣
↪peppers cheese tomatoes');
    insert into recipe (name, ingredients) values ('pumpkin stew', 'pumpkin␣
↪onions garlic celery');
    insert into recipe (name, ingredients) values ('broccoli pie', 'broccoli␣
↪cheese onions flour');
    insert into recipe (name, ingredients) values ('pumpkin pie', 'pumpkin␣
↪sugar flour butter');
    """)
for row in con.execute("select rowid, name, ingredients from recipe where␣
↪name match 'pie'"):
    print row
```

load_extension(*path*)
此例程从共享库加载 SQLite 扩展。在使用此例程之前，必须使用*enable_load_extension()*启用扩展加载。

默认情况下禁用可加载扩展。见[1].

2.7 新版功能.

row_factory
您可以将此属性更改为可接受游标和原始行作为元组的可调用对象，并将返回实际结果行。这样，您可以实现更高级的返回结果的方法，例如返回一个可以按名称访问列的对象。

示例:

[1] The sqlite3 module is not built with loadable extension support by default, because some platforms (notably Mac OS X) have SQLite libraries which are compiled without this feature. To get loadable extension support, you must modify setup.py and remove the line that sets SQLITE_OMIT_LOAD_EXTENSION.

```
import sqlite3

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print cur.fetchone()["a"]
```

如果返回一个元组是不够的，并且你想要对列进行基于名称的访问，你应该考虑将*row_factory*设置为高度优化的*sqlite3.Row*类型。*Row*提供基于索引和不区分大小写的基于名称的访问，几乎没有内存开销。它可能比您自己的基于字典的自定义方法甚至基于 db_row 的解决方案更好。

text_factory

Using this attribute you can control what objects are returned for the `TEXT` data type. By default, this attribute is set to *unicode* and the *sqlite3* module will return Unicode objects for `TEXT`. If you want to return bytestrings instead, you can set it to *str*.

For efficiency reasons, there's also a way to return Unicode objects only for non-ASCII data, and bytestrings otherwise. To activate it, set this attribute to `sqlite3.OptimizedUnicode`.

您还可以将其设置为接受单个 bytestring 参数的任何其他可调用对象，并返回结果对象。

请参阅以下示例代码以进行说明：

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()

AUSTRIA = u"\xd6sterreich"

# by default, rows are returned as Unicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert row[0] == AUSTRIA

# but we can make sqlite3 always return bytestrings ...
con.text_factory = str
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is str
# the bytestrings will be encoded in UTF-8, unless you stored garbage in the
# database ...
assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory ...
# here we implement one that will ignore Unicode characters that cannot be
# decoded from UTF-8
con.text_factory = lambda x: unicode(x, "utf-8", "ignore")
cur.execute("select ?", ("this is latin1 and would normally create errors" +
                         u"\xe4\xf6\xfc".encode("latin1"),))
row = cur.fetchone()
```

(下页继续)

(续上页)

```
assert type(row[0]) is unicode

# sqlite3 offers a built-in optimized text_factory that will return bytestring
# objects, if the data is in ASCII only, and otherwise return unicode objects
con.text_factory = sqlite3.OptimizedUnicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is unicode

cur.execute("select ?", ("Germany",))
row = cur.fetchone()
assert type(row[0]) is str
```

total_changes
返回自打开数据库连接以来已修改，插入或删除的数据库行的总数。

iterdump
返回以 SQL 文本格式转储数据库的迭代器。保存内存数据库以便以后恢复时很有用。此函数提供与 **sqlite3** shell 中的 .dump 命令相同的功能。

2.6 新版功能.

示例:

```
# Convert file existing_db.db to SQL dump file dump.sql
import sqlite3, os

con = sqlite3.connect('existing_db.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
```

11.13.3 Cursor 对象

class sqlite3.**Cursor**
Cursor 游标实例具有以下属性和方法。

execute(*sql*[, *parameters*])
执行 SQL 语句。可以是参数化 SQL 语句（即，在 SQL 语句中使用占位符）。*sqlite3* 模块支持两种占位符：问号（qmark 风格）和命名占位符（命名风格）。

以下是两种风格的示例:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table people (name_last, age)")

who = "Yeltsin"
age = 72

# This is the qmark style:
cur.execute("insert into people values (?, ?)", (who, age))

# And this is the named style:
```

(下页继续)

(续上页)

```
cur.execute("select * from people where name_last=:who and age=:age", {"who":␣
↪who, "age": age})

print cur.fetchone()
```

execute() will only execute a single SQL statement. If you try to execute more than one statement with it, it will raise a Warning. Use *executescript()* if you want to execute multiple SQL statements with one call.

executemany(*sql*, *seq_of_parameters*)

Executes an SQL command against all parameter sequences or mappings found in the sequence *sql*. The *sqlite3* module also allows using an *iterator* yielding parameters instead of a sequence.

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self

    def next(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),) # this is a 1-tuple

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

cur.execute("select c from characters")
print cur.fetchall()
```

这是一个使用生成器*generator* 的简短示例：

```
import sqlite3
import string

def char_generator():
    for c in string.lowercase:
        yield (c,)

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

cur.executemany("insert into characters(c) values (?)", char_generator())

cur.execute("select c from characters")
print cur.fetchall()
```

executescript(*sql_script*)

这是一个非标准的便捷方法，可用于一次执行多条 SQL 语句。它会首先执行一条 COMMIT 语句，再执行以形参方式获取的 SQL 脚本。

sql_script can be a bytestring or a Unicode string.

示例:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
    create table person(
        firstname,
        lastname,
        age
    );

    create table book(
        title,
        author,
        published
    );

    insert into book(title, author, published)
    values (
        'Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',
        1987
    );
    """)
```

fetchone()

获取一个查询结果集的下一行，返回一个单独序列，或是在没有更多可用数据时返回*None*。

fetchmany([*size=cursor.arraysize*])

获取下一个多行查询结果集，返回一个列表。当没有更多可用行时将返回一个空列表。

每次调用获取的行数由 *size* 形参指定。如果没有给出该形参，则由 cursor 的 arraysize 决定要获取的行数。此方法将基于 size 形参值尝试获取指定数量的行。如果获取不到指定的行数，则可能返回较少的行。

请注意 *size* 形参会涉及到性能方面的考虑。为了获得优化的性能，通常最好是使用 arraysize 属性。如果使用 *size* 形参，则最好在从一个*fetchmany()* 调用到下一个调用之间保持相同的值。

fetchall()

获取一个查询结果的所有（剩余）行，返回一个列表。请注意 cursor 的 arraysize 属性会影响此操作的执行效率。当没有可用行时将返回一个空列表。

rowcount

虽然*sqlite3* 模块的*Cursor* 类实现了此属性，但数据库引擎本身对于确定“受影响行”/”已选择行”的支持并不完善。

对于*executemany()* 语句，修改行数会被汇总至*rowcount*。

根据 Python DB API 规格描述的要求，*rowcount* 属性“当未在 cursor 上执行 executeXX() 或者上一次操作的 rowcount 不是由接口确定时为 -1”。这包括 SELECT 语句，因为我们无法确定一次查询将产生的行计数，而要等获取了所有行时才会知道。。

在 SQLite 的 3.6.5 版之前，如果你执行 DELETE FROM table 时不附带任何条件，则*rowcount* 将被设为 0。

lastrowid
This read-only attribute provides the rowid of the last modified row. It is only set if you issued an INSERT statement using the *execute()* method. For operations other than INSERT or when *executemany()* is called, *lastrowid* is set to *None*.

description
这个只读属性将提供上一次查询的列名称。为了与 Python DB API 保持兼容，它会为每个列返回一个 7 元组，每个元组的最后六个条目均为*None*。

对于没有任何匹配行的 SELECT 语句同样会设置该属性。

connection
这个只读属性将提供*Cursor* 对象所使用的 SQLite 数据库*Connection*。通过调用*con.cursor()* 创建的*Cursor* 对象所包含的*connection* 属性将指向 *con*:

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
```

11.13.4 行对象 *Row*

class sqlite3.**Row**
一个*Row* 实例，该实例将作为用于*Connection* 对象的高度优化的*row_factory*。它的大部分行为都会模仿元组的特性。

它支持使用列名称的映射访问以及索引、迭代、文本表示、相等检测和*len()* 等操作。

如果两个*Row* 对象具有完全相同的列并且其成员均相等，则它们的比较结果为相等。

在 2.6 版更改: Added iteration and equality (hashability).

keys()
此方法会在一次查询之后立即返回一个列名称的列表，它是*Cursor.description* 中每个元组的第一个成员。

2.6 新版功能.

让我们假设我们如上面的例子所示初始化一个表:

```
conn = sqlite3.connect(":memory:")
c = conn.cursor()
c.execute('''create table stocks
(date text, trans text, symbol text,
 qty real, price real)''')
c.execute("""insert into stocks
          values ('2006-01-05','BUY','RHAT',100,35.14)""")
conn.commit()
c.close()
```

现在我们将*Row* 插入:

```
>>> conn.row_factory = sqlite3.Row
>>> c = conn.cursor()
>>> c.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = c.fetchone()
>>> type(r)
```

(下页继续)

(续上页)

```
<type 'sqlite3.Row'>
>>> r
(u'2006-01-05', u'BUY', u'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
u'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
...     print member
...
2006-01-05
BUY
RHAT
100.0
35.14
```

11.13.5 SQLite 与 Python 类型

概述

SQLite 原生支持如下的类型：NULL，INTEGER，REAL，TEXT，BLOB。

因此可以将以下 Python 类型发送到 SQLite 而不会出现任何问题：

Python 类型	SQLite 类型
None	NULL
int	INTEGER
long	INTEGER
float	REAL
str (UTF8-encoded)	TEXT
unicode	TEXT
buffer	BLOB

这是 SQLite 类型默认转换为 Python 类型的方式：

SQLite 类型	Python 类型
NULL	*None*
INTEGER	*int* or *long*, depending on size
REAL	*float*
TEXT	depends on *text_factory*, *unicode* by default
BLOB	*buffer*

sqlite3 模块的类型系统可通过两种方式来扩展：你可以通过对象适配将额外的 Python 类型保存在 SQLite 数据库中，你也可以让*sqlite3* 模块通过转换器将 SQLite 类型转换为不同的 Python 类型。

使用适配器将额外的 Python 类型保存在 SQLite 数据库中。

As described before, SQLite supports only a limited set of types natively. To use other Python types with SQLite, you must **adapt** them to one of the sqlite3 module's supported types for SQLite: one of NoneType, int, long, float, str, unicode, buffer.

有两种方式能让*sqlite3*模块将某个定制的 Python 类型适配为受支持的类型。

让对象自行调整

如果自己编写类，这是一种很好的方法。假设有这样的类::

```
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y
```

现在你想将这种点对象保存在一个 SQLite 列中。首先你必须选择一种受支持的类型用来表示点对象。让我们就用 str 并使用一个分号来分隔坐标值。然后你需要给你的类加一个方法 `__conform__(self, protocol)`，它必须返回转换后的值。形参 *protocol* 将为 `PrepareProtocol`。

```
import sqlite3

class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "%f;%f" % (self.x, self.y)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print cur.fetchone()[0]
```

注册可调用的适配器

另一种可能的做法是创建一个将该类型转换为字符串表示的函数并使用*register_adapter()*注册该函数。

注解： The type/class to adapt must be a *new-style class*, i. e. it must have *object* as one of its bases.

```
import sqlite3

class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)
```

(下页继续)

(续上页)

```
sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print cur.fetchone()[0]
```

sqlite3 模块有两个适配器可用于 Python 的内置*datetime.date* 和*datetime.datetime* 类型。现在假设我们想要存储*datetime.datetime* 对象，但不是表示为 ISO 格式，而是表示为 Unix 时间戳。

```
import sqlite3
import datetime, time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())

sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(":memory:")
cur = con.cursor()

now = datetime.datetime.now()
cur.execute("select ?", (now,))
print cur.fetchone()[0]
```

将 SQLite 值转换为自定义 Python 类型

编写适配器让你可以将定制的 Python 类型发送给 SQLite。但要令它真正有用，我们需要实现从 Python 到 SQLite 再回到 Python 的双向转换。

输入转换器。

让我们回到 `Point` 类。我们以字符串形式在 SQLite 中存储了 x 和 y 坐标值。

首先，我们将定义一个转换器函数，它接受这样的字符串作为形参并根据该参数构造一个 `Point` 对象。

注解： Converter functions **always** get called with a string, no matter under which data type you sent the value to SQLite.

```
def convert_point(s):
    x, y = map(float, s.split(";"))
    return Point(x, y)
```

现在你需要让*sqlite3* 模块知道你从数据库中选取的其实是一个点对象。有两种方式都可以做到这件事：

- 隐式的声明类型
- 显式的通过列名

这两种方式会在模块函数和常量 一节中描述，相应条目为*PARSE_DECLTYPES* 和*PARSE_COLNAMES* 常量。

下面的示例说明了这两种方法。

```
import sqlite3

class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "(%f;%f)" % (self.x, self.y)

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

def convert_point(s):
    x, y = map(float, s.split(";"))
    return Point(x, y)

# Register the adapter
sqlite3.register_adapter(Point, adapt_point)

# Register the converter
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

#########################
# 1) Using declared types
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p point)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute("select p from test")
print "with declared types:", cur.fetchone()[0]
cur.close()
con.close()

######################
# 1) Using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(p)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute('select p as "p [point]" from test')
print "with column names:", cur.fetchone()[0]
cur.close()
con.close()
```

默认适配器和转换器

对于 datetime 模块中的 date 和 datetime 类型已提供了默认的适配器。它们将会以 ISO 日期/ISO 时间戳的形式发给 SQLite。

默认转换器使用的注册名称是针对*datetime.date*的“date”和针对*datetime.datetime*的“timestamp”。

通过这种方式，你可以在大多数情况下使用 Python 的 date/timestamp 对象而无须任何额外处理。适配器的格式还与实验性的 SQLite date/time 函数兼容。

下面的示例演示了这一点。

```
import sqlite3
import datetime

con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_
↪COLNAMES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()
now = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print today, "=>", row[0], type(row[0])
print now, "=>", row[1], type(row[1])

cur.execute('select current_date as "d [date]", current_timestamp as "ts [timestamp]"
↪')
row = cur.fetchone()
print "current_date", row[0], type(row[0])
print "current_timestamp", row[1], type(row[1])
```

如果存储在 SQLite 中的时间戳的小数位多于 6 个数字，则时间戳转换器会将该值截断至微秒精度。

11.13.6 控制事务

By default, the *sqlite3* module opens transactions implicitly before a Data Modification Language (DML) statement (i.e. INSERT/UPDATE/DELETE/REPLACE), and commits transactions implicitly before a non-DML, non-query statement (i. e. anything other than SELECT or the aforementioned).

So if you are within a transaction and issue a command like CREATE TABLE ..., VACUUM, PRAGMA, the *sqlite3* module will commit implicitly before executing that command. There are two reasons for doing that. The first is that some of these commands don’t work within transactions. The other reason is that sqlite3 needs to keep track of the transaction state (if a transaction is active or not).

You can control which kind of BEGIN statements sqlite3 implicitly executes (or none at all) via the *isolation_level* parameter to the *connect()* call, or via the isolation_level property of connections.

If you want **autocommit mode**, then set isolation_level to None.

Otherwise leave it at its default, which will result in a plain “BEGIN” statement, or set it to one of SQLite’s supported isolation levels: “DEFERRED”, “IMMEDIATE” or “EXCLUSIVE”.

11.13.7 有效使用 sqlite3

使用快捷方式

使用*Connection*对象的非标准 execute(),executemany() 和 executescript() 方法，可以更简洁地编写代码，因为不必显式创建（通常是多余的）*Cursor*对象。相反，*Cursor*对象是隐式创建的，这些快捷方法返回游标对象。这样，只需对*Connection*对象调用一次，就能直接执行 SELECT 语句并遍历对象。

```
import sqlite3

persons = [
    ("Hugo", "Boss"),
    ("Calvin", "Klein")
    ]

con = sqlite3.connect(":memory:")

# Create the table
con.execute("create table person(firstname, lastname)")

# Fill the table
con.executemany("insert into person(firstname, lastname) values (?, ?)", persons)

# Print the table contents
for row in con.execute("select firstname, lastname from person"):
    print row

print "I just deleted", con.execute("delete from person").rowcount, "rows"
```

通过名称而不是索引访问索引

*sqlite3*模块的一个有用功能是内置的*sqlite3.Row*类，它被设计用作行对象的工厂。

该类的行装饰器可以用索引（如元组）和不区分大小写的名称访问：

```
import sqlite3

con = sqlite3.connect(":memory:")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select 'John' as name, 42 as age")
for row in cur:
    assert row[0] == row["name"]
    assert row["name"] == row["nAmE"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]
```

使用连接作为上下文管理器

2.6 新版功能.

连接对象可以用来作为上下文管理器，它可以自动提交或者回滚事务。如果出现异常，事务会被回滚；否则，事务会被提交。

```
import sqlite3

con = sqlite3.connect(":memory:")
con.execute("create table person (id integer primary key, firstname varchar unique)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("insert into person(firstname) values (?)", ("Joe",))

# con.rollback() is called after the with block finishes with an exception, the
# exception is still raised and must be caught
try:
    with con:
        con.execute("insert into person(firstname) values (?)", ("Joe",))
except sqlite3.IntegrityError:
    print "couldn't add Joe twice"
```

11.13.8 常见问题

多线程

较老版本的 SQLite 在共享线程之间存在连接问题。这就是 Python 模块不允许线程之间共享连接和游标的原因。如果仍然尝试这样做，则在运行时会出现异常。

唯一的例外是调用*interrupt()*方法，该方法仅在从其他线程进行调用时才有意义。

备注

CHAPTER 12

数据压缩和存档

The modules described in this chapter support data compression with the zlib, gzip, and bzip2 algorithms, and the creation of ZIP- and tar-format archives. See also 归档操作 provided by the *shutil* module.

12.1 zlib —与 gzip 兼容的压缩

此模块为需要数据压缩的程序提供了一系列函数，用于压缩和解压缩。这些函数使用了 zlib 库。zlib 库的项目主页是 http://www.zlib.net. 版本低于 1.1.3 的 zlib 与此 Python 模块之间存在已知的不兼容。1.1.3 版本的 zlib 存在一个安全漏洞，我们推荐使用 1.1.4 或更新的版本。

zlib 的函数有很多选项，一般需要按特定顺序使用。本文档没有覆盖全部的用法。更多详细信息请于 http://www.zlib.net/manual.html 参阅官方手册。

要读写 `.gz` 格式的文件，请参考*gzip* 模块。

此模块中可用的异常和函数如下：

exception zlib.**error**
在压缩或解压缩过程中发生错误时的异常。

zlib.**adler32**(*data*[, *value*])
Computes an Adler-32 checksum of *data*. (An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much more quickly.) If *value* is present, it is used as the starting value of the checksum; otherwise, a fixed default value is used. This allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

This function always returns an integer object.

注解： To generate the same numeric value across all Python versions and platforms use adler32(data) & 0xffffffff. If you are only using the checksum in packed binary format this is not necessary as the return value is the correct 32bit binary representation regardless of sign.

在 2.6 版更改: The return value is in the range [-2**31, 2**31-1] regardless of platform. In older versions the value is signed on some platforms and unsigned on others.

在 3.0 版更改: The return value is unsigned and in the range [0, 2**32-1] regardless of platform.

zlib.**compress**(*string*[, *level*])
Compresses the data in *string*, returning a string contained compressed data. *level* is an integer from 0 to 9 controlling the level of compression; 1 is fastest and produces the least compression, 9 is slowest and produces the most. 0 is no compression. The default value is 6. Raises the *error* exception if any error occurs.

zlib.**compressobj**([*level*[, *method*[, *wbits*[, *memlevel*[, *strategy*]]]]])
Returns a compression object, to be used for compressing data streams that won't fit into memory at once. *level* is an integer from 0 to 9 or −1, controlling the level of compression; 1 is fastest and produces the least compression, 9 is slowest and produces the most. 0 is no compression. The default value is −1 (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION represents a default compromise between speed and compression (currently equivalent to level 6).

method is the compression algorithm. Currently, the only supported value is DEFLATED.

The *wbits* argument controls the size of the history buffer (or the "window size") used when compressing data, and whether a header and trailer is included in the output. It can take several ranges of values. The default is 15.

- +9 到 +15：窗口大小以 2 为底的对数。即这些值对应着 512 到 32768 的窗口大小。更大的值会提供更好的压缩，同时内存开销也会更大。压缩输出会包含 zlib 特定格式的头部和尾部。
- −9 到 −15：绝对值为窗口大小以 2 为底的对数。压缩输出仅包含压缩数据，没有头部和尾部。
- +25 到 +31 = 16 + (9 到 15)：后 4 个比特位为窗口大小以 2 为底的对数。压缩输出包含一个基本的 **gzip** 头部，并以校验和为尾部。

memlevel controls the amount of memory used for internal compression state. Valid values range from 1 to 9. Higher values using more memory, but are faster and produce smaller output. The default is 8.

strategy is used to tune the compression algorithm. Possible values are Z_DEFAULT_STRATEGY, Z_FILTERED, and Z_HUFFMAN_ONLY. The default is Z_DEFAULT_STRATEGY.

zlib.**crc32**(*data*[, *value*])
Computes a CRC (Cyclic Redundancy Check) checksum of *data*. If *value* is present, it is used as the starting value of the checksum; otherwise, a fixed default value is used. This allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

This function always returns an integer object.

注解： To generate the same numeric value across all Python versions and platforms use crc32(data) & 0xffffffff. If you are only using the checksum in packed binary format this is not necessary as the return value is the correct 32bit binary representation regardless of sign.

在 2.6 版更改: The return value is in the range [-2**31, 2**31-1] regardless of platform. In older versions the value would be signed on some platforms and unsigned on others.

在 3.0 版更改: The return value is unsigned and in the range [0, 2**32-1] regardless of platform.

zlib.**decompress**(*string*[, *wbits*[, *bufsize*]])
Decompresses the data in *string*, returning a string containing the uncompressed data. The *wbits* parameter depends on the format of *string*, and is discussed further below. If *bufsize* is given, it is used as the initial size of the output buffer. Raises the *error* exception if any error occurs.

wbits 形参控制历史缓冲区（或称“窗口尺寸”）的大小以及所期望的头部和尾部格式。它类似于*compressobj()* 的形参，但可接受更大范围的值：

- +8 至 +15：窗口尺寸以二为底的对数。输入必须包含 zlib 头部和尾部。
- 0：根据 zlib 头部自动确定窗口大小。只从 zlib 1.2.3.5 版起受支持。
- −8 to −15：使用 *wbits* 的绝对值作为窗口大小以二为底的对数。输入必须为原始数据流，没有头部和尾部。
- +24 至 +31 = 16 + (8 至 15)：使用后 4 个比特位作为窗口大小以二为底的对数。输入必须包括 gzip 头部和尾部。
- +40 至 +47 = 32 + (8 至 15)：使用后 4 个比特位作为窗口大小以二为底的对数，并且自动接受 zlib 或 gzip 格式。

When decompressing a stream, the window size must not be smaller than the size originally used to compress the stream; using a too-small value may result in an *error* exception. The default *wbits* value is 15, which corresponds to the largest window size and requires a zlib header and trailer to be included.

bufsize is the initial size of the buffer used to hold decompressed data. If more space is required, the buffer size will be increased as needed, so you don't have to get this value exactly right; tuning it will only save a few calls to `malloc()`. The default size is 16384.

`zlib.`**`decompressobj`**`(`[*wbits*]`)`
返回一个解压对象，用来解压无法被一次性放入内存的数据流。

wbits 形参控制历史缓冲区的大小（或称“窗口大小”）以及所期望的头部和尾部格式。它的含义与*对 decompress() 的描述* 相同。

压缩对象支持以下方法：

`Compress.`**`compress`**`(`*string*`)`
Compress *string*, returning a string containing compressed data for at least part of the data in *string*. This data should be concatenated to the output produced by any preceding calls to the *compress()* method. Some input may be kept in internal buffers for later processing.

`Compress.`**`flush`**`(`[*mode*]`)`
All pending input is processed, and a string containing the remaining compressed output is returned. *mode* can be selected from the constants `Z_SYNC_FLUSH`, `Z_FULL_FLUSH`, or `Z_FINISH`, defaulting to `Z_FINISH`. `Z_SYNC_FLUSH` and `Z_FULL_FLUSH` allow compressing further strings of data, while `Z_FINISH` finishes the compressed stream and prevents compressing any more data. After calling *flush()* with *mode* set to `Z_FINISH`, the *compress()* method cannot be called again; the only realistic action is to delete the object.

`Compress.`**`copy`**`()`
返回此压缩对象的一个拷贝。它可以用来高效压缩一系列拥有相同前缀的数据。

2.5 新版功能.

Decompression objects support the following methods, and two attributes:

`Decompress.`**`unused_data`**
A string which contains any bytes past the end of the compressed data. That is, this remains `""` until the last byte that contains compression data is available. If the whole string turned out to contain compressed data, this is `""`, the empty string.

The only way to determine where a string of compressed data ends is by actually decompressing it. This means that when compressed data is contained part of a larger file, you can only find the end of it by reading data and feeding it followed by some non-empty string into a decompression object's *decompress()* method until the *unused_data* attribute is no longer the empty string.

`Decompress.`**`unconsumed_tail`**
A string that contains any data that was not consumed by the last *decompress()* call because it exceeded the

limit for the uncompressed data buffer. This data has not yet been seen by the zlib machinery, so you must feed it (possibly with further data concatenated to it) back to a subsequent `decompress()` method call in order to get correct output.

`Decompress.`**`decompress`**(*string*[, *max_length*])
Decompress *string*, returning a string containing the uncompressed data corresponding to at least part of the data in *string*. This data should be concatenated to the output produced by any preceding calls to the `decompress()` method. Some of the input data may be preserved in internal buffers for later processing.

If the optional parameter *max_length* is non-zero then the return value will be no longer than *max_length*. This may mean that not all of the compressed input can be processed; and unconsumed data will be stored in the attribute `unconsumed_tail`. This string must be passed to a subsequent call to `decompress()` if decompression is to continue. If *max_length* is not supplied then the whole input is decompressed, and `unconsumed_tail` is an empty string.

`Decompress.`**`flush`**([*length*])
All pending input is processed, and a string containing the remaining uncompressed output is returned. After calling `flush()`, the `decompress()` method cannot be called again; the only realistic action is to delete the object.

可选的形参 *length* 设置输出缓冲区的初始大小。

`Decompress.`**`copy`**()
返回解压缩对象的一个拷贝。它可以用来在数据流的中途保存解压缩器的状态以便加快随机查找数据流后续位置的速度。

2.5 新版功能.

参见:

模块*gzip* 读写 **gzip** 格式的文件。

http://www.zlib.net zlib 库项目主页。

http://www.zlib.net/manual.html zlib 库用户手册。提供了库的许多功能的解释和用法。

12.2 gzip —对 gzip 格式的支持

源代码: Lib/gzip.py

此模块提供的简单接口帮助用户压缩和解压缩文件，功能类似于 GNU 应用程序 **gzip** 和 **gunzip**。

数据压缩由`zlib`模块提供。

The `gzip` module provides the `GzipFile` class which is modeled after Python’s File Object. The `GzipFile` class reads and writes **gzip**-format files, automatically compressing or decompressing the data so that it looks like an ordinary file object.

注意，此模块不支持部分可以被 **gzip** 和 **gunzip** 解压的格式，如利用 **compress** 或 **pack** 压缩所得的文件。

这个模块定义了以下内容:

class `gzip.`**`GzipFile`**([*filename*[, *mode*[, *compresslevel*[, *fileobj*[, *mtime*]]]]])
Constructor for the `GzipFile` class, which simulates most of the methods of a file object, with the exception of the `readinto()` and `truncate()` methods. At least one of *fileobj* and *filename* must be given a non-trivial value.

The new class instance is based on *fileobj*, which can be a regular file, a `StringIO` object, or any other object which simulates a file. It defaults to `None`, in which case *filename* is opened to provide a file object.

当 *fileobj* 为 None 时，*filename* 参数只用于 **gzip** 文件头中，这个文件有可能包含未压缩文件的源文件名。如果文件可以被识别，默认 *fileobj* 的文件名；否则默认为空字符串，在这种情况下文件头将不包含源文件名。

The *mode* argument can be any of 'r', 'rb', 'a', 'ab', 'w', or 'wb', depending on whether the file will be read or written. The default is the mode of *fileobj* if discernible; otherwise, the default is 'rb'. If not given, the 'b' flag will be added to the mode to ensure the file is opened in binary mode for cross-platform portability.

compresslevel 参数是一个从 0 to 9 的整数，用于控制压缩等级；1 最快但压缩比例最小，9 最慢但压缩比例最大。0 不压缩。默认为 9。

The *mtime* argument is an optional numeric timestamp to be written to the stream when compressing. All **gzip** compressed streams are required to contain a timestamp. If omitted or None, the current time is used. This module ignores the timestamp when decompressing; however, some programs, such as **gunzip**, make use of it. The format of the timestamp is the same as that of the return value of time.time() and of the st_mtime attribute of the object returned by os.stat().

Calling a *GzipFile* object's close() method does not close *fileobj*, since you might wish to append more material after the compressed data. This also allows you to pass a *StringIO* object opened for writing as *fileobj*, and retrieve the resulting memory buffer using the *StringIO* object's *getvalue()* method.

GzipFile supports iteration and the with statement.

在 2.7 版更改: Support for the with statement was added.

在 2.7 版更改: Support for zero-padded files was added.

2.7 新版功能: The *mtime* argument.

gzip.**open**(*filename*[, *mode*[, *compresslevel*]])

This is a shorthand for GzipFile(filename, mode, compresslevel). The *filename* argument is required; *mode* defaults to 'rb' and *compresslevel* defaults to 9.

12.2.1 用法示例

读取压缩文件示例：

```
import gzip
with gzip.open('file.txt.gz', 'rb') as f:
    file_content = f.read()
```

创建 GZIP 文件示例：

```
import gzip
content = "Lots of content here"
with gzip.open('file.txt.gz', 'wb') as f:
    f.write(content)
```

使用 GZIP 压缩已有的文件示例：

```
import gzip
import shutil
with open('file.txt', 'rb') as f_in, gzip.open('file.txt.gz', 'wb') as f_out:
    shutil.copyfileobj(f_in, f_out)
```

参见:

模块*zlib* 支持 **gzip** 格式所需要的基本压缩模块。

12.3 bz2 —Compression compatible with bzip2

2.3 新版功能.

This module provides a comprehensive interface for the bz2 compression library. It implements a complete file interface, one-shot (de)compression functions, and types for sequential (de)compression.

Here is a summary of the features offered by the bz2 module:

- *BZ2File* class implements a complete file interface, including *readline()*, *readlines()*, *writelines()*, *seek()*, etc;
- *BZ2File* class implements emulated *seek()* support;
- *BZ2File* class implements universal newline support;
- *BZ2File* class offers an optimized line iteration using the readahead algorithm borrowed from file objects;
- Sequential (de)compression supported by *BZ2Compressor* and *BZ2Decompressor* classes;
- One-shot (de)compression supported by *compress()* and *decompress()* functions;
- Thread safety uses individual locking mechanism.

注解: Handling of multi-stream bzip2 files is not supported. Modules such as bz2file let you overcome this.

12.3.1 文件压缩和解压

Handling of compressed files is offered by the *BZ2File* class.

class bz2.**BZ2File**(*filename*[, *mode*[, *buffering*[, *compresslevel*]]])

Open a bz2 file. Mode can be either `'r'` or `'w'`, for reading (default) or writing. When opened for writing, the file will be created if it doesn't exist, and truncated otherwise. If *buffering* is given, `0` means unbuffered, and larger numbers specify the buffer size; the default is `0`. If *compresslevel* is given, it must be a number between `1` and `9`; the default is `9`. Add a `'U'` to mode to open the file for input in *universal newlines* mode. Any line ending in the input file will be seen as a `'\n'` in Python. Also, a file so opened gains the attribute `newlines`; the value for this attribute is one of `None` (no newline read yet), `'\r'`, `'\n'`, `'\r\n'` or a tuple containing all the newline types seen. Universal newlines are available only when reading. Instances support iteration in the same way as normal *file* instances.

BZ2File supports the `with` statement.

在 2.7 版更改: 支持了 `with` 语句。

注解: This class does not support input files containing multiple streams (such as those produced by the **pbzip2** tool). When reading such an input file, only the first stream will be accessible. If you require support for multi-stream files, consider using the third-party `bz2file` module (available from PyPI). This module provides a backport of Python 3.3's *BZ2File* class, which does support multi-stream files.

close()

Close the file. Sets data attribute `closed` to true. A closed file cannot be used for further I/O operations. *close()* may be called more than once without error.

read([*size*])

Read at most *size* uncompressed bytes, returned as a string. If the *size* argument is negative or omitted, read until EOF is reached.

readline([*size*])
Return the next line from the file, as a string, retaining newline. A non-negative *size* argument limits the maximum number of bytes to return (an incomplete line may be returned then). Return an empty string at EOF.

readlines([*size*])
Return a list of lines read. The optional *size* argument, if given, is an approximate bound on the total number of bytes in the lines returned.

xreadlines()
For backward compatibility. *BZ2File* objects now include the performance optimizations previously implemented in the `xreadlines` module.

2.3 版后已移除: This exists only for compatibility with the method by this name on *file* objects, which is deprecated. Use `for line in file` instead.

seek(*offset*[, *whence*])
Move to new file position. Argument *offset* is a byte count. Optional argument *whence* defaults to `os.SEEK_SET` or `0` (offset from start of file; offset should be `>= 0`); other values are `os.SEEK_CUR` or `1` (move relative to current position; offset can be positive or negative), and `os.SEEK_END` or `2` (move relative to end of file; offset is usually negative, although many platforms allow seeking beyond the end of a file).

Note that seeking of bz2 files is emulated, and depending on the parameters the operation may be extremely slow.

tell()
Return the current file position, an integer (may be a long integer).

write(*data*)
Write string *data* to file. Note that due to buffering, *close()* may be needed before the file on disk reflects the data written.

writelines(*sequence_of_strings*)
Write the sequence of strings to the file. Note that newlines are not added. The sequence can be any iterable object producing strings. This is equivalent to calling write() for each string.

12.3.2 Sequential (de)compression

Sequential compression and decompression is done using the classes *BZ2Compressor* and *BZ2Decompressor*.

class bz2.**BZ2Compressor**([*compresslevel*])
Create a new compressor object. This object may be used to compress data sequentially. If you want to compress data in one shot, use the *compress()* function instead. The *compresslevel* parameter, if given, must be a number between `1` and `9`; the default is `9`.

compress(*data*)
Provide more data to the compressor object. It will return chunks of compressed data whenever possible. When you've finished providing data to compress, call the *flush()* method to finish the compression process, and return what is left in internal buffers.

flush()
Finish the compression process and return what is left in internal buffers. You must not use the compressor object after calling this method.

class bz2.**BZ2Decompressor**
Create a new decompressor object. This object may be used to decompress data sequentially. If you want to decompress data in one shot, use the *decompress()* function instead.

decompress(*data*)
Provide more data to the decompressor object. It will return chunks of decompressed data whenever possible. If you try to decompress data after the end of stream is found, *EOFError* will be raised. If any data was found after the end of stream, it'll be ignored and saved in `unused_data` attribute.

12.3.3 一次性压缩或解压

One-shot compression and decompression is provided through the *compress()* and *decompress()* functions.

`bz2.`**compress**(*data*[, *compresslevel*])
Compress *data* in one shot. If you want to compress data sequentially, use an instance of *BZ2Compressor* instead. The *compresslevel* parameter, if given, must be a number between `1` and `9`; the default is `9`.

`bz2.`**decompress**(*data*)
Decompress *data* in one shot. If you want to decompress data sequentially, use an instance of *BZ2Decompressor* instead.

12.4 zipfile —使用 ZIP 存档

1.6 新版功能.

源代码： Lib/zipfile.py

ZIP 文件格式是一个常用的归档与压缩标准。这个模块提供了创建、读取、写入、添加及列出 ZIP 文件的工具。任何对此模块的进阶使用都将需要理解此格式，其定义参见 PKZIP 应用程序笔记。

This module does not currently handle multi-disk ZIP files. It can handle ZIP files that use the ZIP64 extensions (that is ZIP files that are more than 4 GByte in size). It supports decryption of encrypted files in ZIP archives, but it currently cannot create an encrypted file. Decryption is extremely slow as it is implemented in native Python rather than C.

这个模块定义了以下内容：

exception `zipfile.`**BadZipfile**
The error raised for bad ZIP files (old name: `zipfile.error`).

exception `zipfile.`**LargeZipFile**
当 ZIP 文件需要 ZIP64 功能但是未启用时会抛出此错误。

class `zipfile.`**ZipFile**
用于读写 ZIP 文件的类。欲了解构造函数的描述，参阅段落*ZipFile 对象*。

class `zipfile.`**PyZipFile**
用于创建包含 Python 库的 ZIP 归档的类。

class `zipfile.`**ZipInfo**([*filename*[, *date_time*]])
用于表示档案内一个成员信息的类。此类的实例会由*ZipFile* 对象的*getinfo()* 和*infolist()* 方法返回。大多数*zipfile* 模块的用户都不必创建它们，只需使用此模块所创建的实例。*filename* 应当是档案成员的全名，*date_time* 应当是包含六个字段的描述最近修改时间的元组；这些字段的描述请参阅*ZipInfo 对象*。

`zipfile.`**is_zipfile**(*filename*)
根据文件的 Magic Number，如果 *filename* 是一个有效的 ZIP 文件则返回 `True`，否则返回 `False`。*filename* 也可能是一个文件或类文件对象。

在 2.7 版更改: 支持文件或类文件对象。

zipfile.**ZIP_STORED**
未被压缩的归档成员的数字常数。

zipfile.**ZIP_DEFLATED**
The numeric constant for the usual ZIP compression method. This requires the *zlib* module. No other compression methods are currently supported.

参见:

PKZIP 应用程序笔记 Phil Katz 编写的 ZIP 文件格式文档，此格式和使用的算法的创建者。

Info-ZIP 主页 有关 Info-ZIP 项目的 ZIP 存档程序和开发库的信息。

12.4.1 ZipFile 对象

class zipfile.**ZipFile**(*file*[, *mode*[, *compression*[, *allowZip64*]]])
Open a ZIP file, where *file* can be either a path to a file (a string) or a file-like object. The *mode* parameter should be `'r'` to read an existing file, `'w'` to truncate and write a new file, or `'a'` to append to an existing file. If *mode* is `'a'` and *file* refers to an existing ZIP file, then additional files are added to it. If *file* does not refer to a ZIP file, then a new ZIP archive is appended to the file. This is meant for adding a ZIP archive to another file (such as `python.exe`).

在 2.6 版更改: If *mode* is `a` and the file does not exist at all, it is created.

compression is the ZIP compression method to use when writing the archive, and should be *ZIP_STORED* or *ZIP_DEFLATED*; unrecognized values will cause *RuntimeError* to be raised. If *ZIP_DEFLATED* is specified but the *zlib* module is not available, *RuntimeError* is also raised. The default is *ZIP_STORED*. If *allowZip64* is `True` zipfile will create ZIP files that use the ZIP64 extensions when the zipfile is larger than 2 GB. If it is false (the default) *zipfile* will raise an exception when the ZIP file would require ZIP64 extensions. ZIP64 extensions are disabled by default because the default **zip** and **unzip** commands on Unix (the InfoZIP utilities) don't support these extensions.

在 2.7.1 版更改: If the file is created with mode `'a'` or `'w'` and then *closed* without adding any files to the archive, the appropriate ZIP structures for an empty archive will be written to the file.

ZipFile is also a context manager and therefore supports the `with` statement. In the example, *myzip* is closed after the `with` statement's suite is finished—even if an exception occurs:

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

2.7 新版功能: 添加了将*ZipFile* 用作上下文管理员的功能。

ZipFile.**close**()
关闭归档文件。你必须在退出程序之前调用*close()* 否则将不会写入关键记录数据。

ZipFile.**getinfo**(*name*)
返回一个*ZipInfo* 对象，其中包含有关归档成员 *name* 的信息。针对一个目前并不包含于归档中的名称调用*getinfo()* 将会引发*KeyError*。

ZipFile.**infolist**()
返回一个列表，其中包含每个归档成员的*ZipInfo* 对象。如果是打开一个现有归档则这些对象的排列顺序与它们对应条目在磁盘上的实际 ZIP 文件中的顺序一致。

ZipFile.**namelist**()
返回按名称排序的归档成员列表。

ZipFile.**open**(*name*[, *mode*[, *pwd*]])
Extract a member from the archive as a file-like object (ZipExtFile). *name* is the name of the file in the archive, or a *ZipInfo* object. The *mode* parameter, if included, must be one of the following: `'r'` (the default), `'U'`, or

'rU'. Choosing 'U' or 'rU' will enable *universal newline* support in the read-only object. *pwd* is the password used for encrypted files. Calling `open()` on a closed ZipFile will raise a `RuntimeError`.

注解: The file-like object is read-only and provides the following methods: `read()`, `readline()`, `readlines()`, `__iter__()`, `next()`.

注解: If the ZipFile was created by passing in a file-like object as the first argument to the constructor, then the object returned by `open()` shares the ZipFile's file pointer. Under these circumstances, the object returned by `open()` should not be used after any additional operations are performed on the ZipFile object. If the ZipFile was created by passing in a string (the filename) as the first argument to the constructor, then `open()` will create a new file object that will be held by the ZipExtFile, allowing it to operate independently of the ZipFile.

注解: `open()`, `read()` 和`extract()` 方法可接受文件名或`ZipInfo` 对象。当尝试读取一个包含重复名称成员的 ZIP 文件时你将发现此功能很有好处。

2.6 新版功能.

ZipFile.**extract**(*member*[, *path*[, *pwd*]])
Extract a member from the archive to the current working directory; *member* must be its full name or a `ZipInfo` object). Its file information is extracted as accurately as possible. *path* specifies a different directory to extract to. *member* can be a filename or a `ZipInfo` object. *pwd* is the password used for encrypted files.

返回所创建的经正规化的路径（对应于目录或新文件）。

2.6 新版功能.

注解: 如果一个成员文件名为绝对路径，则将去掉驱动器/UNC 共享点和前导的（反）斜杠，例如: `///foo/bar` 在 Unix 上将变为 `foo/bar`，而 `C:\foo\bar` 在 Windows 上将变为 `foo\bar`。并且一个成员文件名中的所有 `".."` 都将被移除，例如: `../../foo../../ba..r` 将变为 `foo../ba..r`。在 Windows 上非法字符 (`:`, <, >, `|`, `"`, `?`, and `*`) 会被替换为下划线 (_)。

ZipFile.**extractall**([*path*[, *members*[, *pwd*]]])
从归档中提取出所有成员放入当前工作目录。*path* 指定一个要提取到的不同目录。*members* 为可选项且必须为`namelist()` 所返回列表的一个子集。*pwd* 是用于解密文件的密码。

警告: Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of *path*, e.g. members that have absolute filenames starting with `"/"` or filenames with two dots `".."`.

在 2.7.4 版更改: The zipfile module attempts to prevent that. See `extract()` note.

2.6 新版功能.

ZipFile.**printdir**()
将归档的目录表打印到 `sys.stdout`。

ZipFile.**setpassword**(*pwd*)
设置 *pwd* 为用于提取已加密文件的默认密码。

2.6 新版功能.

ZipFile.**read**(*name*[, *pwd*])
Return the bytes of the file *name* in the archive. *name* is the name of the file in the archive, or a *ZipInfo* object. The archive must be open for read or append. *pwd* is the password used for encrypted files and, if specified, it will override the default password set with *setpassword()*. Calling *read()* on a closed ZipFile will raise a *RuntimeError*.

在 2.6 版更改: *pwd* was added, and *name* can now be a *ZipInfo* object.

ZipFile.**testzip**()
Read all the files in the archive and check their CRC's and file headers. Return the name of the first bad file, or else return None. Calling *testzip()* on a closed ZipFile will raise a *RuntimeError*.

ZipFile.**write**(*filename*[, *arcname*[, *compress_type*]])
Write the file named *filename* to the archive, giving it the archive name *arcname* (by default, this will be the same as *filename*, but without a drive letter and with leading path separators removed). If given, *compress_type* overrides the value given for the *compression* parameter to the constructor for the new entry. The archive must be open with mode 'w' or 'a' –calling *write()* on a ZipFile created with mode 'r' will raise a *RuntimeError*. Calling *write()* on a closed ZipFile will raise a *RuntimeError*.

注解: There is no official file name encoding for ZIP files. If you have unicode file names, you must convert them to byte strings in your desired encoding before passing them to *write()*. WinZip interprets all file names as encoded in CP437, also known as DOS Latin.

注解: 归档名称应当是基于归档根目录的相对路径，也就是说，它们不应以路径分隔符开头。

注解: 如果 arcname (或 filename，如果 arcname 未给出) 包含一个空字节，则归档中该文件的名称将在空字节位置被截断。

ZipFile.**writestr**(*zinfo_or_arcname*, *bytes*[, *compress_type*])
Write the string *bytes* to the archive; *zinfo_or_arcname* is either the file name it will be given in the archive, or a *ZipInfo* instance. If it's an instance, at least the filename, date, and time must be given. If it's a name, the date and time is set to the current date and time. The archive must be opened with mode 'w' or 'a' –calling *writestr()* on a ZipFile created with mode 'r' will raise a *RuntimeError*. Calling *writestr()* on a closed ZipFile will raise a *RuntimeError*.

If given, *compress_type* overrides the value given for the *compression* parameter to the constructor for the new entry, or in the *zinfo_or_arcname* (if that is a *ZipInfo* instance).

注解: 当传入一个*ZipInfo* 实例作为 *zinfo_or_arcname* 形参时，所使用的压缩方法将为在给定的*ZipInfo* 实例的 *compress_type* 成员中指定的方法。默认情况下，*ZipInfo* 构造器将将此成员设为*ZIP_STORED*。

在 2.7 版更改: *compress_type* 参数。

以下数据属性也是可用的:

ZipFile.**debug**
要使用的调试输出等级。这可以设为从 0 (默认无输出) 到 3 (最多输出) 的值。调试信息会被写入 sys.stdout。

ZipFile.**comment**
The comment text associated with the ZIP file. If assigning a comment to a *ZipFile* instance created with mode

'a' or 'w', this should be a string no longer than 65535 bytes. Comments longer than this will be truncated in the written archive when *close()* is called.

12.4.2 PyZipFile 对象

The *PyZipFile* constructor takes the same parameters as the *ZipFile* constructor. Instances have one method in addition to those of *ZipFile* objects.

PyZipFile.**writepy**(*pathname*[, *basename*])
Search for files *.py and add the corresponding file to the archive. The corresponding file is a *.pyo file if available, else a *.pyc file, compiling if necessary. If the pathname is a file, the filename must end with .py, and just the (corresponding *.py[co]) file is added at the top level (no path information). If the pathname is a file that does not end with .py, a *RuntimeError* will be raised. If it is a directory, and the directory is not a package directory, then all the files *.py[co] are added at the top level. If the directory is a package directory, then all *.py[co] are added under the package name as a file path, and if any subdirectories are package directories, all of these are added recursively. *basename* is intended for internal use only. The *writepy()* method makes archives with file names like this:

```
string.pyc                                    # Top level name
test/__init__.pyc                             # Package directory
test/test_support.pyc                             # Module test.test_support
test/bogus/__init__.pyc                       # Subpackage directory
test/bogus/myfile.pyc                         # Submodule test.bogus.myfile
```

12.4.3 ZipInfo 对象

ZipInfo 类的实例会通过*getinfo()* 和*ZipFile* 对象的*infolist()* 方法返回。每个对象将存储关于 ZIP 归档的一个成员的信息。

Instances have the following attributes:

ZipInfo.**filename**
归档中的文件名称。

ZipInfo.**date_time**
上次修改存档成员的时间和日期。这是六个值的元组：

索引	值
0	Year (>= 1980)
1	月（1 为基数）
2	月份中的日期（1 为基数）
3	小时（0 为基数）
4	分钟（0 为基数）
5	秒（0 为基数）

注解： ZIP 文件格式不支持 1980 年以前的时间戳。

ZipInfo.**compress_type**
归档成员的压缩类型。

ZipInfo.**comment**
Comment for the individual archive member.

ZipInfo.**extra**
Expansion field data. The PKZIP Application Note contains some comments on the internal structure of the data contained in this string.

ZipInfo.**create_system**
创建 ZIP 归档所用的系统。

ZipInfo.**create_version**
创建 ZIP 归档所用的 PKZIP 版本。

ZipInfo.**extract_version**
需要用来提取归档的 PKZIP 版本。

ZipInfo.**reserved**
必须为零。

ZipInfo.**flag_bits**
ZIP 标志位。

ZipInfo.**volume**
文件中的分卷号。

ZipInfo.**internal_attr**
内部属性。

ZipInfo.**external_attr**
外部文件属性。

ZipInfo.**header_offset**
文件头的字节偏移量。

ZipInfo.**CRC**
未压缩文件的 CRC-32。

ZipInfo.**compress_size**
已压缩数据的大小。

ZipInfo.**file_size**
未压缩文件的大小。

12.4.4 命令行界面

zipfile 模块提供了简单的命令行接口用于与 ZIP 归档的交互。

如果你想要创建一个新的 ZIP 归档，请在*-c* 选项后指定其名称然后列出应当被包含的文件名:

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

传入一个字典也是可接受的:

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

如果你想要将一个 ZIP 归档提取到指定的目录，请使用*-e* 选项:

```
$ python -m zipfile -e monty.zip target-dir/
```

对于一个 ZIP 归档中的文件列表，请使用*-l* 选项:

```
$ python -m zipfile -l monty.zip
```

命令行选项

-l <zipfile>
列出一个 zipfile 中的文件名。

-c <zipfile> <source1> ... <sourceN>
基于源文件创建 zipfile。

-e <zipfile> <output_dir>
将 zipfile 提取到目标目录中。

-t <zipfile>
检测 zipfile 是否有效。

12.5 tarfile —读写 tar 归档文件

2.3 新版功能.

源代码: Lib/tarfile.py

The *tarfile* module makes it possible to read and write tar archives, including those using gzip or bz2 compression. Use the *zipfile* module to read or write `.zip` files, or the higher-level functions in *shutil*.

一些事实和数字:

- reads and writes *gzip* and *bz2* compressed archives if the respective modules are available.
- 支持读取 / 写入 POSIX.1-1988 (ustar) 格式。
- read/write support for the GNU tar format including *longname* and *longlink* extensions, read-only support for the *sparse* extension.
- 对 POSIX.1-2001 (pax) 格式的读/写支持。

  2.6 新版功能.
- 处理目录、正常文件、硬链接、符号链接、fifo 管道、字符设备和块设备，并且能够获取和恢复文件信息例如时间戳、访问权限和所有者等。

注解: Handling of multi-stream bzip2 files is not supported. Modules such as bz2file let you overcome this.

---

tarfile.**open**(*name=None*, *mode='r'*, *fileobj=None*, *bufsize=10240*, ***kwargs*)
针对路径名 *name* 返回*TarFile* 对象。有关*TarFile* 对象以及所允许的关键字参数的详细信息请参阅*TarFile 对象*。

mode 必须是 `'filemode[:compression]'` 形式的字符串，其默认值为 `'r'`。以下是模式组合的完整列表:

模式	action
`'r' or 'r:*'`	打开和读取使用透明压缩（推荐）。
`'r:'`	打开和读取不使用压缩。
`'r:gz'`	打开和读取使用 gzip 压缩。
`'r:bz2'`	打开和读取使用 bzip2 压缩。
`'a' or 'a:'`	打开以便在没有压缩的情况下追加。如果文件不存在，则创建该文件。
`'w' or 'w:'`	打开用于未压缩的写入。
`'w:gz'`	打开用于 gzip 压缩的写入。
`'w:bz2'`	打开用于 bzip2 压缩的写入。

Note that `'a:gz'` or `'a:bz2'` is not possible. If *mode* is not suitable to open a certain (compressed) file for reading, *ReadError* is raised. Use *mode* `'r'` to avoid this. If a compression method is not supported, *CompressionError* is raised.

If *fileobj* is specified, it is used as an alternative to a file object opened for *name*. It is supposed to be at position 0.

For modes `'w:gz'`, `'r:gz'`, `'w:bz2'`, `'r:bz2'`, *tarfile.open()* accepts the keyword argument *compresslevel* (default `9`) to specify the compression level of the file.

For special purposes, there is a second format for *mode*: `'filemode|[compression]'`. *tarfile.open()* will return a *TarFile* object that processes its data as a stream of blocks. No random seeking will be done on the file. If given, *fileobj* may be any object that has a `read()` or `write()` method (depending on the *mode*). *bufsize* specifies the blocksize and defaults to `20 * 512` bytes. Use this variant in combination with e.g. `sys.stdin`, a socket file object or a tape device. However, such a *TarFile* object is limited in that it does not allow random access, see 例子. The currently possible modes:

模式	动作
`'r|*'`	打开 tar 块的 流以进行透明压缩读取。
`'r|'`	打开一个未压缩的 tar 块的 *stream* 用于读取。
`'r|gz'`	打开一个 gzip 压缩的 *stream* 用于读取。
`'r|bz2'`	打开一个 bzip2 压缩的 *stream* 用于读取。
`'w|'`	打开一个未压缩的 *stream* 用于写入。
`'w|gz'`	打开一个 gzip 压缩的 *stream* 用于写入。
`'w|bz2'`	打开一个 bzip2 压缩的 *stream* 用于写入。

class tarfile.**TarFile**
Class for reading and writing tar archives. Do not use this class directly, better use *tarfile.open()* instead. See *TarFile 对象*.

tarfile.**is_tarfile**(*name*)
如果 *name* 是一个*tarfile* 模块能读取的 tar 归档文件则返回*True*。

class tarfile.**TarFileCompat**(*filename*, *mode='r'*, *compression=TAR_PLAIN*)
Class for limited access to tar archives with a *zipfile*-like interface. Please consult the documentation of the *zipfile* module for more details. *compression* must be one of the following constants:

TAR_PLAIN
Constant for an uncompressed tar archive.

TAR_GZIPPED
Constant for a *gzip* compressed tar archive.

2.6 版后已移除: The *TarFileCompat* class has been removed in Python 3.

exception tarfile.**TarError**
所有*tarfile* 异常的基类。

exception tarfile.**ReadError**
当一个不能被*tarfile*模块处理或者因某种原因而无效的 tar 归档被打开时将被引发。

exception tarfile.**CompressionError**
当一个压缩方法不受支持或者当数据无法被正确解码时将被引发。

exception tarfile.**StreamError**
当达到流式*TarFile*对象的典型限制时将被引发。

exception tarfile.**ExtractError**
当使用*TarFile.extract()*时针对 *non-fatal* 所引发的异常，但是仅限 TarFile.errorlevel==2。

以下常量在模块层级上可用:

tarfile.**ENCODING**
默认的字符编码格式: 在 Windows 上为 'utf-8'，其他系统上则为*sys.getfilesystemencoding()*所返回的值。

exception tarfile.**HeaderError**
如果获取的缓冲区无效则会由*TarInfo.frombuf()*引发的异常。

2.6 新版功能.

以下常量各自定义了一个*tarfile*模块能够创建的 tar 归档格式。相关细节请参阅*受支持的 tar 格式*小节。

tarfile.**USTAR_FORMAT**
POSIX.1-1988 (ustar) 格式。

tarfile.**GNU_FORMAT**
GNU tar 格式。

tarfile.**PAX_FORMAT**
POSIX.1-2001 (pax) 格式。

tarfile.**DEFAULT_FORMAT**
用于创建归档的默认格式。目前为*GNU_FORMAT*。

参见:

模块*zipfile* *zipfile* 标准模块的文档。

归档操作 标准*shutil*模块所提供的高层级归档工具的文档。

GNU tar manual, Basic Tar Format 针对 tar 归档文件的文档，包含 GNU tar 扩展。

12.5.1 TarFile 对象

*TarFile*对象提供了一个 tar 归档的接口。一个 tar 归档就是数据块的序列。一个归档成员（被保存文件）是由一个标头块加多个数据块组成的。一个文件可以在一个 tar 归档中多次被保存。每个归档成员都由一个*TarInfo*对象来代表，详情参见*TarInfo 对象*。

*TarFile*对象可在 with 语句中作为上下文管理器使用。当语句块结束时它将自动被关闭。请注意在发生异常事件时被打开用于写入的归档将不会被终结；只有内部使用的文件对象将被关闭。相关用例请参见*例子*。

2.7 新版功能: 添加了对上下文管理器协议的支持。

class tarfile.**TarFile**(*name=None*, *mode='r'*, *fileobj=None*, *format=DEFAULT_FORMAT*, *tarinfo=TarInfo*, *dereference=False*, *ignore_zeros=False*, *encoding=ENCODING*, *errors=None*, *pax_headers=None*, *debug=0*, *errorlevel=0*)
下列所有参数都是可选项并且也可作为实例属性来访问。

name is the pathname of the archive. It can be omitted if *fileobj* is given. In this case, the file object's `name` attribute is used if it exists.

mode is either `'r'` to read from an existing archive, `'a'` to append data to an existing file or `'w'` to create a new file overwriting an existing one.

如果给定了 *fileobj*，它会被用于读取或写入数据。如果可以被确定，则 *mode* 会被 *fileobj* 的模式所覆盖。*fileobj* 的使用将从位置 0 开始。

注解： 当*TarFile* 被关闭时，*fileobj* 不会被关闭。

---

format 控制归档格式。它必须为在模块层级定义的常量*USTAR_FORMAT*, *GNU_FORMAT* 或*PAX_FORMAT* 中的一个。

2.6 新版功能.

tarinfo 参数可以被用来将默认的*TarInfo* 类替换为另一个。

2.6 新版功能.

如果 *dereference* 为*False*，则会将符号链接和硬链接添加到归档中。如果为*True*，则会将目标文件的内容添加到归档中。在不支持符号链接的系统上参数将不起作用。

如果 *ignore_zeros* 为*False*，则会将空的数据块当作归档的末尾来处理。如果为*True*，则会跳过空的（和无效的）数据块并尝试获取尽可能多的成员。此参数仅适用于读取拼接的或损坏的归档。

debug 可设为从 `0` (无调试消息) 到 `3` (全部调试消息)。消息会被写入到 `sys.stderr`。

If *errorlevel* is `0`, all errors are ignored when using *TarFile.extract()*. Nevertheless, they appear as error messages in the debug output, when debugging is enabled. If `1`, all *fatal* errors are raised as *OSError* or *IOError* exceptions. If `2`, all *non-fatal* errors are raised as *TarError* exceptions as well.

The *encoding* and *errors* arguments control the way strings are converted to unicode objects and vice versa. The default settings will work for most users. See section *Unicode 问题* for in-depth information.

2.6 新版功能.

The *pax_headers* argument is an optional dictionary of unicode strings which will be added as a pax global header if *format* is *PAX_FORMAT*.

2.6 新版功能.

classmethod TarFile.**open**(...)
作为替代的构造器。*tarfile.open()* 函数实际上是这个类方法的快捷方式。

TarFile.**getmember**(*name*)
返回成员 *name* 的*TarInfo* 对象。如果 *name* 在归档中找不到，则会引发*KeyError*。

注解： 如果一个成员在归档中出现超过一次，它的最后一次出现会被视为是最新的版本。

TarFile.**getmembers**()
以*TarInfo* 对象列表的形式返回归档的成员。列表的顺序与归档中成员的顺序一致。

TarFile.**getnames**()
以名称列表的形式返回成员。它的顺序与*getmembers()* 所返回列表的顺序一致。

TarFile.**list**(*verbose=True*)
Print a table of contents to `sys.stdout`. If *verbose* is *False*, only the names of the members are printed. If it is *True*, output similar to that of **`ls -l`** is produced.

TarFile.**next**()
当*TarFile* 被打开用于读取时，以*TarInfo* 对象的形式返回归档的下一个成员。如果不再有可用对象则返回*None*。

TarFile.**extractall**(*path=".", members=None*)
将归档中的所有成员提取到当前工作目录或 *path* 目录。如果给定了可选的 *members*，则它必须为*getmembers()* 所返回的列表的一个子集。字典信息例如所有者、修改时间和权限会在所有成员提取完毕后被设置。这样做是为了避免两个问题：目录的修改时间会在每当在其中创建文件时被重置。并且如果目录的权限不允许写入，提取文件到目录的操作将失败。

> **警告：** 绝不要未经预先检验就从不可靠的源中提取归档文件。这样有可能在 *path* 之外创建文件，例如某些成员具有以 "/" 开始的绝对路径文件名或带有两个点号 ".." 的文件名。

2.5 新版功能.

TarFile.**extract**(*member, path=""*)
Extract a member from the archive to the current working directory, using its full name. Its file information is extracted as accurately as possible. *member* may be a filename or a *TarInfo* object. You can specify a different directory using *path*.

注解： *extract()* 方法不会处理某些提取问题。在大多数情况下你应当考虑使用*extractall()* 方法。

> **警告：** 参看*extractall()* 的警告信息。

TarFile.**extractfile**(*member*)
Extract a member from the archive as a file object. *member* may be a filename or a *TarInfo* object. If *member* is a regular file, a file-like object is returned. If *member* is a link, a file-like object is constructed from the link's target. If *member* is none of the above, *None* is returned.

注解： The file-like object is read-only. It provides the methods `read()`, *readline()*, `readlines()`, `seek()`, `tell()`, and *close()*, and also supports iteration over its lines.

TarFile.**add**(*name, arcname=None, recursive=True, exclude=None, filter=None*)
Add the file *name* to the archive. *name* may be any type of file (directory, fifo, symbolic link, etc.). If given, *arcname* specifies an alternative name for the file in the archive. Directories are added recursively by default. This can be avoided by setting *recursive* to *False*. If *exclude* is given it must be a function that takes one filename argument and returns a boolean value. Depending on this value the respective file is either excluded (*True*) or added (*False*). If *filter* is specified it must be a function that takes a *TarInfo* object argument and returns the changed *TarInfo* object. If it instead returns *None* the *TarInfo* object will be excluded from the archive. See 例子 for an example.

在 2.6 版更改: Added the *exclude* parameter.

在 2.7 版更改: 添加了 *filter* 形参。

2.7 版后已移除: The *exclude* parameter is deprecated, please use the *filter* parameter instead. For maximum portability, *filter* should be used as a keyword argument rather than as a positional argument so that code won't be affected when *exclude* is ultimately removed.

TarFile.**addfile**(*tarinfo*, *fileobj=None*)
Add the *TarInfo* object *tarinfo* to the archive. If *fileobj* is given, `tarinfo.size` bytes are read from it and added to the archive. You can create *TarInfo* objects directly, or by using *gettarinfo()*.

注解: On Windows platforms, *fileobj* should always be opened with mode `'rb'` to avoid irritation about the file size.

TarFile.**gettarinfo**(*name=None*, *arcname=None*, *fileobj=None*)
Create a *TarInfo* object from the result of *os.stat()* or equivalent on an existing file. The file is either named by *name*, or specified as a file object *fileobj* with a file descriptor. If given, *arcname* specifies an alternative name for the file in the archive, otherwise, the name is taken from *fileobj*' s *name* attribute, or the *name* argument.

你可以在使用*addfile()* 添加*TarInfo* 的某些属性之前修改它们。如果文件对象不是从文件开头进行定位的普通文件对象，*size* 之类的属性就可能需要修改。例如*GzipFile* 之类的文件就属于这种情况。*name* 也可以被修改，在这种情况下 *arcname* 可以是一个占位字符串。

TarFile.**close**()
关闭*TarFile*。在写入模式下，会向归档添加两个表示结束的零数据块。

TarFile.**posix**
Setting this to *True* is equivalent to setting the *format* attribute to *USTAR_FORMAT*, *False* is equivalent to *GNU_FORMAT*.

在 2.4 版更改: *posix* defaults to *False*.

2.6 版后已移除: Use the *format* attribute instead.

TarFile.**pax_headers**
一个包含 pax 全局标头的键值对的字典。

2.6 新版功能.

12.5.2 TarInfo 对象

TarInfo 对象代表*TarFile* 中的一个文件。除了会存储所有必要的文件属性（例如文件类型、大小、时间、权限、所有者等），它还提供了一些确定文件类型的有用方法。此对象 并不包含文件数据本身。

TarInfo 对象可通过*TarFile* 的方法 `getmember()`, `getmembers()` 和 `gettarinfo()` 返回。

class tarfile.**TarInfo**(*name=""*)
创建一个*TarInfo* 对象。

TarInfo.**frombuf**(*buf*)
基于字符串缓冲区 *buf* 创建并返回一个*TarInfo* 对象。

2.6 新版功能: Raises *HeaderError* if the buffer is invalid..

TarInfo.**fromtarfile**(*tarfile*)
从*TarFile* 对象 *tarfile* 读取下一个成员并将其作为*TarInfo* 对象返回。

2.6 新版功能.

TarInfo.**tobuf**(*format=DEFAULT_FORMAT*, *encoding=ENCODING*, *errors='strict'*)
基于*TarInfo* 对象创建一个字符串缓冲区。有关参数的信息请参见*TarFile* 类的构造器。

在 2.6 版更改: The arguments were added.

`TarInfo` 对象具有以下公有数据属性:

TarInfo.**name**
归档成员的名称。

TarInfo.**size**
以字节表示的大小。

TarInfo.**mtime**
上次修改的时间。

TarInfo.**mode**
权限位。

TarInfo.**type**
文件类型。*type* 通常为以下常量之一: REGTYPE, AREGTYPE, LNKTYPE, SYMTYPE, DIRTYPE, FIFOTYPE, CONTTYPE, CHRTYPE, BLKTYPE, GNUTYPE_SPARSE。要更方便地确定一个*TarInfo*对象的类型，请使用下述的 is*() 方法。

TarInfo.**linkname**
目标文件名的名称，该属性仅在类型为 LNKTYPE 和 SYMTYPE 的*TarInfo*对象中存在。

TarInfo.**uid**
最初保存该成员的用户的用户 ID。

TarInfo.**gid**
最初保存该成员的用户的分组 ID。

TarInfo.**uname**
用户名。

TarInfo.**gname**
分组名。

TarInfo.**pax_headers**
一个包含所关联的 pax 扩展标头的键值对的字典。

2.6 新版功能.

*TarInfo*对象还提供了一些便捷查询方法:

TarInfo.**isfile**()
如果 Tarinfo 对象为普通文件则返回*True*。

TarInfo.**isreg**()
与*isfile()*相同。

TarInfo.**isdir**()
如果为目录则返回*True*。

TarInfo.**issym**()
如果为符号链接则返回*True*。

TarInfo.**islnk**()
如果为硬链接则返回*True*。

TarInfo.**ischr**()
如果为字符设备则返回*True*。

TarInfo.**isblk**()
如果为块设备则返回*True*。

TarInfo.**isfifo**()
如果为 FIFO 则返回*True*。.

TarInfo.**isdev**()
如果为字符设备、块设备或 FIFO 之一则返回*True*。

12.5.3 例子

如何将整个 tar 归档提取到当前工作目录:

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

如何通过*TarFile.extractall()* 使用生成器函数而非列表来提取一个 tar 归档的子集:

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

如何基于一个文件名列表创建未压缩的 tar 归档:

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

使用 `with` 语句的同一个示例:

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

如何读取一个 gzip 压缩的 tar 归档并显示一些成员信息:

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print tarinfo.name, "is", tarinfo.size, "bytes in size and is",
    if tarinfo.isreg():
        print "a regular file."
    elif tarinfo.isdir():
        print "a directory."
    else:
        print "something else."
tar.close()
```

如何创建一个归档并使用*TarFile.add()* 中的 *filter* 形参来重置用户信息:

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

12.5.4 受支持的 tar 格式

通过*tarfile*模块可以创建三种 tar 格式:

- The POSIX.1-1988 ustar format (*USTAR_FORMAT*). It supports filenames up to a length of at best 256 characters and linknames up to 100 characters. The maximum file size is 8 gigabytes. This is an old and limited but widely supported format.
- The GNU tar format (*GNU_FORMAT*). It supports long filenames and linknames, files bigger than 8 gigabytes and sparse files. It is the de facto standard on GNU/Linux systems. *tarfile* fully supports the GNU tar extensions for long names, sparse file support is read-only.
- POSIX.1-2001 pax 格式 (*PAX_FORMAT*)。它是几乎无限制的最灵活格式。它支持长文件名和链接名，大文件以及使用便捷方式存储路径名。但是并非所有现今的 tar 实现都能够正确地处理 pax 归档。

  pax 格式是对现有 *ustar* 格式的扩展。它会对无法以其他方式存储的信息使用附加标头。存在两种形式的 pax 标头：扩展标头只影响后续的文件标头，全局标头则适用于完整归档并会影响所有后续的文件。为了便于移植，在 pax 标头中的所有数据均以 *UTF-8* 编码。

还有一些 tar 格式的其他变种，它们可以被读取但不能被创建:

- 古老的 V7 格式。这是来自 Unix 第七版的第一个 tar 格式，它只存储常规文件和目录。名称长度不能超过 100 个字符，并且没有用户/分组名信息。某些归档在带有非 ASCII 字符字段的情况下会产生计算错误的标头校验和。
- SunOS tar 扩展格式。此格式是 POSIX.1-2001 pax 格式的一个变种，但并不保持兼容。

12.5.5 Unicode 问题

The tar format was originally conceived to make backups on tape drives with the main focus on preserving file system information. Nowadays tar archives are commonly used for file distribution and exchanging archives over networks. One problem of the original format (that all other formats are merely variants of) is that there is no concept of supporting different character encodings. For example, an ordinary tar archive created on a *UTF-8* system cannot be read correctly on a *Latin-1* system if it contains non-ASCII characters. Names (i.e. filenames, linknames, user/group names) containing these characters will appear damaged. Unfortunately, there is no way to autodetect the encoding of an archive.

The pax format was designed to solve this problem. It stores non-ASCII names using the universal character encoding *UTF-8*. When a pax archive is read, these *UTF-8* names are converted to the encoding of the local file system.

The details of unicode conversion are controlled by the *encoding* and *errors* keyword arguments of the *TarFile* class.

The default value for *encoding* is the local character encoding. It is deduced from *sys.getfilesystemencoding()* and *sys.getdefaultencoding()*. In read mode, *encoding* is used exclusively to convert unicode names from a pax archive to strings in the local character encoding. In write mode, the use of *encoding* depends on the chosen archive format. In case of *PAX_FORMAT*, input names that contain non-ASCII characters need to be decoded before being stored as *UTF-8* strings. The other formats do not make use of *encoding* unless unicode objects are used as input names. These are converted to 8-bit character strings before they are added to the archive.

The *errors* argument defines how characters are treated that cannot be converted to or from *encoding*. Possible values are listed in section 编解码器基类. In read mode, there is an additional scheme `'utf-8'` which means that bad characters are replaced by their *UTF-8* representation. This is the default scheme. In write mode the default value for *errors* is `'strict'` to ensure that name information is not altered unnoticed.

CHAPTER 13

文件格式

The modules described in this chapter parse various miscellaneous file formats that aren't markup languages or are related to e-mail.

13.1 csv —CSV 文件读写

2.3 新版功能.

The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. There is no "CSV standard", so the format is operationally defined by the many applications which read and write it. The lack of a standard means that subtle differences often exist in the data produced and consumed by different applications. These differences can make it annoying to process CSV files from multiple sources. Still, while the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.

The *csv* module implements classes to read and write tabular data in CSV format. It allows programmers to say, "write this data in the format preferred by Excel," or "read data from this file which was generated by Excel," without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.

csv 模块中的*reader* 类和*writer* 类可用于读写序列化的数据。也可使用*DictReader* 类和*DictWriter* 类以字典的形式读写数据。

注解: This version of the *csv* module doesn't support Unicode input. Also, there are currently some issues regarding ASCII NUL characters. Accordingly, all input should be UTF-8 or printable ASCII to be safe; see the examples in section 例子.

参见:

该实现在 "Python 增强提议" - PEP *305* (CSV 文件 API) 中被提出《Python 增强提议》提出了对 Python 的这一补充。

13.1.1 模块内容

csv 模块定义了以下函数:

csv.**reader**(*csvfile*, *dialect='excel'*, ***fmtparams*)
Return a reader object which will iterate over lines in the given *csvfile*. *csvfile* can be any object which supports the *iterator* protocol and returns a string each time its `next()` method is called —file objects and list objects are both suitable. If *csvfile* is a file object, it must be opened with the 'b' flag on platforms where that makes a difference. An optional *dialect* parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the *Dialect* class or one of the strings returned by the *list_dialects()* function. The other optional *fmtparams* keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section 变种与格式参数.

Each row read from the csv file is returned as a list of strings. No automatic data type conversion is performed.

一个简短的用法示例:

```
>>> import csv
>>> with open('eggs.csv', 'rb') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print ', '.join(row)
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

在 2.5 版更改: The parser is now stricter with respect to multi-line quoted fields. Previously, if a line ended within a quoted field without a terminating newline character, a newline would be inserted into the returned field. This behavior caused problems when reading files which contained carriage return characters within fields. The behavior was changed to return the field without inserting newlines. As a consequence, if newlines embedded within fields are important, the input should be split into lines in a manner which preserves the newline characters.

csv.**writer**(*csvfile*, *dialect='excel'*, ***fmtparams*)
Return a writer object responsible for converting the user's data into delimited strings on the given file-like object. *csvfile* can be any object with a `write()` method. If *csvfile* is a file object, it must be opened with the 'b' flag on platforms where that makes a difference. An optional *dialect* parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the *Dialect* class or one of the strings returned by the *list_dialects()* function. The other optional *fmtparams* keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section 变种与格式参数. To make it as easy as possible to interface with modules which implement the DB API, the value *None* is written as the empty string. While this isn't a reversible transformation, it makes it easier to dump SQL NULL data values to CSV files without preprocessing the data returned from a `cursor.fetch*` call. Floats are stringified with *repr()* before being written. All other non-string data are stringified with *str()* before being written.

一个简短的用法示例:

```
import csv
with open('eggs.csv', 'wb') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

csv.**register_dialect**(*name*[, *dialect*], ***fmtparams*)
Associate *dialect* with *name*. *name* must be a string or Unicode object. The dialect can be specified either by passing a sub-class of *Dialect*, or by *fmtparams* keyword arguments, or both, with keyword arguments overriding

parameters of the dialect. For full details about the dialect and formatting parameters, see section 变种与格式参数.

csv.**unregister_dialect**(*name*)
从变种注册表中删除 *name* 对应的变种。如果 *name* 不是已注册的变种名称，则抛出*Error* 异常。

csv.**get_dialect**(*name*)
Return the dialect associated with *name*. An *Error* is raised if *name* is not a registered dialect name.

在 2.5 版更改: This function now returns an immutable *Dialect*. Previously an instance of the requested dialect was returned. Users could modify the underlying class, changing the behavior of active readers and writers.

csv.**list_dialects**()
返回所有已注册变种的名称。

csv.**field_size_limit**([*new_limit*])
返回解析器当前允许的最大字段大小。如果指定了 *new_limit*，则它将成为新的最大字段大小。

2.5 新版功能.

csv 模块定义了以下类:

class csv.**DictReader**(*f*, *fieldnames=None*, *restkey=None*, *restval=None*, *dialect='excel'*, **args*, ***kwds*)
Create an object which operates like a regular reader but maps the information read into a dict whose keys are given by the optional *fieldnames* parameter. The *fieldnames* parameter is a *sequence* whose elements are associated with the fields of the input data in order. These elements become the keys of the resulting dictionary. If the *fieldnames* parameter is omitted, the values in the first row of the file *f* will be used as the fieldnames. If the row read has more fields than the fieldnames sequence, the remaining data is added as a sequence keyed by the value of *restkey*. If the row read has fewer fields than the fieldnames sequence, the remaining keys take the value of the optional *restval* parameter. Any other optional or keyword arguments are passed to the underlying *reader* instance.

一个简短的用法示例:

```
>>> import csv
>>> with open('names.csv') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Baked Beans
Lovely Spam
Wonderful Spam
```

class csv.**DictWriter**(*f*, *fieldnames*, *restval=''*, *extrasaction='raise'*, *dialect='excel'*, **args*, ***kwds*)
Create an object which operates like a regular writer but maps dictionaries onto output rows. The *fieldnames* parameter is a *sequence* of keys that identify the order in which values in the dictionary passed to the `writerow()` method are written to the file *f*. The optional *restval* parameter specifies the value to be written if the dictionary is missing a key in *fieldnames*. If the dictionary passed to the `writerow()` method contains a key not found in *fieldnames*, the optional *extrasaction* parameter indicates what action to take. If it is set to `'raise'` a *ValueError* is raised. If it is set to `'ignore'`, extra values in the dictionary are ignored. Any other optional or keyword arguments are passed to the underlying *writer* instance.

Note that unlike the *DictReader* class, the *fieldnames* parameter of the *DictWriter* is not optional. Since Python's *dict* objects are not ordered, there is not enough information available to deduce the order in which the row should be written to the file *f*.

一个简短的用法示例:

```
import csv
```

(下页继续)

(续上页)

```
with open('names.csv', 'w') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

class csv.**Dialect**
Dialect 类是主要依赖于其属性的容器类，用于将定义好的参数传递给特定的*reader* 或*writer* 实例。

class csv.**excel**
excel 类定义了 Excel 生成的 CSV 文件的常规属性。它在变种注册表中的名称是 'excel'。

class csv.**excel_tab**
excel_tab 类定义了 Excel 生成的、制表符分隔的 CSV 文件的常规属性。它在变种注册表中的名称是 'excel-tab'。

class csv.**Sniffer**
Sniffer 类用于推断 CSV 文件的格式。

Sniffer 类提供了两个方法：

sniff(*sample*, *delimiters=None*)
分析给定的 *sample* 并返回一个*Dialect* 子类，该子类中包含了分析出的格式参数。如果给出可选的 *delimiters* 参数，则该参数会被解释为字符串，该字符串包含了可能的有效分隔符。

has_header(*sample*)
分析示例文本（假定为 CSV 格式），如果第一行很可能是一系列列标题，则返回*True*。

使用*Sniffer* 的示例：

```
with open('example.csv', 'rb') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

csv 模块定义了以下常量：

csv.**QUOTE_ALL**
指示*writer* 对象给所有字段加上引号。

csv.**QUOTE_MINIMAL**
指示*writer* 对象仅为包含特殊字符（例如 定界符、引号字符或 行结束符中的任何字符）的字段加上引号。

csv.**QUOTE_NONNUMERIC**
指示*writer* 对象为所有非数字字段加上引号。

指示 reader 将所有未用引号引出的字段转换为 *float* 类型。

csv.**QUOTE_NONE**
指示*writer* 对象不使用引号引出字段。当 定界符出现在输出数据中时，其前面应该有 转义符。如果未设置 转义符，则遇到任何需要转义的字符时，writer 都会抛出*Error* 异常。

指示*reader* 不对引号字符进行特殊处理。

csv 模块定义了以下异常：

exception csv.**Error**
该异常可能由任何发生错误的函数抛出。

13.1.2 变种与格式参数

为了更容易指定输入和输出记录的格式，特定的一组格式参数组合为一个 dialect（变种）。一个 dialect 是一个*Dialect* 类的子类，它具有一组特定的方法和一个 validate() 方法。创建*reader* 或*writer* 对象时，程序员可以将某个字符串或*Dialect* 类的子类指定为 dialect 参数。要想补充或覆盖 *dialect* 参数，程序员还可以单独指定某些格式参数，这些参数的名称与下面*Dialect* 类定义的属性相同。

Dialect 类支持以下属性：

Dialect.**delimiter**
一个用于分隔字段的单字符，默认为 ','。

Dialect.**doublequote**
控制出现在字段中的 引号字符本身应如何被引出。当该属性为*True* 时，双写引号字符。如果该属性为*False*，则在 引号字符的前面放置 转义符。默认值为*True*。

在输出时，如果 *doublequote* 是*False*，且 转义符未指定，且在字段中发现 引号字符时，会抛出*Error* 异常。

Dialect.**escapechar**
一个用于 writer 的单字符，用来在 *quoting* 设置为*QUOTE_NONE* 的情况下转义 定界符，在 *doublequote* 设置为*False* 的情况下转义 引号字符。在读取时，*escapechar* 去除了其后所跟字符的任何特殊含义。该属性默认为*None*，表示禁用转义。

Dialect.**lineterminator**
放在*writer* 产生的行的结尾，默认为 '\r\n'。

注解： *reader* 经过硬编码，会识别 '\r' 或 '\n' 作为行尾，并忽略 *lineterminator*。未来可能会更改这一行为。

Dialect.**quotechar**
一个单字符，用于包住含有特殊字符的字段，特殊字符如 定界符或 引号字符或换行符。默认为 '"'。

Dialect.**quoting**
控制 writer 何时生成引号，以及 reader 何时识别引号。该属性可以等于任何 QUOTE_* 常量（参见模块内容 段落），默认为*QUOTE_MINIMAL*。

Dialect.**skipinitialspace**
如果为*True*，则忽略 定界符之后的空格。默认值为*False*。

Dialect.**strict**
如果为 True，则在输入错误的 CSV 时抛出*Error* 异常。默认值为 False。

13.1.3 Reader 对象

Reader 对象（*DictReader* 实例和*reader()* 函数返回的对象）具有以下公开方法：

csvreader.**next**()
Return the next row of the reader’s iterable object as a list, parsed according to the current dialect.

Reader 对象具有以下公开属性：

csvreader.**dialect**
变种描述，只读，供解析器使用。

csvreader.**line_num**
源迭代器已经读取了的行数。它与返回的记录数不同，因为记录可能跨越多行。

2.5 新版功能.

DictReader 对象具有以下公开属性：

csvreader.**fieldnames**
字段名称。如果在创建对象时未传入字段名称，则首次访问时或从文件中读取第一条记录时会初始化此属性。

在 2.6 版更改.

13.1.4 Writer 对象

Writer objects (*DictWriter* instances and objects returned by the *writer()* function) have the following public methods. A *row* must be a sequence of strings or numbers for Writer objects and a dictionary mapping fieldnames to strings or numbers (by passing them through *str()* first) for *DictWriter* objects. Note that complex numbers are written out surrounded by parens. This may cause some problems for other programs which read CSV files (assuming they support complex numbers at all).

csvwriter.**writerow**(*row*)
将 *row* 形参写入 writer 的文件对象，并按照当前设定形式进行格式化。

csvwriter.**writerows**(*rows*)
将 *rows** （即能迭代出多个上述 **row* 对象的迭代器）中的所有元素写入 writer 的文件对象，并根据当前设置的变种进行格式化。

Writer 对象具有以下公开属性：

csvwriter.**dialect**
变种描述，只读，供 writer 使用。

DictWriter 对象具有以下公开方法：

DictWriter.**writeheader**()
使用（构造器所规定的）字段名写入一行。

2.7 新版功能.

13.1.5 例子

读取 CSV 文件最简单的一个例子:

```
import csv
with open('some.csv', 'rb') as f:
    reader = csv.reader(f)
    for row in reader:
        print row
```

读取其他格式的文件:

```
import csv
with open('passwd', 'rb') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print row
```

相应最简单的写入示例是:

```
import csv
with open('some.csv', 'wb') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

注册一个新的变种:

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', 'rb') as f:
    reader = csv.reader(f, 'unixpwd')
```

Reader 的更高级用法——捕获并报告错误:

```
import csv, sys
filename = 'some.csv'
with open(filename, 'rb') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print row
    except csv.Error as e:
        sys.exit('file %s, line %d: %s' % (filename, reader.line_num, e))
```

尽管该模块不直接支持解析字符串，但仍可如下轻松完成:

```
import csv
for row in csv.reader(['one,two,three']):
    print row
```

The *csv* module doesn't directly support reading and writing Unicode, but it is 8-bit-clean save for some problems with ASCII NUL characters. So you can write functions or classes that handle the encoding and decoding for you as long as you avoid encodings like UTF-16 that use NULs. UTF-8 is recommended.

`unicode_csv_reader()` below is a *generator* that wraps *csv.reader* to handle Unicode CSV data (a list of Unicode strings). `utf_8_encoder()` is a *generator* that encodes the Unicode strings as UTF-8, one string (or row) at a time. The encoded strings are parsed by the CSV reader, and `unicode_csv_reader()` decodes the UTF-8-encoded cells back into Unicode:

```
import csv

def unicode_csv_reader(unicode_csv_data, dialect=csv.excel, **kwargs):
    # csv.py doesn't do Unicode; encode temporarily as UTF-8:
    csv_reader = csv.reader(utf_8_encoder(unicode_csv_data),
                            dialect=dialect, **kwargs)
    for row in csv_reader:
        # decode UTF-8 back to Unicode, cell by cell:
        yield [unicode(cell, 'utf-8') for cell in row]

def utf_8_encoder(unicode_csv_data):
    for line in unicode_csv_data:
        yield line.encode('utf-8')
```

For all other encodings the following `UnicodeReader` and `UnicodeWriter` classes can be used. They take an additional *encoding* parameter in their constructor and make sure that the data passes the real reader or writer encoded as UTF-8:

```
import csv, codecs, cStringIO

class UTF8Recoder:
    """
    Iterator that reads an encoded stream and reencodes the input to UTF-8
    """
    def __init__(self, f, encoding):
        self.reader = codecs.getreader(encoding)(f)

    def __iter__(self):
        return self

    def next(self):
        return self.reader.next().encode("utf-8")

class UnicodeReader:
    """
    A CSV reader which will iterate over lines in the CSV file "f",
    which is encoded in the given encoding.
    """

    def __init__(self, f, dialect=csv.excel, encoding="utf-8", **kwds):
        f = UTF8Recoder(f, encoding)
        self.reader = csv.reader(f, dialect=dialect, **kwds)

    def next(self):
        row = self.reader.next()
        return [unicode(s, "utf-8") for s in row]

    def __iter__(self):
        return self

class UnicodeWriter:
    """
    A CSV writer which will write rows to CSV file "f",
    which is encoded in the given encoding.
    """

    def __init__(self, f, dialect=csv.excel, encoding="utf-8", **kwds):
```

(下页继续)

(续上页)

```
        # Redirect output to a queue
        self.queue = cStringIO.StringIO()
        self.writer = csv.writer(self.queue, dialect=dialect, **kwds)
        self.stream = f
        self.encoder = codecs.getincrementalencoder(encoding)()

    def writerow(self, row):
        self.writer.writerow([s.encode("utf-8") for s in row])
        # Fetch UTF-8 output from the queue ...
        data = self.queue.getvalue()
        data = data.decode("utf-8")
        # ... and reencode it into the target encoding
        data = self.encoder.encode(data)
        # write to the target stream
        self.stream.write(data)
        # empty queue
        self.queue.truncate(0)

    def writerows(self, rows):
        for row in rows:
            self.writerow(row)
```

13.2 ConfigParser —Configuration file parser

注解： The *ConfigParser* module has been renamed to configparser in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

This module defines the class *ConfigParser*. The *ConfigParser* class implements a basic configuration file parser language which provides a structure similar to what you would find on Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

注解： 这个库 并不能够解析或写入在 Windows Registry 扩展版本 INI 语法中所使用的值-类型前缀。

参见：

模块*shlex* 支持创建可被用作应用配置文件的替代的 Unix 终端式微语言。

模块*json* json 模块实现了一个 JavaScript 语法的子集，它也可被用于这种目的。

The configuration file consists of sections, led by a `[section]` header and followed by `name: value` entries, with continuations in the style of **RFC 822** (see section 3.1.1, "LONG HEADER FIELDS"); `name=value` is also accepted. Note that leading whitespace is removed from values. The optional values can contain format strings which refer to other values in the same section, or values in a special `DEFAULT` section. Additional defaults can be provided on initialization and retrieval. Lines beginning with `'#'` or `';'` are ignored and may be used to provide comments.

Configuration files may include comments, prefixed by specific characters (`#` and `;`). Comments may appear on their own in an otherwise empty line, or may be entered in lines holding values or section names. In the latter case, they need to be preceded by a whitespace character to be recognized as a comment. (For backwards compatibility, only `;` starts an inline comment, while `#` does not.)

On top of the core functionality, *SafeConfigParser* supports interpolation. This means values can contain format strings which refer to other values in the same section, or values in a special `DEFAULT` section. Additional defaults can

be provided on initialization.

For example:

```
[My Section]
foodir: %(dir)s/whatever
dir=frob
long: this value continues
   in the next line
```

would resolve the `%(dir)s` to the value of `dir` (`frob` in this case). All reference expansions are done on demand.

Default values can be specified by passing them into the *ConfigParser* constructor as a dictionary. Additional defaults may be passed into the `get()` method which will override all others.

Sections are normally stored in a built-in dictionary. An alternative dictionary type can be passed to the *ConfigParser* constructor. For example, if a dictionary type is passed that sorts its keys, the sections will be sorted on write-back, as will be the keys within each section.

class ConfigParser.**RawConfigParser**([*defaults*[, *dict_type*[, *allow_no_value*]]])

The basic configuration object. When *defaults* is given, it is initialized into the dictionary of intrinsic defaults. When *dict_type* is given, it will be used to create the dictionary objects for the list of sections, for the options within a section, and for the default values. When *allow_no_value* is true (default: `False`), options without values are accepted; the value presented for these is `None`.

This class does not support the magical interpolation behavior.

All option names are passed through the *optionxform()* method. Its default implementation converts option names to lower case.

2.3 新版功能.

在 2.6 版更改: *dict_type* was added.

在 2.7 版更改: The default *dict_type* is *collections.OrderedDict*. *allow_no_value* was added.

class ConfigParser.**ConfigParser**([*defaults*[, *dict_type*[, *allow_no_value*]]])

Derived class of *RawConfigParser* that implements the magical interpolation feature and adds optional arguments to the *get()* and *items()* methods. The values in *defaults* must be appropriate for the `%()s` string interpolation. Note that *__name__* is an intrinsic default; its value is the section name, and will override any value provided in *defaults*.

All option names used in interpolation will be passed through the `optionxform()` method just like any other option name reference. Using the default implementation of `optionxform()`, the values `foo %(bar)s` and `foo %(BAR)s` are equivalent.

2.3 新版功能.

在 2.6 版更改: *dict_type* was added.

在 2.7 版更改: The default *dict_type* is *collections.OrderedDict*. *allow_no_value* was added.

class ConfigParser.**SafeConfigParser**([*defaults*[, *dict_type*[, *allow_no_value*]]])

Derived class of *ConfigParser* that implements a more-sane variant of the magical interpolation feature. This implementation is more predictable as well. New applications should prefer this version if they don't need to be compatible with older versions of Python.

2.3 新版功能.

在 2.6 版更改: *dict_type* was added.

在 2.7 版更改: The default *dict_type* is *collections.OrderedDict*. *allow_no_value* was added.

exception ConfigParser.**Error**
Base class for all other configparser exceptions.

exception ConfigParser.**NoSectionError**
当找不到指定节时引发的异常。

exception ConfigParser.**DuplicateSectionError**
Exception raised if `add_section()` is called with the name of a section that is already present.

exception ConfigParser.**NoOptionError**
Exception raised when a specified option is not found in the specified section.

exception ConfigParser.**InterpolationError**
当执行字符串插值发生问题时所引发的异常的基类。

exception ConfigParser.**InterpolationDepthError**
Exception raised when string interpolation cannot be completed because the number of iterations exceeds *MAX_INTERPOLATION_DEPTH*. Subclass of *InterpolationError*.

exception ConfigParser.**InterpolationMissingOptionError**
当从某个值引用的选项并不存在时引发的异常。为*InterpolationError* 的子类。

2.3 新版功能.

exception ConfigParser.**InterpolationSyntaxError**
Exception raised when the source text into which substitutions are made does not conform to the required syntax. Subclass of *InterpolationError*.

2.3 新版功能.

exception ConfigParser.**MissingSectionHeaderError**
当尝试解析一个不带节标头的文件时引发的异常。

exception ConfigParser.**ParsingError**
当尝试解析一个文件而发生错误时引发的异常。

ConfigParser.**MAX_INTERPOLATION_DEPTH**
The maximum depth for recursive interpolation for `get()` when the *raw* parameter is false. This is relevant only for the *ConfigParser* class.

参见:

模块*shlex* Support for a creating Unix shell-like mini-languages which can be used as an alternate format for application configuration files.

13.2.1 RawConfigParser 对象

RawConfigParser instances have the following methods:

RawConfigParser.**defaults**()
返回包含实例范围内默认值的字典。

RawConfigParser.**sections**()
Return a list of the sections available; `DEFAULT` is not included in the list.

RawConfigParser.**add_section**(*section*)
Add a section named *section* to the instance. If a section by the given name already exists, *DuplicateSectionError* is raised. If the name `DEFAULT` (or any of it's case-insensitive variants) is passed, *ValueError* is raised.

RawConfigParser.**has_section**(*section*)
Indicates whether the named section is present in the configuration. The `DEFAULT` section is not acknowledged.

RawConfigParser.**options**(*section*)
Returns a list of options available in the specified *section*.

RawConfigParser.**has_option**(*section*, *option*)
If the given section exists, and contains the given option, return `True`; otherwise return `False`.

1.6 新版功能.

RawConfigParser.**read**(*filenames*)
Attempt to read and parse a list of filenames, returning a list of filenames which were successfully parsed. If *filenames* is a string or Unicode string, it is treated as a single filename. If a file named in *filenames* cannot be opened, that file will be ignored. This is designed so that you can specify a list of potential configuration file locations (for example, the current directory, the user's home directory, and some system-wide directory), and all existing configuration files in the list will be read. If none of the named files exist, the `ConfigParser` instance will contain an empty dataset. An application which requires initial values to be loaded from a file should load the required file or files using `readfp()` before calling `read()` for any optional files:

```
import ConfigParser, os

config = ConfigParser.ConfigParser()
config.readfp(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')])
```

在 2.4 版更改: Returns list of successfully parsed filenames.

RawConfigParser.**readfp**(*fp*[, *filename*])
Read and parse configuration data from the file or file-like object in *fp* (only the `readline()` method is used). If *filename* is omitted and *fp* has a `name` attribute, that is used for *filename*; the default is `<???>`.

RawConfigParser.**get**(*section*, *option*)
Get an *option* value for the named *section*.

RawConfigParser.**getint**(*section*, *option*)
A convenience method which coerces the *option* in the specified *section* to an integer.

RawConfigParser.**getfloat**(*section*, *option*)
A convenience method which coerces the *option* in the specified *section* to a floating point number.

RawConfigParser.**getboolean**(*section*, *option*)
A convenience method which coerces the *option* in the specified *section* to a Boolean value. Note that the accepted values for the option are `"1"`, `"yes"`, `"true"`, and `"on"`, which cause this method to return `True`, and `"0"`, `"no"`, `"false"`, and `"off"`, which cause it to return `False`. These string values are checked in a case-insensitive manner. Any other value will cause it to raise `ValueError`.

RawConfigParser.**items**(*section*)
Return a list of `(name, value)` pairs for each option in the given *section*.

RawConfigParser.**set**(*section*, *option*, *value*)
If the given section exists, set the given option to the specified value; otherwise raise `NoSectionError`. While it is possible to use `RawConfigParser` (or `ConfigParser` with *raw* parameters set to true) for *internal* storage of non-string values, full functionality (including interpolation and output to files) can only be achieved using string values.

1.6 新版功能.

RawConfigParser.**write**(*fileobject*)
Write a representation of the configuration to the specified file object. This representation can be parsed by a future `read()` call.

1.6 新版功能.

RawConfigParser.**remove_option**(*section*, *option*)
Remove the specified *option* from the specified *section*. If the section does not exist, raise *NoSectionError*. If the option existed to be removed, return *True*; otherwise return *False*.

1.6 新版功能.

RawConfigParser.**remove_section**(*section*)
Remove the specified *section* from the configuration. If the section in fact existed, return `True`. Otherwise return `False`.

RawConfigParser.**optionxform**(*option*)
Transforms the option name *option* as found in an input file or as passed in by client code to the form that should be used in the internal structures. The default implementation returns a lower-case version of *option*; subclasses may override this or client code can set an attribute of this name on instances to affect this behavior.

You don't necessarily need to subclass a ConfigParser to use this method, you can also re-set it on an instance, to a function that takes a string argument. Setting it to `str`, for example, would make option names case sensitive:

```
cfgparser = ConfigParser()
...
cfgparser.optionxform = str
```

Note that when reading configuration files, whitespace around the option names are stripped before *optionxform()* is called.

13.2.2 ConfigParser 对象

The *ConfigParser* class extends some methods of the *RawConfigParser* interface, adding some optional arguments.

ConfigParser.**get**(*section*, *option*[, *raw*[, *vars*]])
Get an *option* value for the named *section*. If *vars* is provided, it must be a dictionary. The *option* is looked up in *vars* (if provided), *section*, and in *defaults* in that order.

所有 `'%'` 插值会在返回值中被展开，除非 *raw* 参数为真值。插值键所使用的值会按与选项相同的方式来查找。

ConfigParser.**items**(*section*[, *raw*[, *vars*]])
Return a list of `(name, value)` pairs for each option in the given *section*. Optional arguments have the same meaning as for the *get()* method.

2.3 新版功能.

13.2.3 SafeConfigParser Objects

The *SafeConfigParser* class implements the same extended interface as *ConfigParser*, with the following addition:

SafeConfigParser.**set**(*section*, *option*, *value*)
If the given section exists, set the given option to the specified value; otherwise raise *NoSectionError*. *value* must be a string (*str* or *unicode*); if not, *TypeError* is raised.

2.4 新版功能.

13.2.4 Examples

一个写入配置文件的示例:

```
import ConfigParser

config = ConfigParser.RawConfigParser()

# When adding sections or items, add them in the reverse order of
# how you want them to be displayed in the actual file.
# In addition, please note that using RawConfigParser's and the raw
# mode of ConfigParser's respective set functions, you can assign
# non-string values to keys internally, but will receive an error
# when attempting to write to a file or when you get it in non-raw
# mode. SafeConfigParser does not allow such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'wb') as configfile:
    config.write(configfile)
```

一个再次读取配置文件的示例:

```
import ConfigParser

config = ConfigParser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective types
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print a_float + an_int

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print config.get('Section1', 'foo')
```

To get interpolation, you will need to use a *ConfigParser* or *SafeConfigParser*:

```
import ConfigParser

config = ConfigParser.ConfigParser()
config.read('example.cfg')

# Set the third, optional argument of get to 1 if you wish to use raw mode.
print config.get('Section1', 'foo', 0)  # -> "Python is fun!"
print config.get('Section1', 'foo', 1)  # -> "%(bar)s is %(baz)s!"

# The optional fourth argument is a dict with members that will take
# precedence in interpolation.
```

(下页继续)

(续上页)

```
print config.get('Section1', 'foo', 0, {'bar': 'Documentation',
                                        'baz': 'evil'})
```

Defaults are available in all three types of ConfigParsers. They are used in interpolation if an option used is not defined elsewhere.

```
import ConfigParser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
config = ConfigParser.SafeConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print config.get('Section1', 'foo')  # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print config.get('Section1', 'foo')  # -> "Life is hard!"
```

The function `opt_move` below can be used to move options between sections:

```
def opt_move(config, section1, section2, option):
    try:
        config.set(section2, option, config.get(section1, option, 1))
    except ConfigParser.NoSectionError:
        # Create non-existent section
        config.add_section(section2)
        opt_move(config, section1, section2, option)
    else:
        config.remove_option(section1, option)
```

Some configuration files are known to include settings without values, but which otherwise conform to the syntax supported by *ConfigParser*. The *allow_no_value* parameter to the constructor can be used to indicate that such values should be accepted:

```
>>> import ConfigParser
>>> import io

>>> sample_config = """
... [mysqld]
... user = mysql
... pid-file = /var/run/mysqld/mysqld.pid
... skip-external-locking
... old_passwords = 1
... skip-bdb
... skip-innodb
... """
>>> config = ConfigParser.RawConfigParser(allow_no_value=True)
>>> config.readfp(io.BytesIO(sample_config))

>>> # Settings with values are treated as before:
>>> config.get("mysqld", "user")
'mysql'

>>> # Settings without values provide None:
>>> config.get("mysqld", "skip-bdb")

>>> # Settings which aren't specified still raise an error:
```

(下页继续)

(续上页)

```
>>> config.get("mysqld", "does-not-exist")
Traceback (most recent call last):
  ...
ConfigParser.NoOptionError: No option 'does-not-exist' in section: 'mysqld'
```

13.3 robotparser —Parser for robots.txt

注解: The *robotparser* module has been renamed `urllib.robotparser` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

This module provides a single class, *RobotFileParser*, which answers questions about whether or not a particular user agent can fetch a URL on the Web site that published the `robots.txt` file. For more details on the structure of `robots.txt` files, see http://www.robotstxt.org/orig.html.

class robotparser.**RobotFileParser**(*url=''*)

This class provides methods to read, parse and answer questions about the `robots.txt` file at *url*.

set_url(*url*)

Sets the URL referring to a `robots.txt` file.

read()

Reads the `robots.txt` URL and feeds it to the parser.

parse(*lines*)

Parses the lines argument.

can_fetch(*useragent*, *url*)

Returns `True` if the *useragent* is allowed to fetch the *url* according to the rules contained in the parsed `robots.txt` file.

mtime()

Returns the time the `robots.txt` file was last fetched. This is useful for long-running web spiders that need to check for new `robots.txt` files periodically.

modified()

Sets the time the `robots.txt` file was last fetched to the current time.

The following example demonstrates basic use of the RobotFileParser class.

```
>>> import robotparser
>>> rp = robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

13.4 netrc —netrc 文件处理

1.5.2 新版功能.

源代码: Lib/netrc.py

---

netrc 类解析并封装了 Unix 的 **ftp** 程序和其他 FTP 客户端所使用的 netrc 文件格式。

class netrc.**netrc**([*file*])

A *netrc* instance or subclass instance encapsulates data from a netrc file. The initialization argument, if present, specifies the file to parse. If no argument is given, the file `.netrc` in the user' s home directory will be read. Parse errors will raise *NetrcParseError* with diagnostic information including the file name, line number, and terminating token. If no argument is specified on a POSIX system, the presence of passwords in the `.netrc` file will raise a *NetrcParseError* if the file ownership or permissions are insecure (owned by a user other than the user running the process, or accessible for read or write by any other user). This implements security behavior equivalent to that of ftp and other programs that use `.netrc`.

在 2.7.6 版更改: Added the POSIX permissions check.

exception netrc.**NetrcParseError**

当在源文本中遇到语法错误时由*netrc* 类引发的异常。此异常的实例提供了三个有用属性: `msg` 为错误的文本说明，`filename` 为源文件的名称，而 `lineno` 给出了错误所在的行号。

13.4.1 netrc 对象

netrc 实例具有下列方法:

netrc.**authenticators**(*host*)

针对 *host* 的身份验证者返回一个 3 元组 `(login, account, password)`。如果 netrc 文件不包含针对给定主机的条目，则返回关联到‘default’条目的元组。如果匹配的主机或默认条目均不可用，则返回 `None`。

netrc.**__repr__**()

将类数据以 netrc 文件的格式转储为一个字符串。(这会丢弃注释并可能重排条目顺序。)

netrc 的实例具有一些公共实例变量:

netrc.**hosts**

将主机名映射到 `(login, account, password)` 元组的字典。如果存在‘default’条目，则会表示为使用该名称的伪主机。

netrc.**macros**

将宏名称映射到字符串列表的字典。

注解: Passwords are limited to a subset of the ASCII character set. Versions of this module prior to 2.3 were extremely limited. Starting with 2.3, all ASCII punctuation is allowed in passwords. However, note that whitespace and non-printable characters are not allowed in passwords. This is a limitation of the way the .netrc file is parsed and may be removed in the future.

13.5 xdrlib —编码与解码 XDR 数据

源代码: Lib/xdrlib.py

---

xdrlib 模块为外部数据表示标准提供支持，该标准的描述见 **RFC 1014**，由 Sun Microsystems, Inc. 在 1987 年 6 月撰写。它支持该 RFC 中描述的大部分数据类型。

xdrlib 模块定义了两个类，一个用于将变量打包为 XDR 表示形式，另一个用于从 XDR 表示形式解包。此外还有两个异常类。

class xdrlib.**Packer**
Packer 是用于将数据打包为 XDR 表示形式的类。*Packer* 类的实例化不附带参数。

class xdrlib.**Unpacker**(*data*)
Unpacker 是用于相应地从字符串缓冲区解包 XDR 数据值的类。输入缓冲区将作为 *data* 给出。

参见:

RFC 1014 - XDR: 外部数据表示标准 这个 RFC 定义了最初编写此模块时 XDR 所用的数据编码格式。显然它已被 **RFC 1832** 所淘汰。

RFC 1832 - XDR: 外部数据表示标准 更新的 RFC，它提供了经修订的 XDR 定义。

13.5.1 Packer 对象

Packer 实例具有下列方法:

Packer.**get_buffer**()
将当前打包缓冲区以字符串的形式返回。

Packer.**reset**()
将打包缓冲区重置为空字符串。

总体来说，你可以通过调用适当的 pack_type() 方法来打包任何最常见的 XDR 数据类型。每个方法都是接受单个参数，即要打包的值。受支持的简单数据类型打包方法如下: pack_uint(), pack_int(), pack_enum(), pack_bool(), pack_uhyper() 以及 pack_hyper()。

Packer.**pack_float**(*value*)
打包单精度浮点数 *value*。

Packer.**pack_double**(*value*)
打包双精度浮点数 *value*。

以下方法支持打包字符串、字节串以及不透明数据。

Packer.**pack_fstring**(*n*, *s*)
打包固定长度字符串 *s*。*n* 为字符串的长度，但它 不会被打包进数据缓冲区。如有必要字符串会以空字节串填充以保证 4 字节对齐。

Packer.**pack_fopaque**(*n*, *data*)
打包固定长度不透明数据流，类似于*pack_fstring()*。

Packer.**pack_string**(*s*)
打包可变长度字符串 *s*。先将字符串的长度打包为无符号整数，再用*pack_fstring()* 来打包字符串数据。

Packer.**pack_opaque**(*data*)
打包可变长度不透明数据流，类似于*pack_string()*。

Packer.**pack_bytes**(*bytes*)
打包可变长度字节流，类似于*pack_string()*。

下列方法支持打包数组和列表:

Packer.**pack_list**(*list*, *pack_item*)
打包由同质条目构成的 *list*。此方法适用于不确定长度的列表；即其长度无法在遍历整个列表之前获知。对于列表中的每个条目，先打包一个无符号整数 `1`，再添加列表中数据的值。*pack_item* 是在打包单个条目时要调用的函数。在列表的末尾，会再打包一个无符号整数 `0`.

例如，要打包一个整数列表，代码看起来会是这样:

```
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

Packer.**pack_farray**(*n*, *array*, *pack_item*)
打包由同质条目构成的固定长度列表 (*array*)。*n* 为列表长度；它 不会被打包到缓冲区，但是如果 `len(array)` 不等于 *n* 则会引发*ValueError*。如上所述，*pack_item* 是在打包每个元素时要使用的函数。

Packer.**pack_array**(*list*, *pack_item*)
打包由同质条目构成的可变长度 *list*。先将列表的长度打包为无符号整数，再像上面的*pack_farray()*一样打包每个元素。

13.5.2 Unpacker 对象

Unpacker 类提供以下方法:

Unpacker.**reset**(*data*)
使用给定的 *data* 重置字符串缓冲区。

Unpacker.**get_position**()
返回数据缓冲区中的当前解包位置。

Unpacker.**set_position**(*position*)
将数据缓冲区的解包位置设为 *position*。你应当小心使用*get_position()* 和*set_position()*。

Unpacker.**get_buffer**()
将当前解包数据缓冲区以字符串的形式返回。

Unpacker.**done**()
表明解包完成。如果数据没有全部完成解包则会引发*Error* 异常。

此外，每种可通过*Packer* 打包的数据类型都可通过*Unpacker* 来解包。解包方法的形式为 `unpack_type()`，并且不接受任何参数。该方法将返回解包后的对象。

Unpacker.**unpack_float**()
解包单精度浮点数。

Unpacker.**unpack_double**()
解包双精度浮点数，类似于*unpack_float()*。

此外，以下方法可用来解包字符串、字节串以及不透明数据:

Unpacker.**unpack_fstring**(*n*)
解包并返回固定长度字符串。*n* 为期望的字符数量。会预设以空字节串填充以保证 4 字节对齐。

Unpacker.**unpack_fopaque**(*n*)
解包并返回固定长度数据流，类似于*unpack_fstring()*。

Unpacker.**unpack_string**()
解包并返回可变长度字符串。先将字符串的长度解包为无符号整数，再用*unpack_fstring()* 来解包字符串数据。

Unpacker.**unpack_opaque**()
解包并返回可变长度不透明数据流，类似于*unpack_string()*。

Unpacker.**unpack_bytes**()
解包并返回可变长度字节流，类似于*unpack_string()*。

下列方法支持解包数组和列表:

Unpacker.**unpack_list**(*unpack_item*)
解包并返回同质条目的列表。该列表每次解包一个元素，先解包一个无符号整数旗标。如果旗标为 1，则解包条目并将其添加到列表。旗标为 0 表明列表结束。*unpack_item* 为在解包条目时调用的函数。

Unpacker.**unpack_farray**(*n*, *unpack_item*)
解包并（以列表形式）返回由同质条目构成的固定长度数组。*n* 为期望的缓冲区内列表元素数量。如上所述，*unpack_item* 是解包每个元素时要使用的函数。

Unpacker.**unpack_array**(*unpack_item*)
解包并返回由同质条目构成的可变长度 *list*。先将列表的长度解包为无符号整数，再像上面的*unpack_farray()* 一样解包每个元素。

13.5.3 异常

此模块中的异常会表示为类实例代码:

exception xdrlib.**Error**
基本异常类。*Error* 具有一个公共属性 msg，其中包含对错误的描述。

exception xdrlib.**ConversionError**
从*Error* 所派生的类。不包含额外的实例变量。

以下是一个应该如何捕获这些异常的示例:

```
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError as instance:
    print 'packing the double failed:', instance.msg
```

13.6 plistlib —生成与解析 Mac OS X .plist 文件

在 2.6 版更改: This module was previously only available in the Mac-specific library, it is now available for all platforms.

源代码: Lib/plistlib.py

This module provides an interface for reading and writing the "property list" XML files used mainly by Mac OS X.

The property list (.plist) file format is a simple XML pickle supporting basic object types, like dictionaries, lists, numbers and strings. Usually the top level object is a dictionary.

Values can be strings, integers, floats, booleans, tuples, lists, dictionaries (but only with string keys), *Data* or *datetime.datetime* objects. String values (including dictionary keys) may be unicode strings –they will be written out as UTF-8.

The <data> plist type is supported through the *Data* class. This is a thin wrapper around a Python string. Use *Data* if your strings contain control characters.

参见:

PList manual page 针对该文件格式的 Apple 文档。

这个模块定义了以下函数:

plistlib.**readPlist**(*pathOrFile*)
Read a plist file. *pathOrFile* may either be a file name or a (readable) file object. Return the unpacked root object (which usually is a dictionary).

The XML data is parsed using the Expat parser from *xml.parsers.expat* –see its documentation for possible exceptions on ill-formed XML. Unknown elements will simply be ignored by the plist parser.

plistlib.**writePlist**(*rootObject*, *pathOrFile*)
Write *rootObject* to a plist file. *pathOrFile* may either be a file name or a (writable) file object.

如果对象是不受支持的类型或者是包含不受支持类型的对象的容器则将引发*TypeError*。

plistlib.**readPlistFromString**(*data*)
Read a plist from a string. Return the root object.

plistlib.**writePlistToString**(*rootObject*)
Return *rootObject* as a plist-formatted string.

plistlib.**readPlistFromResource**(*path*, *restype='plst'*, *resid=0*)
Read a plist from the resource with type *restype* from the resource fork of *path*. Availability: Mac OS X.

注解: In Python 3.x, this function has been removed.

plistlib.**writePlistToResource**(*rootObject*, *path*, *restype='plst'*, *resid=0*)
Write *rootObject* as a resource with type *restype* to the resource fork of *path*. Availability: Mac OS X.

注解: In Python 3.x, this function has been removed.

The following class is available:

class plistlib.**Data**(*data*)
Return a "data" wrapper object around the string *data*. This is used in functions converting from/to plists to represent the <data> type available in plists.

It has one attribute, data, that can be used to retrieve the Python string stored in it.

13.6.1 例子

生成一个 plist:

```
pl = dict(
    aString="Doodah",
    aList=["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict=dict(
        anotherString="<hello & hi there!>",
        aUnicodeValue=u'M\xe4ssig, Ma\xdf',
        aTrueValue=True,
        aFalseValue=False,
    ),
    someData = Data("<binary gunk>"),
    someMoreData = Data("<lots of binary gunk>" * 10),
    aDate = datetime.datetime.fromtimestamp(time.mktime(time.gmtime())),
)
# unicode keys are possible, but a little awkward to use:
pl[u'\xc5benraa'] = "That was a unicode key."
writePlist(pl, fileName)
```

解析一个 plist:

```
pl = readPlist(pathOrFile)
print pl["aKey"]
```

CHAPTER 14

加密服务

The modules described in this chapter implement various algorithms of a cryptographic nature. They are available at the discretion of the installation. Here's an overview:

14.1 hashlib —安全哈希与消息摘要

2.5 新版功能.

源码： Lib/hashlib.py

This module implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA1, SHA224, SHA256, SHA384, and SHA512 (defined in FIPS 180-2) as well as RSA's MD5 algorithm (defined in Internet **RFC 1321**). The terms secure hash and message digest are interchangeable. Older algorithms were called message digests. The modern term is secure hash.

注解： 如果你想找到 adler32 或 crc32 哈希函数，它们在*zlib* 模块中。

警告： 有些算法已知存在哈希碰撞弱点，请参考最后的“另请参阅”段。

There is one constructor method named for each type of *hash*. All return a hash object with the same simple interface. For example: use `sha1()` to create a SHA1 hash object. You can now feed this object with arbitrary strings using the `update()` method. At any point you can ask it for the *digest* of the concatenation of the strings fed to it so far using the `digest()` or `hexdigest()` methods.

Constructors for hash algorithms that are always present in this module are *md5()*, `sha1()`, `sha224()`, `sha256()`, `sha384()`, and `sha512()`. Additional algorithms may also be available depending upon the OpenSSL library that Python uses on your platform.

For example, to obtain the digest of the string `'Nobody inspects the spammish repetition'`:

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
>>> m.digest_size
16
>>> m.block_size
64
```

更简要的写法:

```
>>> hashlib.sha224("Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

A generic *new()* constructor that takes the string name of the desired algorithm as its first parameter also exists to allow access to the above listed hashes as well as any other algorithms that your OpenSSL library may offer. The named constructors are much faster than *new()* and should be preferred.

使用*new()* 并附带由 OpenSSL 所提供了算法:

```
>>> h = hashlib.new('ripemd160')
>>> h.update("Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

This module provides the following constant attribute:

hashlib.**algorithms**
: A tuple providing the names of the hash algorithms guaranteed to be supported by this module.

  2.7 新版功能.

hashlib.**algorithms_guaranteed**
: A set containing the names of the hash algorithms guaranteed to be supported by this module on all platforms.

  2.7.9 新版功能.

hashlib.**algorithms_available**
: 一个集合，其中包含在所运行的 Python 解释器上可用的哈希算法的名称。将这些名称传给*new()* 时将可被识别。*algorithms_guaranteed* 将总是它的一个子集。同样的算法在此集合中可能以不同的名称出现多次（这是 OpenSSL 的原因）。

  2.7.9 新版功能.

下列值会以构造器所返回的哈希对象的常量属性的形式被提供:

hash.**digest_size**
: 以字节表示的结果哈希对象的大小。

hash.**block_size**
: 以字节表示的哈希算法的内部块大小。

哈希对象具有下列方法:

hash.**update**(*arg*)
: Update the hash object with the string *arg*. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a+b)`.

  在 2.7 版更改: The Python GIL is released to allow other threads to run while hash updates on data larger than 2048 bytes is taking place when using hash algorithms supplied by OpenSSL.

hash.**digest**()
Return the digest of the strings passed to the *update()* method so far. This is a string of *digest_size* bytes which may contain non-ASCII characters, including null bytes.

hash.**hexdigest**()
Like *digest()* except the digest is returned as a string of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

hash.**copy**()
Return a copy ("clone") of the hash object. This can be used to efficiently compute the digests of strings that share a common initial substring.

14.1.1 密钥派生

密钥派生和密钥延展算法被设计用于安全密码哈希。`sha1(password)` 这样的简单算法无法防御暴力攻击。好的密码哈希函数必须可以微调、放慢步调，并且包含 加盐。

hashlib.**pbkdf2_hmac**(*name*, *password*, *salt*, *rounds*, *dklen=None*)
此函数提供 PKCS#5 基于密码的密钥派生函数 2。它使用 HMAC 作为伪随机函数。

The string *name* is the desired name of the hash digest algorithm for HMAC, e.g. 'sha1' or 'sha256'. *password* and *salt* are interpreted as buffers of bytes. Applications and libraries should limit *password* to a sensible value (e.g. 1024). *salt* should be about 16 or more bytes from a proper source, e.g. *os.urandom()*.

The number of *rounds* should be chosen based on the hash algorithm and computing power. As of 2013, at least 100,000 rounds of SHA-256 is suggested.

dklen is the length of the derived key. If *dklen* is `None` then the digest size of the hash algorithm *name* is used, e.g. 64 for SHA-512.

```
>>> import hashlib, binascii
>>> dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 100000)
>>> binascii.hexlify(dk)
b'0394a2ede332c9a13eb82e9b24631604c31df978b4e2f0fbd2c549944f9d79a5'
```

2.7.8 新版功能.

注解: 随同 OpenSSL 提供了一个快速的 *pbkdf2_hmac* 实现。Python 实现是使用*hmac* 的内联版本。它的速度大约要慢上三倍并且不会释放 GIL。

参见:

模块*hmac* 使用哈希运算来生成消息验证代码的模块。

模块*base64* 针对非二进制环境对二进制哈希值进行编辑的另一种方式。

http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf 有关安全哈希算法的 FIPS 180-2 出版物。

https://en.wikipedia.org/wiki/Cryptographic_hash_function#Cryptographic_hash_algorithms 包含关于哪些算法存在已知问题以及对其使用所造成的影响的信息的 Wikipedia 文章。

14.2 hmac —基于密钥的消息验证

2.2 新版功能.

源代码: Lib/hmac.py

此模块实现了 HMAC 算法，算法的描述参见 **RFC 2104**。

hmac.**new**(*key*[, *msg*[, *digestmod*]])
Return a new hmac object. If *msg* is present, the method call `update(msg)` is made. *digestmod* is the digest constructor or module for the HMAC object to use. It defaults to the `hashlib.md5` constructor.

HMAC 对象具有下列方法:

HMAC.**update**(*msg*)
Update the hmac object with the string *msg*. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a + b)`.

HMAC.**digest**()
Return the digest of the strings passed to the *update()* method so far. This string will be the same length as the *digest_size* of the digest given to the constructor. It may contain non-ASCII characters, including NUL bytes.

> **警告:** 在验证例程运行期间将*digest()* 的输出与外部提供的摘要进行比较时，建议使用*compare_digest()* 函数而不是 == 运算符以减少定时攻击防御力的不足。

HMAC.**hexdigest**()
类似于*digest()* 但摘要会以两倍长度字符串的形式返回，其中仅包含十六进制数码。这可以被用于在电子邮件或其他非二进制环境中安全地交换数据值。

> **警告:** 在验证例程运行期间将*hexdigest()* 的输出与外部提供的摘要进行比较时，建议使用*compare_digest()* 函数而不是 == 运算符以减少定时攻击防御力的不足。

HMAC.**copy**()
返回 hmac 对象的副本（“克隆)。这可被用来高效地计算共享相同初始子串的数据的摘要。

这个模块还提供了下列辅助函数:

hmac.**compare_digest**(*a*, *b*)
Return `a == b`. This function uses an approach designed to prevent timing analysis by avoiding content-based short circuiting behaviour, making it appropriate for cryptography. *a* and *b* must both be of the same type: either *unicode* or a *bytes-like object*.

> **注解:** 如果 *a* 和 *b* 具有不同的长度，或者如果发生了错误，定时攻击在理论上可以获取有关 *a* 和 *b* 的类型和长度信息—但不能获取它们的值。

2.7.7 新版功能.

参见:

模块*hashlib* 提供安全哈希函数的 Python 模块。

14.3 md5 —MD5 message digest algorithm

2.5 版后已移除: Use the *hashlib* module instead.

This module implements the interface to RSA's MD5 message digest algorithm (see also Internet **RFC 1321**). Its use is quite straightforward: use *new()* to create an md5 object. You can now feed this object with arbitrary strings using the `update()` method, and at any point you can ask it for the *digest* (a strong kind of 128-bit checksum, a.k.a. "fingerprint") of the concatenation of the strings fed to it so far using the `digest()` method.

For example, to obtain the digest of the string `'Nobody inspects the spammish repetition'`:

```
>>> import md5
>>> m = md5.new()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
```

More condensed:

```
>>> md5.new("Nobody inspects the spammish repetition").digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
```

The following values are provided as constants in the module and as attributes of the md5 objects returned by *new()*:

`md5.`**digest_size**
The size of the resulting digest in bytes. This is always `16`.

The md5 module provides the following functions:

`md5.`**new**([*arg*])
Return a new md5 object. If *arg* is present, the method call `update(arg)` is made.

`md5.`**md5**([*arg*])
For backward compatibility reasons, this is an alternative name for the *new()* function.

An md5 object has the following methods:

`md5.`**update**(*arg*)
Update the md5 object with the string *arg*. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a+b)`.

`md5.`**digest**()
Return the digest of the strings passed to the *update()* method so far. This is a 16-byte string which may contain non-ASCII characters, including null bytes.

`md5.`**hexdigest**()
Like *digest()* except the digest is returned as a string of length 32, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

`md5.`**copy**()
Return a copy ("clone") of the md5 object. This can be used to efficiently compute the digests of strings that share a common initial substring.

参见:

Module *sha* Similar module implementing the Secure Hash Algorithm (SHA). The SHA algorithm is considered a more secure hash.

14.4 sha —SHA-1 message digest algorithm

2.5 版后已移除: Use the *hashlib* module instead.

This module implements the interface to NIST's secure hash algorithm, known as SHA-1. SHA-1 is an improved version of the original SHA hash algorithm. It is used in the same way as the *md5* module: use *new()* to create an sha object, then feed this object with arbitrary strings using the `update()` method, and at any point you can ask it for the *digest* of the concatenation of the strings fed to it so far. SHA-1 digests are 160 bits instead of MD5's 128 bits.

`sha.`**`new`**`(`[*string*]`)`
Return a new sha object. If *string* is present, the method call `update(string)` is made.

The following values are provided as constants in the module and as attributes of the sha objects returned by *new()*:

`sha.`**`blocksize`**
Size of the blocks fed into the hash function; this is always `1`. This size is used to allow an arbitrary string to be hashed.

`sha.`**`digest_size`**
The size of the resulting digest in bytes. This is always `20`.

An sha object has the same methods as md5 objects:

`sha.`**`update`**`(`*arg*`)`
Update the sha object with the string *arg*. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a+b)`.

`sha.`**`digest`**`()`
Return the digest of the strings passed to the *update()* method so far. This is a 20-byte string which may contain non-ASCII characters, including null bytes.

`sha.`**`hexdigest`**`()`
Like *digest()* except the digest is returned as a string of length 40, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

`sha.`**`copy`**`()`
Return a copy ("clone") of the sha object. This can be used to efficiently compute the digests of strings that share a common initial substring.

参见:

Secure Hash Standard The Secure Hash Algorithm is defined by NIST document FIPS PUB 180-2: Secure Hash Standard, published in August 2002.

Cryptographic Toolkit (Secure Hashing) Links from NIST to various information on secure hashing.

CHAPTER 15

通用操作系统服务

本章中描述的各模块提供了在（几乎）所有的操作系统上可用的操作系统特性的接口，例如文件和时钟。这些接口通常以 Unix 或 C 接口为参考对象，不过在大多数其他系统上也可用。这里有一个概述：

15.1 os —操作系统接口模块

该模块提供了一些方便使用操作系统相关功能的函数。如果你是想读写一个文件，请参阅*open()*，如果你想操作路径，请参阅*os.path* 模块，如果你想在命令行上读取所有文件中的所有行请参阅*fileinput* 模块。有关创建临时文件和目录的方法，请参阅*tempfile* 模块，对于高级文件目录处理，请参阅*shutil* 模块。

关于这些函数的适用性的说明：

- 所有 Python 内建的操作系统相关的模块的设计都是为了使得在同一功能可用的情况下，保持接口的一致性；例如，函数 `os.stat(path)` 以相同的格式返回关于 *path* 的统计信息（这个函数同时也是起源于 POSIX 接口）。
- 针对特定的操作的拓展同样在可用于*os* 模块，但是使用它们必然会对可移植性产生威胁。
- An “Availability: Unix” note means that this function is commonly found on Unix systems. It does not make any claims about its existence on a specific operating system.
- If not separately noted, all functions that claim “Availability: Unix” are supported on Mac OS X, which builds on a Unix core.

注解： All functions in this module raise *OSError* in the case of invalid or inaccessible file names and paths, or other arguments that have the correct type, but are not accepted by the operating system.

exception `os.`**error**
内建的*OSError* 异常的一个别名。

os.**name**
The name of the operating system dependent module imported. The following names have currently been registered: 'posix', 'nt', 'os2', 'ce', 'java', 'riscos'.

参见:

sys.platform 有更详细的描述. *os.uname()* 只给出系统提供的版本信息。

platform 模块对系统的标识有更详细的检查。

15.1.1 进程参数

这些函数和数据项提供了操作当前进程和用户的信息。

os.**environ**
一个表示字符串环境的*mapping* 对象。例如，environ['HOME'] 是你的主目录（在某些平台上）的路径名，相当于 C 中的 getenv("HOME")。

这个映射是在第一次导入*os* 模块时捕获的，通常作为 Python 启动时处理 site.py 的一部分。除了通过直接修改 os.environ 之外，在此之后对环境所做的更改不会反映在 os.environ 中。

如果平台支持*putenv()* 函数，这个映射除了可以用于查询环境外还能用于修改环境。当这个映射被修改时，*putenv()* 将被自动调用。

注解: 直接调用*putenv()* 并不会影响 os.environ，所以推荐直接修改 “os.environ“。

注解: 在某些平台上，包括 FreeBSD 和 Mac OS X，设置 environ 可能导致内存泄露。参阅 putenv() 的系统文档。

如果平台没有提供*putenv()*，为了使启动的子进程使用修改后的环境，一份修改后的映射会被传给合适的进程创建函数。

如果平台支持*unsetenv()* 函数，你可以通过删除映射中元素的方式来删除对应的环境变量。当一个元素被从 os.environ 删除时，以及 pop() 或 clear() 被调用时，*unsetenv()* 会被自动调用。

在 2.6 版更改: Also unset environment variables when calling os.environ.clear() and os.environ.pop().

os.**chdir**(*path*)
os.**fchdir**(*fd*)
os.**getcwd**()
以上函数请参阅文件和目录 。

os.**ctermid**()
返回与进程控制终端对应的文件名。

Availability: Unix.

os.**getegid**()
返回当前进程的有效组 ID。对应当前进程执行文件的 “set id” 位。

Availability: Unix.

os.**geteuid**()
返回当前进程的有效用户 ID。

Availability: Unix.

os.**getgid**()
返回当前进程的实际组 ID。

Availability: Unix.

os.**getgroups**()
返回当前进程关联的附加组 ID 列表

Availability: Unix.

注解: 在 Mac OS X 系统中，*getgroups()* 会和其他 Unix 平台有些不同。如果 Python 解释器是在 10.5 或更早版本中部署，*getgroups()* 返回当前用户进程相关的有效组 ID 列表。该列表长度由于系统预设的接口限制，最长为 16。而且在适当的权限下，返回结果还会因*getgroups()* 而发生变化；如果 Python 解释器是在 10.5 以上版本中部署，*getgroups()* 返回进程所属有效用户 ID 所对应的用户的组 ID 列表，组用户列表可能因为进程的生存周期而发生变动，而且也不会因为*setgroups()* 的调用而发生，返回的组用户列表长度也没有长度 16 的限制。在部署中，Python 解释器用到的变量 MACOSX_DEPLOYMENT_TARGET 可以用*sysconfig.get_config_var()*。

---

os.**initgroups**(*username*, *gid*)
调用系统 initgroups()，使用指定用户所在的所有值来初始化组访问列表，包括指定的组 ID。

Availability: Unix.

2.7 新版功能.

os.**getlogin**()
Return the name of the user logged in on the controlling terminal of the process. For most purposes, it is more useful to use the environment variable LOGNAME to find out who the user is, or pwd.getpwuid(os.getuid())[0] to get the login name of the process' s real user id.

Availability: Unix.

os.**getpgid**(*pid*)
根据进程 id *pid* 返回进程的组 ID 列表。如果 *pid* 为 0，则返回当前进程的进程组 ID 列表

Availability: Unix.

2.3 新版功能.

os.**getpgrp**()
返回当时进程组的 ID

Availability: Unix.

os.**getpid**()
返回当前进程 ID

Availability: Unix, Windows.

os.**getppid**()
Return the parent' s process id.

Availability: Unix.

os.**getresuid**()
返回一个由 (ruid, euid, suid) 所组成的元组，分别表示当前进程的真实用户 ID，有效用户 ID 和暂存用户 ID。

Availability: Unix.

2.7 新版功能.

os.**getresgid**()
返回一个由 (rgid, egid, sgid) 所组成的元组，分别表示当前进程的真实组 ID，有效组 ID 和暂存组 ID。

Availability: Unix.

2.7 新版功能.

os.**getuid**()
返回当前进程的真实用户 ID。

Availability: Unix.

os.**getenv**(*varname*[, *value*])
Return the value of the environment variable *varname* if it exists, or *value* if it doesn't. *value* defaults to `None`.

Availability: most flavors of Unix, Windows.

os.**putenv**(*varname*, *value*)
Set the environment variable named *varname* to the string *value*. Such changes to the environment affect subprocesses started with *os.system()*, *popen()* or *fork()* and *execv()*.

Availability: most flavors of Unix, Windows.

注解: 在一些平台，包括 FreeBSD 和 Mac OS X，设置 `environ` 可能导致内存泄露。详情参考 putenv 相关系统文档。

当系统支持*putenv()* 时，`os.environ` 中的参数赋值会自动转换为对*putenv()* 的调用。不过*putenv()* 的调用不会更新 `os.environ`，因此最好使用 `os.environ` 对变量赋值。

os.**setegid**(*egid*)
设置当前进程的有效组 ID。

Availability: Unix.

os.**seteuid**(*euid*)
设置当前进程的有效用户 ID。

Availability: Unix.

os.**setgid**(*gid*)
设置当前进程的组 ID。

Availability: Unix.

os.**setgroups**(*groups*)
将 *group* 参数值设置为与当进程相关联的附加组 ID 列表。*group* 参数必须为一个序列，每个元素应为每个组的数字 ID。该操作通常只适用于超级用户。

Availability: Unix.

2.2 新版功能.

注解: 在 Mac OS X 中，*groups* 的长度不能超过系统定义的最大有效组 ID 个数，一般为 16。如果它没有返回与调用 setgroups() 所设置的相同的组列表，请参阅*getgroups()* 的文档。

---

os.**setpgrp**()
根据已实现的版本（如果有）来调用系统 `setpgrp()` 或 `setpgrp(0, 0)` 。相关说明，请参考 Unix 手册。

Availability: Unix.

os.**setpgid**(*pid*, *pgrp*)
使用系统调用 setpgid()，将 *pid* 对应进程的组 ID 设置为 *pgrp*。相关说明，请参考 Unix 手册。

Availability: Unix.

os.**setregid**(*rgid*, *egid*)
设置当前进程的真实和有效组 ID。

Availability: Unix.

os.**setresgid**(*rgid*, *egid*, *sgid*)
设置当前进程的真实，有效和暂存组 ID。

Availability: Unix.

2.7 新版功能.

os.**setresuid**(*ruid*, *euid*, *suid*)
设置当前进程的真实，有效和暂存用户 ID。

Availability: Unix.

2.7 新版功能.

os.**setreuid**(*ruid*, *euid*)
设置当前进程的真实和有效用户 ID。

Availability: Unix.

os.**getsid**(*pid*)
调用系统调用 getsid()。相关语义请参阅 Unix 手册。

Availability: Unix.

2.4 新版功能.

os.**setsid**()
使用系统调用 getsid()。相关说明，请参考 Unix 手册。

Availability: Unix.

os.**setuid**(*uid*)
设置当前进程的用户 ID。

Availability: Unix.

os.**strerror**(*code*)
根据 *code* 中的错误码返回错误消息。在某些平台上当给出未知错误码时 strerror() 将返回 NULL 并会引发*ValueError*。

Availability: Unix, Windows.

os.**umask**(*mask*)
设定当前数值掩码并返回之前的掩码。

Availability: Unix, Windows.

os.**uname**()
Return a 5-tuple containing information identifying the current operating system. The tuple contains 5 strings: (sysname, nodename, release, version, machine). Some systems truncate the nodename to 8 characters or to the leading component; a better way to get the hostname is *socket.gethostname()* or even socket.gethostbyaddr(socket.gethostname()).

Availability: recent flavors of Unix.

os.**unsetenv**(*varname*)
Unset (delete) the environment variable named *varname*. Such changes to the environment affect subprocesses started with *os.system()*, *popen()* or *fork()* and *execv()*.

当系统支持*unsetenv()*，删除在 os.environ 中的变量会自动转换为对*unsetenv()* 的调用。但是*unsetenv()* 不能更新 os.environ，因此最好直接删除 os.environ 中的变量。

Availability: most flavors of Unix, Windows.

15.1.2 创建文件对象

These functions create new file objects. (See also *open()*.)

os.**fdopen**(*fd*[, *mode*[, *bufsize*]])
Return an open file object connected to the file descriptor *fd*. The *mode* and *bufsize* arguments have the same meaning as the corresponding arguments to the built-in *open()* function. If *fdopen()* raises an exception, it leaves *fd* untouched (unclosed).

Availability: Unix, Windows.

在 2.3 版更改: When specified, the *mode* argument must now start with one of the letters 'r', 'w', or 'a', otherwise a *ValueError* is raised.

在 2.5 版更改: On Unix, when the *mode* argument starts with 'a', the *O_APPEND* flag is set on the file descriptor (which the fdopen() implementation already does on most platforms).

os.**popen**(*command*[, *mode*[, *bufsize*]])
Open a pipe to or from *command*. The return value is an open file object connected to the pipe, which can be read or written depending on whether *mode* is 'r' (default) or 'w'. The *bufsize* argument has the same meaning as the corresponding argument to the built-in *open()* function. The exit status of the command (encoded in the format specified for *wait()*) is available as the return value of the *close()* method of the file object, except that when the exit status is zero (termination without errors), None is returned.

Availability: Unix, Windows.

2.6 版后已移除: This function is obsolete. Use the *subprocess* module. Check especially the *Replacing Older Functions with the subprocess Module* section.

在 2.0 版更改: This function worked unreliably under Windows in earlier versions of Python. This was due to the use of the _popen() function from the libraries provided with Windows. Newer versions of Python do not use the broken implementation from the Windows libraries.

os.**tmpfile**()
Return a new file object opened in update mode (w+b). The file has no directory entries associated with it and will be automatically deleted once there are no file descriptors for the file.

Availability: Unix, Windows.

There are a number of different popen*() functions that provide slightly different ways to create subprocesses.

2.6 版后已移除: All of the popen*() functions are obsolete. Use the *subprocess* module.

For each of the popen*() variants, if *bufsize* is specified, it specifies the buffer size for the I/O pipes. *mode*, if provided, should be the string 'b' or 't'; on Windows this is needed to determine whether the file objects should be opened in binary or text mode. The default value for *mode* is 't'.

Also, for each of these variants, on Unix, *cmd* may be a sequence, in which case arguments will be passed directly to the program without shell intervention (as with *os.spawnv()*). If *cmd* is a string it will be passed to the shell (as with *os.system()*).

These methods do not make it possible to retrieve the exit status from the child processes. The only way to control the input and output streams and also retrieve the return codes is to use the *subprocess* module; these are only available on Unix.

For a discussion of possible deadlock conditions related to the use of these functions, see *Flow Control Issues*.

os.**popen2**(*cmd*[, *mode*[, *bufsize*]])
Execute *cmd* as a sub-process and return the file objects (child_stdin, child_stdout).

2.6 版后已移除: This function is obsolete. Use the *subprocess* module. Check especially the *Replacing Older Functions with the subprocess Module* section.

Availability: Unix, Windows.

2.0 新版功能.

os.**popen3**(*cmd*[, *mode*[, *bufsize*]])
Execute *cmd* as a sub-process and return the file objects (child_stdin, child_stdout, child_stderr).

2.6 版后已移除: This function is obsolete. Use the *subprocess* module. Check especially the *Replacing Older Functions with the subprocess Module* section.

Availability: Unix, Windows.

2.0 新版功能.

os.**popen4**(*cmd*[, *mode*[, *bufsize*]])
Execute *cmd* as a sub-process and return the file objects (child_stdin, child_stdout_and_stderr).

2.6 版后已移除: This function is obsolete. Use the *subprocess* module. Check especially the *Replacing Older Functions with the subprocess Module* section.

Availability: Unix, Windows.

2.0 新版功能.

(Note that child_stdin, child_stdout, and child_stderr are named from the point of view of the child process, so *child_stdin* is the child's standard input.)

This functionality is also available in the *popen2* module using functions of the same names, but the return values of those functions have a different order.

15.1.3 文件描述符操作

这些函数对文件描述符所引用的 I/O 流进行操作。

文件描述符是一些小的整数，对应于当前进程所打开的文件。例如，标准输入的文件描述符通常是 0，标准输出是 1，标准错误是 2。之后被进程打开的文件的文件描述符会被依次指定为 3，4，5 等。“文件描述符”这个词有点误导性，在 Unix 平台中套接字和管道也被文件描述符所引用。

The *fileno()* method can be used to obtain the file descriptor associated with a file object when required. Note that using the file descriptor directly will bypass the file object methods, ignoring aspects such as internal buffering of data.

os.**close**(*fd*)
关闭文件描述符 *fd*。

Availability: Unix, Windows.

> **注解:** 该功能适用于低级 I/O 操作，必须用于*os.open()* 或*pipe()* 返回的文件描述符。关闭由内建函数*open()* ，*popen()* 或*fdopen()* 返回的“文件对象”，则使用其相应的*close()* 方法。

os.**closerange**(*fd_low*, *fd_high*)
Close all file descriptors from *fd_low* (inclusive) to *fd_high* (exclusive), ignoring errors. Equivalent to:

```
for fd in xrange(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

Availability: Unix, Windows.

2.6 新版功能.

os.**dup**(*fd*)
Return a duplicate of file descriptor *fd*.

Availability: Unix, Windows.

os.**dup2**(*fd*, *fd2*)
Duplicate file descriptor *fd* to *fd2*, closing the latter first if necessary.

Availability: Unix, Windows.

os.**fchmod**(*fd*, *mode*)
Change the mode of the file given by *fd* to the numeric *mode*. See the docs for *chmod()* for possible values of *mode*.

Availability: Unix.

2.6 新版功能.

os.**fchown**(*fd*, *uid*, *gid*)
Change the owner and group id of the file given by *fd* to the numeric *uid* and *gid*. To leave one of the ids unchanged, set it to -1.

Availability: Unix.

2.6 新版功能.

os.**fdatasync**(*fd*)
强制将文件描述符 *fd* 指定文件写入磁盘。不强制更新元数据。

Availability: Unix.

注解: 该功能在 MacOS 中不可用。

---

os.**fpathconf**(*fd*, *name*)
返回与打开的文件有关的系统配置信息。*name* 指定要查找的配置名称，它可以是字符串，是一个系统已定义的名称，这些名称定义在不同标准（POSIX.1，Unix 95，Unix 98 等）中。一些平台还定义了额外的其他名称。当前操作系统已定义的名称在 pathconf_names 字典中给出。对于未包含在该映射中的配置名称，也可以传递一个整数作为 *name*。

如果 *name* 是一个字符串且不是已定义的名称，将抛出*ValueError* 异常。如果当前系统不支持 *name* 指定的配置名称，即使该名称存在于 pathconf_names，也会抛出*OSError* 异常，错误码为*errno.EINVAL*。

Availability: Unix.

os.**fstat**(*fd*)
Return status for file descriptor *fd*, like *stat()*.

Availability: Unix, Windows.

os.**fstatvfs**(*fd*)
Return information about the filesystem containing the file associated with file descriptor *fd*, like *statvfs()*.

Availability: Unix.

os.**fsync**(*fd*)
强制将文件描述符 *fd* 指向的文件写入磁盘。在 Unix，这将调用原生 fsync() 函数；在 Windows，则是 MS _commit() 函数。

If you're starting with a Python file object *f*, first do f.flush(), and then do os.fsync(f.fileno()), to ensure that all internal buffers associated with *f* are written to disk.

Availability: Unix, and Windows starting in 2.2.3.

os.**ftruncate**(*fd*, *length*)
Truncate the file corresponding to file descriptor *fd*, so that it is at most *length* bytes in size.

Availability: Unix.

os.**isatty**(*fd*)
如果文件描述符 *fd* 打开且已连接至 tty 设备（或类 tty 设备），返回 True，否则返回 False。

os.**lseek**(*fd*, *pos*, *how*)
将文件描述符 *fd* 的当前位置设置为 *pos*，位置的计算方式 *how* 如下：设置为*SEEK_SET* 或 0 表示从文件开头计算，设置为*SEEK_CUR* 或 1 表示从文件当前位置计算，设置为*SEEK_END* 或 2 表示文件末尾计算。返回新指针位置，这个位置是从文件开头计算的，单位是字节。

Availability: Unix, Windows.

os.**SEEK_SET**
os.**SEEK_CUR**
os.**SEEK_END**
lseek() 函数的参数，它们的值分别为 0、1 和 2。

Availability: Windows, Unix.

2.5 新版功能.

os.**open**(*file*, *flags*[, *mode*])
Open the file *file* and set various flags according to *flags* and possibly its mode according to *mode*. The default *mode* is 0777 (octal), and the current umask value is first masked out. Return the file descriptor for the newly opened file.

For a description of the flag and mode values, see the C run-time documentation; flag constants (like *O_RDONLY* and *O_WRONLY*) are defined in this module too (see *open() flag constants*). In particular, on Windows adding *O_BINARY* is needed to open files in binary mode.

Availability: Unix, Windows.

注解： This function is intended for low-level I/O. For normal usage, use the built-in function *open()*, which returns a "file object" with *read()* and *write()* methods (and many more). To wrap a file descriptor in a "file object", use *fdopen()*.

---

os.**openpty**()
Open a new pseudo-terminal pair. Return a pair of file descriptors (master, slave) for the pty and the tty, respectively. For a (slightly) more portable approach, use the *pty* module.

Availability: some flavors of Unix.

os.**pipe**()
Create a pipe. Return a pair of file descriptors (r, w) usable for reading and writing, respectively.

Availability: Unix, Windows.

os.**read**(*fd*, *n*)
Read at most *n* bytes from file descriptor *fd*. Return a string containing the bytes read. If the end of the file referred to by *fd* has been reached, an empty string is returned.

Availability: Unix, Windows.

注解: 该功能适用于低级 I/O 操作，必须用于*os.open()* 或*pipe()* 返回的文件描述符。读取由内建函数*open()*、*popen()*、*fdopen()* 或*sys.stdin* 返回的“文件对象”，则使用其相应的*read()* 或*readline()* 方法。

---

os.**tcgetpgrp**(*fd*)
返回与 *fd* 给定的终端相关联的进程组（*fd* 是由*os.open()* 返回的已打开文件的描述符）。

Availability: Unix.

os.**tcsetpgrp**(*fd*, *pg*)
设置与 *fd* 指定的终端相关联的进程组为 *pg*（**fd* 是由*os.open()* 返回的已打开的文件描述符）。

Availability: Unix.

os.**ttyname**(*fd*)
返回一个字符串，该字符串表示与文件描述符 *fd* 关联的终端。如果 *fd* 没有与终端关联，则抛出异常。

Availability: Unix.

os.**write**(*fd*, *str*)
Write the string *str* to file descriptor *fd*. Return the number of bytes actually written.

Availability: Unix, Windows.

注解: 该功能适用于低级 I/O 操作，必须用于*os.open()* 或*pipe()* 返回的文件描述符。若要写入由内建函数*open()*、*popen()*、*fdopen()*、*sys.stdout* 或*sys.stderr* 返回的“文件对象”，则应使用其相应的*write()* 方法。

---

open() flag constants

The following constants are options for the *flags* parameter to the *open()* function. They can be combined using the bitwise OR operator |. Some of them are not available on all platforms. For descriptions of their availability and use, consult the *open(2)* manual page on Unix or the MSDN on Windows.

os.**O_RDONLY**
os.**O_WRONLY**
os.**O_RDWR**
os.**O_APPEND**
os.**O_CREAT**
os.**O_EXCL**
os.**O_TRUNC**
上述常量在 Unix 和 Windows 上均可用。

os.**O_DSYNC**
os.**O_RSYNC**
os.**O_SYNC**
os.**O_NDELAY**
os.**O_NONBLOCK**

os.**O_NOCTTY**
这个常数仅在 Unix 系统中可用。

os.**O_BINARY**
os.**O_NOINHERIT**
os.**O_SHORT_LIVED**
os.**O_TEMPORARY**
os.**O_RANDOM**
os.**O_SEQUENTIAL**
os.**O_TEXT**
这个常数仅在 Windows 系统中可用。

os.**O_ASYNC**
os.**O_DIRECT**
os.**O_DIRECTORY**
os.**O_NOFOLLOW**
os.**O_NOATIME**
os.**O_SHLOCK**
os.**O_EXLOCK**
上述常量是扩展常量，如果 C 库未定义它们，则不存在。

15.1.4 文件和目录

os.**access**(*path*, *mode*)
使用真实的 uid/gid 测试对 *path* 的访问。请注意，大多数测试操作将使用有效的 uid/gid，因此可以在 suid/sgid 环境中运用此例程，来测试调用用户是否具有对 *path* 的指定访问权限。*mode* 为*F_OK* 时用于测试 *path* 是否存在，也可以对*R_OK*、*W_OK* 和*X_OK* 中的一个或多个进行“或”运算来测试指定权限。允许访问则返回*True*，否则返回*False*。更多信息请参见 Unix 手册页 *access(2)*。

Availability: Unix, Windows.

注解: 使用*access()* 来检查用户是否具有某项权限（如打开文件的权限），然后再使用*open()* 打开文件，这样做存在一个安全漏洞，因为用户可能会在检查和打开文件之间的时间里做其他操作。推荐使用*EAFP* 技术。如:

```
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

最好写成:

```
try:
    fp = open("myfile")
except IOError as e:
    if e.errno == errno.EACCES:
        return "some default data"
    # Not a permission error.
    raise
else:
    with fp:
        return fp.read()
```

注解： 即使*access()* 指示 I/O 操作会成功，但实际操作仍可能失败，尤其是对网络文件系统的操作，其权限语义可能超出常规的 POSIX 权限位模型。

os.**F_OK**
Value to pass as the *mode* parameter of *access()* to test the existence of *path*.

os.**R_OK**
Value to include in the *mode* parameter of *access()* to test the readability of *path*.

os.**W_OK**
Value to include in the *mode* parameter of *access()* to test the writability of *path*.

os.**X_OK**
Value to include in the *mode* parameter of *access()* to determine if *path* can be executed.

os.**chdir**(*path*)
将当前工作目录更改为 *path*。

Availability: Unix, Windows.

os.**fchdir**(*fd*)
Change the current working directory to the directory represented by the file descriptor *fd*. The descriptor must refer to an opened directory, not an open file.

Availability: Unix.

2.3 新版功能.

os.**getcwd**()
返回表示当前工作目录的字符串。

Availability: Unix, Windows.

os.**getcwdu**()
Return a Unicode object representing the current working directory.

Availability: Unix, Windows.

2.3 新版功能.

os.**chflags**(*path*, *flags*)
将 *path* 标志设置为数字 标志。标志可以用以下值按位或组合起来（以下值在*stat* 模块中定义）：

- *stat.UF_NODUMP*
- *stat.UF_IMMUTABLE*
- *stat.UF_APPEND*
- *stat.UF_OPAQUE*
- *stat.UF_NOUNLINK*
- *stat.UF_COMPRESSED*
- *stat.UF_HIDDEN*
- *stat.SF_ARCHIVED*
- *stat.SF_IMMUTABLE*
- *stat.SF_APPEND*
- *stat.SF_NOUNLINK*

- *stat.SF_SNAPSHOT*

Availability: Unix.

2.6 新版功能.

os.**chroot**(*path*)
Change the root directory of the current process to *path*. Availability: Unix.

2.2 新版功能.

os.**chmod**(*path*, *mode*)
将 *path* 的 mode 更改为其他由数字表示的 *mode*。*mode* 可以用以下值之一，也可以将它们按位或组合起来（以下值在*stat* 模块中定义）：

- *stat.S_ISUID*
- *stat.S_ISGID*
- *stat.S_ENFMT*
- *stat.S_ISVTX*
- *stat.S_IREAD*
- *stat.S_IWRITE*
- *stat.S_IEXEC*
- *stat.S_IRWXU*
- *stat.S_IRUSR*
- *stat.S_IWUSR*
- *stat.S_IXUSR*
- *stat.S_IRWXG*
- *stat.S_IRGRP*
- *stat.S_IWGRP*
- *stat.S_IXGRP*
- *stat.S_IRWXO*
- *stat.S_IROTH*
- *stat.S_IWOTH*
- *stat.S_IXOTH*

Availability: Unix, Windows.

注解： Although Windows supports *chmod()*, you can only set the file's read-only flag with it (via the `stat.S_IWRITE` and `stat.S_IREAD` constants or a corresponding integer value). All other bits are ignored.

---

os.**chown**(*path*, *uid*, *gid*)
Change the owner and group id of *path* to the numeric *uid* and *gid*. To leave one of the ids unchanged, set it to -1.

Availability: Unix.

os.**lchflags**(*path*, *flags*)
Set the flags of *path* to the numeric *flags*, like *chflags()*, but do not follow symbolic links.

Availability: Unix.

2.6 新版功能.

os.**lchmod**(*path*, *mode*)
Change the mode of *path* to the numeric *mode*. If path is a symlink, this affects the symlink rather than the target. See the docs for *chmod()* for possible values of *mode*.

Availability: Unix.

2.6 新版功能.

os.**lchown**(*path*, *uid*, *gid*)
Change the owner and group id of *path* to the numeric *uid* and *gid*. This function will not follow symbolic links.

Availability: Unix.

2.3 新版功能.

os.**link**(*source*, *link_name*)
Create a hard link pointing to *source* named *link_name*.

Availability: Unix.

os.**listdir**(*path*)
Return a list containing the names of the entries in the directory given by *path*. The list is in arbitrary order. It does not include the special entries `'.'` and `'..'` even if they are present in the directory.

Availability: Unix, Windows.

在 2.3 版更改: On Windows NT/2k/XP and Unix, if *path* is a Unicode object, the result will be a list of Unicode objects. Undecodable filenames will still be returned as string objects.

os.**lstat**(*path*)
Perform the equivalent of an `lstat()` system call on the given path. Similar to *stat()*, but does not follow symbolic links. On platforms that do not support symbolic links, this is an alias for *stat()*.

os.**mkfifo**(*path*[, *mode*])
Create a FIFO (a named pipe) named *path* with numeric mode *mode*. The default *mode* is `0666` (octal). The current umask value is first masked out from the mode.

Availability: Unix.

FIFO 是可以像常规文件一样访问的管道。FIFO 如果没有被删除（如使用*os.unlink()*），会一直存在。通常，FIFO 用作“客户端”和“服务器”进程之间的汇合点：服务器打开 FIFO 进行读取，而客户端打开 FIFO 进行写入。请注意，*mkfifo()* 不会打开 FIFO —它只是创建汇合点。

os.**mknod**(*filename*[, *mode=0600*[, *device=0*]])
Create a filesystem node (file, device special file or named pipe) named *filename*. *mode* specifies both the permissions to use and the type of node to be created, being combined (bitwise OR) with one of `stat.S_IFREG`, `stat.S_IFCHR`, `stat.S_IFBLK`, and `stat.S_IFIFO` (those constants are available in *stat*). For `stat.S_IFCHR` and `stat.S_IFBLK`, *device* defines the newly created device special file (probably using *os.makedev()*), otherwise it is ignored.

2.3 新版功能.

os.**major**(*device*)
提取主设备号，提取自原始设备号（通常是 `stat` 中的 `st_dev` 或 `st_rdev` 字段）。

2.3 新版功能.

os.**minor**(*device*)
提取次设备号，提取自原始设备号（通常是 `stat` 中的 `st_dev` 或 `st_rdev` 字段）。

2.3 新版功能.

os.**makedev**(*major*, *minor*)
将主设备号和次设备号组合成原始设备号。

2.3 新版功能.

os.**mkdir**(*path*[, *mode*])
Create a directory named *path* with numeric mode *mode*. The default *mode* is `0777` (octal). If the directory already exists, *OSError* is raised.

某些系统会忽略 *mode*。如果没有忽略它，那么将首先从它中减去当前的 umask 值。如果除最后 9 位（即 *mode* 八进制的最后 3 位）之外，还设置了其他位，则其他位的含义取决于各个平台。在某些平台上，它们会被忽略，应显式调用*chmod()* 进行设置。

如果需要创建临时目录，请参阅*tempfile* 模块中的*tempfile.mkdtemp()* 函数。

Availability: Unix, Windows.

os.**makedirs**(*path*[, *mode*])
Recursive directory creation function. Like *mkdir()*, but makes all intermediate-level directories needed to contain the leaf directory. Raises an *error* exception if the leaf directory already exists or cannot be created. The default *mode* is `0777` (octal).

The *mode* parameter is passed to *mkdir()*; see *the mkdir() description* for how it is interpreted.

注解： *makedirs()* will become confused if the path elements to create include *os.pardir*.

1.5.2 新版功能.

在 2.3 版更改: This function now handles UNC paths correctly.

os.**pathconf**(*path*, *name*)
返回所给名称的文件有关的系统配置信息。*name* 指定要查找的配置名称，它可以是字符串，是一个系统已定义的名称，这些名称定义在不同标准（POSIX.1，Unix 95，Unix 98 等）中。一些平台还定义了额外的其他名称。当前操作系统已定义的名称在 `pathconf_names` 字典中给出。对于未包含在该映射中的配置名称，也可以传递一个整数作为 *name*。

如果 *name* 是一个字符串且不是已定义的名称，将抛出*ValueError* 异常。如果当前系统不支持 *name* 指定的配置名称，即使该名称存在于 `pathconf_names`，也会抛出*OSError* 异常，错误码为*errno.EINVAL*。

Availability: Unix.

os.**pathconf_names**
Dictionary mapping names accepted by *pathconf()* and *fpathconf()* to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system. Availability: Unix.

os.**readlink**(*path*)
返回一个字符串，为符号链接指向的实际路径。其结果可以是绝对或相对路径。如果是相对路径，则可用 `os.path.join(os.path.dirname(path), result)` 转换为绝对路径。

在 2.6 版更改: If the *path* is a Unicode object the result will also be a Unicode object.

Availability: Unix.

os.**remove**(*path*)
Remove (delete) the file *path*. If *path* is a directory, *OSError* is raised; see *rmdir()* below to remove a directory. This is identical to the *unlink()* function documented below. On Windows, attempting to remove a file that is in use causes an exception to be raised; on Unix, the directory entry is removed but the storage allocated to the file is not made available until the original file is no longer in use.

Availability: Unix, Windows.

os.**removedirs**(*path*)
递归删除目录。工作方式类似于*rmdir()*，不同之处在于，如果成功删除了末尾一级目录，*removedirs()* 会尝试依次删除 *path* 中提到的每个父目录，直到抛出错误为止（但该错误会被忽略，因为这通常表示父目录不是空目录）。例如，os.removedirs('foo/bar/baz') 将首先删除目录 'foo/bar/baz'，然后如果 'foo/bar' 和 'foo' 为空，则继续删除它们。如果无法成功删除末尾一级目录，则抛出*OSError* 异常。

1.5.2 新版功能.

os.**rename**(*src*, *dst*)
Rename the file or directory *src* to *dst*. If *dst* is a directory, *OSError* will be raised. On Unix, if *dst* exists and is a file, it will be replaced silently if the user has permission. The operation may fail on some Unix flavors if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement). On Windows, if *dst* already exists, *OSError* will be raised even if it is a file; there may be no way to implement an atomic rename when *dst* names an existing file.

Availability: Unix, Windows.

os.**renames**(*old*, *new*)
递归重命名目录或文件。工作方式类似*rename()*，除了会首先创建新路径所需的中间目录。重命名后，将调用*removedirs()* 删除旧路径中不需要的目录。

1.5.2 新版功能.

注解: 如果用户没有权限删除末级的目录或文件，则本函数可能会无法建立新的目录结构。

os.**rmdir**(*path*)
Remove (delete) the directory *path*. Only works when the directory is empty, otherwise, *OSError* is raised. In order to remove whole directory trees, *shutil.rmtree()* can be used.

Availability: Unix, Windows.

os.**stat**(*path*)
Perform the equivalent of a stat() system call on the given path. (This function follows symlinks; to stat a symlink use *lstat()*.)

The return value is an object whose attributes correspond to the members of the stat structure, namely:

- st_mode - protection bits,
- st_ino - inode number,
- st_dev - device,
- st_nlink - number of hard links,
- st_uid - user id of owner,
- st_gid - group id of owner,
- st_size - size of file, in bytes,
- st_atime - time of most recent access,
- st_mtime - time of most recent content modification,
- st_ctime - platform dependent; time of most recent metadata change on Unix, or the time of creation on Windows)

在 2.3 版更改: If *stat_float_times()* returns True, the time values are floats, measuring seconds. Fractions of a second may be reported if the system supports that. See *stat_float_times()* for further discussion.

在某些 Unix 系统上（如 Linux 上），以下属性可能也可用：

- st_blocks - number of 512-byte blocks allocated for file
- st_blksize - filesystem blocksize for efficient file system I/O
- st_rdev - type of device if an inode device
- st_flags - user defined flags for file

在其他 Unix 系统上（如 FreeBSD 上），以下属性可能可用（但仅当 root 使用它们时才被填充）：

- st_gen - file generation number
- st_birthtime - time of file creation

On RISCOS systems, the following attributes are also available:

- st_ftype (file type)
- st_attrs (attributes)
- st_obtype (object type).

注解： st_atime、st_mtime 和 st_ctime 属性的确切含义和分辨率取决于操作系统和文件系统。例如，在使用 FAT 或 FAT32 文件系统的 Windows 上，st_mtime 有 2 秒的分辨率，而 st_atime 仅有 1 天的分辨率。详细信息请参阅操作系统文档。

For backward compatibility, the return value of *stat()* is also accessible as a tuple of at least 10 integers giving the most important (and portable) members of the stat structure, in the order st_mode, st_ino, st_dev, st_nlink, st_uid, st_gid, st_size, st_atime, st_mtime, st_ctime. More items may be added at the end by some implementations.

标准模块*stat* 中定义了函数和常量，这些函数和常量可用于从 stat 结构体中提取信息。（在 Windows 上，某些项填充的是虚值。）

示例:

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
(33188, 422511, 769, 1, 1032, 100, 926, 1105022698,1105022732, 1105022732)
>>> statinfo.st_size
926
```

Availability: Unix, Windows.

在 2.2 版更改: Added access to values as attributes of the returned object.

在 2.5 版更改: Added st_gen and st_birthtime.

os.**stat_float_times**([*newvalue*])

Determine whether stat_result represents time stamps as float objects. If *newvalue* is True, future calls to *stat()* return floats, if it is False, future calls return ints. If *newvalue* is omitted, return the current setting.

For compatibility with older Python versions, accessing stat_result as a tuple always returns integers.

在 2.5 版更改: Python now returns float values by default. Applications which do not work correctly with floating point time stamps can use this function to restore the old behaviour.

The resolution of the timestamps (that is the smallest possible fraction) depends on the system. Some systems only support second resolution; on these systems, the fraction will always be zero.

It is recommended that this setting is only changed at program startup time in the *__main__* module; libraries should never change this setting. If an application uses a library that works incorrectly if floating point time stamps are processed, this application should turn the feature off until the library has been corrected.

os.**statvfs**(*path*)

Perform a `statvfs()` system call on the given path. The return value is an object whose attributes describe the filesystem on the given path, and correspond to the members of the `statvfs` structure, namely: `f_bsize`, `f_frsize`, `f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`, `f_flag`, `f_namemax`.

For backward compatibility, the return value is also accessible as a tuple whose values correspond to the attributes, in the order given above. The standard module *statvfs* defines constants that are useful for extracting information from a `statvfs` structure when accessing it as a sequence; this remains useful when writing code that needs to work with versions of Python that don't support accessing the fields as attributes.

Availability: Unix.

在 2.2 版更改: Added access to values as attributes of the returned object.

os.**symlink**(*source*, *link_name*)

Create a symbolic link pointing to *source* named *link_name*.

Availability: Unix.

os.**tempnam**([*dir*[, *prefix*]])

Return a unique path name that is reasonable for creating a temporary file. This will be an absolute path that names a potential directory entry in the directory *dir* or a common location for temporary files if *dir* is omitted or `None`. If given and not `None`, *prefix* is used to provide a short prefix to the filename. Applications are responsible for properly creating and managing files created using paths returned by *tempnam()*; no automatic cleanup is provided. On Unix, the environment variable `TMPDIR` overrides *dir*, while on Windows `TMP` is used. The specific behavior of this function depends on the C library implementation; some aspects are underspecified in system documentation.

> **警告:** Use of *tempnam()* is vulnerable to symlink attacks; consider using *tmpfile()* (section 创建文件对象) instead.

Availability: Unix, Windows.

os.**tmpnam**()

Return a unique path name that is reasonable for creating a temporary file. This will be an absolute path that names a potential directory entry in a common location for temporary files. Applications are responsible for properly creating and managing files created using paths returned by *tmpnam()*; no automatic cleanup is provided.

> **警告:** Use of *tmpnam()* is vulnerable to symlink attacks; consider using *tmpfile()* (section 创建文件对象) instead.

Availability: Unix, Windows. This function probably shouldn't be used on Windows, though: Microsoft's implementation of *tmpnam()* always creates a name in the root directory of the current drive, and that's generally a poor location for a temp file (depending on privileges, you may not even be able to open a file using this name).

os.**TMP_MAX**

The maximum number of unique names that *tmpnam()* will generate before reusing names.

os.**unlink**(*path*)
Remove (delete) the file *path*. This is the same function as *remove()*; the *unlink()* name is its traditional Unix name.

Availability: Unix, Windows.

os.**utime**(*path*, *times*)
Set the access and modified times of the file specified by *path*. If *times* is None, then the file's access and modified times are set to the current time. (The effect is similar to running the Unix program **touch** on the path.) Otherwise, *times* must be a 2-tuple of numbers, of the form (atime, mtime) which is used to set the access and modified times, respectively. Whether a directory can be given for *path* depends on whether the operating system implements directories as files (for example, Windows does not). Note that the exact times you set here may not be returned by a subsequent *stat()* call, depending on the resolution with which your operating system records access and modification times; see *stat()*.

在 2.0 版更改: Added support for None for *times*.

Availability: Unix, Windows.

os.**walk**(*top*, *topdown=True*, *onerror=None*, *followlinks=False*)
生成目录树中的文件名，方式是按上-> 下或下-> 上顺序浏览目录树。对于以 *top* 为根的目录树中的每个目录（包括 *top* 本身），它都会生成一个三元组 (dirpath, dirnames, filenames)。

dirpath 是一个字符串，表示目录的路径。*dirnames* 是一个列表，内含 *dirpath* 中子目录的名称（不包括 '.' 和 '..'）。*filenames* 也是列表，内含 *dirpath* 中非目录文件的名称。注意，列表中的名称不包含路径部分。要获取 *dirpath* 中文件或目录的完整路径（从 *top* 起始），请执行 os.path.join(dirpath, name)。

如果可选参数 *topdown* 为 True 或未指定，则在所有子目录的三元组之前生成父目录的三元组（目录是自上而下生成的）。如果 *topdown* 为 False，则在所有子目录的三元组生成之后再生成父目录的三元组（目录是自下而上生成的）。无论 *topdown* 为何值，在生成目录及其子目录的元组之前，都将检索全部子目录列表。

当 *topdown* 为 True 时，调用者可以就地修改 *dirnames* 列表（也许用到了 del 或切片），而*walk()* 将仅仅递归到仍保留在 *dirnames* 中的子目录内。这可用于减少搜索、加入特定的访问顺序，甚至可在继续*walk()* 之前告知*walk()* 由调用者新建或重命名的目录的信息。当 *topdown* 为 False 时，修改 *dirnames* 对 walk 的行为没有影响，因为在自下而上模式中，*dirnames* 中的目录是在 *dirpath* 本身之前生成的。

By default, errors from the *listdir()* call are ignored. If optional argument *onerror* is specified, it should be a function; it will be called with one argument, an *OSError* instance. It can report the error to continue with the walk, or raise the exception to abort the walk. Note that the filename is available as the filename attribute of the exception object.

walk() 默认不会递归进指向目录的符号链接。可以在支持符号链接的系统上将 *followlinks* 设置为 True，以访问符号链接指向的目录。

2.6 新版功能: The *followlinks* parameter.

注解: 注意，如果链接指向自身的父目录，则将 *followlinks* 设置为 True 可能导致无限递归。*walk()* 不会记录它已经访问过的目录。

注解: 如果传入的是相对路径，请不要在恢复*walk()* 之间更改当前工作目录。*walk()* 不会更改当前目录，并假定其调用者也不会更改当前目录。

下面的示例遍历起始目录内所有子目录，打印每个目录内的文件占用的字节数，CVS 子目录不会被遍历:

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print root, "consumes",
    print sum(getsize(join(root, name)) for name in files),
    print "bytes in", len(files), "non-directory files"
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

在下一个示例中，必须使树自下而上遍历，因为*rmdir()* 只允许在目录为空时删除目录:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

2.3 新版功能.

15.1.5 进程管理

下列函数可用于创建和管理进程。

所有*exec** 函数都接受一个参数列表，用来给新程序加载到它的进程中。在所有情况下，传递给新程序的第一个参数是程序本身的名称，而不是用户在命令行上输入的参数。对于 C 程序员来说，这就是传递给 `main()` 函数的 `argv[0]`。例如，`os.execv('/bin/echo', ['foo', 'bar'])` 只会在标准输出上打印 `bar`，而 `foo` 会被忽略。

`os.`**`abort`**`()`
发送 `SIGABRT` 信号到当前进程。在 Unix 上，默认行为是生成一个核心转储。在 Windows 上，该进程立即返回退出代码 `3`。请注意，使用*signal.signal()* 可以为 `SIGABRT` 注册 Python 信号处理程序，而调用本函数将不会调用按前述方法注册的程序。

Availability: Unix, Windows.

`os.`**`execl`**`(`*path*, *arg0*, *arg1*, ...`)`
`os.`**`execle`**`(`*path*, *arg0*, *arg1*, ..., *env*`)`
`os.`**`execlp`**`(`*file*, *arg0*, *arg1*, ...`)`
`os.`**`execlpe`**`(`*file*, *arg0*, *arg1*, ..., *env*`)`
`os.`**`execv`**`(`*path*, *args*`)`
`os.`**`execve`**`(`*path*, *args*, *env*`)`
`os.`**`execvp`**`(`*file*, *args*`)`
`os.`**`execvpe`**`(`*file*, *args*, *env*`)`
这些函数都将执行一个新程序，以替换当前进程。它们没有返回值。在 Unix 上，新程序会加载到当前进程中，且进程号与调用者相同。过程中的错误会被报告为*OSError* 异常。

当前进程会被立即替换。打开的文件对象和描述符都不会刷新，因此如果这些文件上可能缓冲了数据，则应在调用*exec** 函数之前使用 `sys.stdout.flush()` 或*os.fsync()* 刷新它们。

*exec** 函数的“l”和“v”变体不同在于命令行参数的传递方式。如果在编码时固定了参数数量，则“l”变体可能是最方便的，各参数作为 `execl*()` 函数的附加参数传入即可。当参数数量可变时，”v”

变体更方便，参数以列表或元组的形式作为 *args* 参数传递。在这两种情况下，子进程的第一个参数都应该是即将运行的命令名称，但这不是强制性的。

结尾包含“p” 的变体（*execlp()*、*execlpe()*、*execvp()* 和*execvpe()* ）将使用 PATH 环境变量来查找程序 *file*。当环境被替换时（使用下一段讨论的*exec*e* 变体之一），PATH 变量将来自于新环境。其他变体*execl()*、*execle()*、*execv()* 和*execve()* 不使用 PATH 变量来查找程序，因此 *path* 必须包含正确的绝对或相对路径。

对于*execle()*、*execlpe()*、*execve()* 和*execvpe()* （都以“e”结尾），*env* 参数是一个映射，用于定义新进程的环境变量（代替当前进程的环境变量）。而函数*execl()*、*execlp()*、*execv()* 和*execvp()* 会将当前进程的环境变量过继给新进程。

Availability: Unix, Windows.

os.**_exit**(*n*)
以状态码 *n* 退出进程，不会调用清理处理程序，不会刷新 stdio，等等。

Availability: Unix, Windows.

注解： 退出的标准方法是使用 sys.exit(n)。而*_exit()* 通常只应在*fork()* 出的子进程中使用。

以下是已定义的退出代码，可以用于*_exit()*，尽管它们不是必需的。这些退出代码通常用于 Python 编写的系统程序，例如邮件服务器的外部命令传递程序。

注解： 其中部分退出代码在部分 Unix 平台上可能不可用，因为平台间存在差异。如果底层平台定义了这些常量，那上层也会定义。

---

os.**EX_OK**
退出代码，表示未发生任何错误。

Availability: Unix.

2.3 新版功能.

os.**EX_USAGE**
退出代码，表示命令使用不正确，如给出的参数数量有误。

Availability: Unix.

2.3 新版功能.

os.**EX_DATAERR**
退出代码，表示输入数据不正确。

Availability: Unix.

2.3 新版功能.

os.**EX_NOINPUT**
退出代码，表示某个输入文件不存在或不可读。

Availability: Unix.

2.3 新版功能.

os.**EX_NOUSER**
退出代码，表示指定的用户不存在。

Availability: Unix.

2.3 新版功能.

os.**EX_NOHOST**
退出代码，表示指定的主机不存在。

Availability: Unix.

2.3 新版功能.

os.**EX_UNAVAILABLE**
退出代码，表示所需的服务不可用。

Availability: Unix.

2.3 新版功能.

os.**EX_SOFTWARE**
退出代码，表示检测到内部软件错误。

Availability: Unix.

2.3 新版功能.

os.**EX_OSERR**
退出代码，表示检测到操作系统错误，例如无法 fork 或创建管道。

Availability: Unix.

2.3 新版功能.

os.**EX_OSFILE**
退出代码，意味着某些系统文件不存在，无法打开或发生其他错误。

Availability: Unix.

2.3 新版功能.

os.**EX_CANTCREAT**
退出代码，表示无法创建用户指定的输出文件。

Availability: Unix.

2.3 新版功能.

os.**EX_IOERR**
退出代码，表示对某些文件进行读写时发生错误。

Availability: Unix.

2.3 新版功能.

os.**EX_TEMPFAIL**
退出代码，表示发生了暂时性故障。它可能并非意味着真正的错误，例如在可重试的情况下无法建立网络连接。

Availability: Unix.

2.3 新版功能.

os.**EX_PROTOCOL**
退出代码，表示协议交换是非法的、无效的或无法解读的。

Availability: Unix.

2.3 新版功能.

os.**EX_NOPERM**
退出代码，表示没有足够的权限执行该操作（但不适用于文件系统问题）。

Availability: Unix.

2.3 新版功能.

os.**EX_CONFIG**
退出代码，表示发生某种配置错误。

Availability: Unix.

2.3 新版功能.

os.**EX_NOTFOUND**
退出代码，表示的内容类似于“找不到条目”。

Availability: Unix.

2.3 新版功能.

os.**fork**()
Fork 出一个子进程。在子进程中返回 0，在父进程中返回子进程的进程号。如果发生错误，则抛出*OSError* 异常。

Note that some platforms including FreeBSD <= 6.3, Cygwin and OS/2 EMX have known issues when using fork() from a thread.

> **警告：** 有关 SSL 模块与 fork() 结合的应用，请参阅*ssl*。

Availability: Unix.

os.**forkpty**()
Fork 出一个子进程，使用新的伪终端作为子进程的控制终端。返回一对 (pid, fd)，其中 *pid* 在子进程中为 0，这是父进程中新子进程的进程号，而 *fd* 是伪终端主设备的文件描述符。对于更便于移植的方法，请使用*pty* 模块。如果发生错误，则抛出*OSError* 异常。

Availability: some flavors of Unix.

os.**kill**(*pid*, *sig*)
将信号 *sig* 发送至进程 *pid*。特定平台上可用的信号常量定义在*signal* 模块中。

Windows：*signal.CTRL_C_EVENT* 和*signal.CTRL_BREAK_EVENT* 信号是特殊信号，只能发送给共享同一个控制台窗口的控制台进程，如某些子进程。*sig* 取任何其他值将导致该进程被 TerminateProcess API 无条件终止，且退出代码为 *sig*。Windows 版本的*kill()* 还需要传入待结束进程的句柄。

2.7 新版功能: Windows support

os.**killpg**(*pgid*, *sig*)
将信号 *sig* 发送给进程组 *pgid*。

Availability: Unix.

2.3 新版功能.

os.**nice**(*increment*)
将进程的优先级（nice 值）增加 *increment*，返回新的 nice 值。

Availability: Unix.

os.**plock**(*op*)
将程序段锁定到内存中。*op* 的值（定义在 <sys/lock.h> 中）决定了哪些段被锁定。

Availability: Unix.

os.**popen**(...)
os.**popen2**(...)
os.**popen3**(...)

os.**popen4**(...)
Run child processes, returning opened pipes for communications. These functions are described in section 创建文件对象.

os.**spawnl**(*mode*, *path*, ...)
os.**spawnle**(*mode*, *path*, ..., *env*)
os.**spawnlp**(*mode*, *file*, ...)
os.**spawnlpe**(*mode*, *file*, ..., *env*)
os.**spawnv**(*mode*, *path*, *args*)
os.**spawnve**(*mode*, *path*, *args*, *env*)
os.**spawnvp**(*mode*, *file*, *args*)
os.**spawnvpe**(*mode*, *file*, *args*, *env*)
在新进程中执行程序 *path*。

(注意，*subprocess* 模块提供了更强大的工具来生成新进程并跟踪执行结果，使用该模块比使用这些函数更好。尤其应当检查*Replacing Older Functions with the subprocess Module* 部分。)

mode 为*P_NOWAIT* 时，本函数返回新进程的进程号。*mode* 为*P_WAIT* 时，如果进程正常退出，返回退出代码，如果被杀死，返回 -signal，其中 *signal* 是杀死进程的信号。在 Windows 上，进程号实际上是进程句柄，因此可以与*waitpid()* 函数一起使用。

*spawn** 函数的 “l” 和 “v” 变体不同在于命令行参数的传递方式。如果在编码时固定了参数数量，则 “l” 变体可能是最方便的，各参数作为 spawnl*() 函数的附加参数传入即可。当参数数量可变时，” v” 变体更方便，参数以列表或元组的形式作为 *args* 参数传递。在这两种情况下，子进程的第一个参数都必须是即将运行的命令名称。

结尾包含第二个 “p” 的变体 (*spawnlp()*、*spawnlpe()*、*spawnvp()* 和*spawnvpe()*) 将使用 PATH 环境变量来查找程序 *file*。当环境被替换时（使用下一段讨论的*spawn*e* 变体之一)，PATH 变量将来自于新环境。其他变体*spawnl()*、*spawnle()*、*spawnv()* 和*spawnve()* 不使用 PATH 变量来查找程序，因此 *path* 必须包含正确的绝对或相对路径。

对于*spawnle()*、*spawnlpe()*、*spawnve()* 和*spawnvpe()* （都以 “e” 结尾)，*env* 参数是一个映射，用于定义新进程的环境变量（代替当前进程的环境变量)。而函数*spawnl()*、*spawnlp()*、*spawnv()* 和*spawnvp()* 会将当前进程的环境变量过继给新进程。注意，*env* 字典中的键和值必须是字符串。无效的键或值将导致函数出错，返回值为 127。

例如，以下对*spawnlp()* 和*spawnvpe()* 的调用是等效的:

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

Availability: Unix, Windows. *spawnlp()*, *spawnlpe()*, *spawnvp()* and *spawnvpe()* are not available on Windows. *spawnle()* and *spawnve()* are not thread-safe on Windows; we advise you to use the *subprocess* module instead.

1.6 新版功能.

os.**P_NOWAIT**
os.**P_NOWAITO**
*spawn** 系列函数的 *mode* 参数的可取值。如果给出这些值中的任何一个，则 spawn*() 函数将在创建新进程后立即返回，且返回值为进程号。

Availability: Unix, Windows.

1.6 新版功能.

os.**P_WAIT**
*spawn** 系列函数的 *mode* 参数的可取值。如果将 *mode* 指定为该值，则 spawn*() 函数将在新进程运行完毕后返回，运行成功则返回进程的退出代码，被信号终止则返回 -signal。

Availability: Unix, Windows.

1.6 新版功能.

os.**P_DETACH**
os.**P_OVERLAY**
Possible values for the *mode* parameter to the *spawn** family of functions. These are less portable than those listed above. *P_DETACH* is similar to *P_NOWAIT*, but the new process is detached from the console of the calling process. If *P_OVERLAY* is used, the current process will be replaced; the spawn*() function will not return.

Availability: Windows.

1.6 新版功能.

os.**startfile**(*path*[, *operation*])
使用已关联的应用程序打开文件。

当 *operation* 未指定或指定为 'open' 时，这类似于在 Windows 资源管理器中双击文件，或在交互式命令行中将文件名作为 **start** 命令的参数：通过扩展名相关联的应用程序（如果有）打开文件。

当指定另一个 *operation* 时，它必须是一个“命令动词”（“command verb”），该词指定对文件执行的操作。Microsoft 文档中的常用动词有 'print' 和 'edit'（用于文件），以及 'explore' 和 'find'（用于目录）。

关联的应用程序启动后*startfile()* 就会立即返回。本函数没有等待应用程序关闭的选项，也没有办法检索应用程序的退出状态。*path* 参数是基于当前目录的相对路径。如果要使用绝对路径，请确保第一个字符不是斜杠 ('/')，是斜杠的话底层的 Win32 ShellExecute() 函数将失效。使用*os.path.normpath()* 函数确保路径已针对 Win32 正确编码。

Availability: Windows.

2.0 新版功能.

2.5 新版功能: The *operation* parameter.

os.**system**(*command*)
Execute the command (a string) in a subshell. This is implemented by calling the Standard C function system(), and has the same limitations. Changes to *sys.stdin*, etc. are not reflected in the environment of the executed command.

在 Unix 上，返回值是进程的退出状态，编码格式与为*wait()* 指定的格式相同。注意，POSIX 没有指定 C 函数 system() 返回值的含义，因此 Python 函数的返回值与系统有关。

On Windows, the return value is that returned by the system shell after running *command*, given by the Windows environment variable COMSPEC: on **command.com** systems (Windows 95, 98 and ME) this is always 0; on **cmd.exe** systems (Windows NT, 2000 and XP) this is the exit status of the command run; on systems using a non-native shell, consult your shell documentation.

subprocess 模块提供了更强大的工具来生成新进程并跟踪执行结果，使用该模块比使用本函数更好。参阅*subprocess* 文档中的*Replacing Older Functions with the subprocess Module* 部分以获取有用的帮助。

Availability: Unix, Windows.

os.**times**()
Return a 5-tuple of floating point numbers indicating accumulated (processor or other) times, in seconds. The items are: user time, system time, children's user time, children's system time, and elapsed real time since a fixed point in the past, in that order. See the Unix manual page *times(2)* or the corresponding Windows Platform API documentation. On Windows, only the first two items are filled, the others are zero.

Availability: Unix, Windows

os.**wait**()
等待子进程执行完毕，返回一个元组，包含其 pid 和退出状态指示：一个 16 位数字，其低字节是终止该进程的信号编号，高字节是退出状态码（信号编号为零的情况下），如果生成了核心文件，则低字节的高位会置位。

Availability: Unix.

os.**waitpid**(*pid*, *options*)
本函数的细节在 Unix 和 Windows 上有不同之处。

在 Unix 上：等待进程号为 *pid* 的子进程执行完毕，返回一个元组，内含其进程 ID 和退出状态指示（编码与*wait()* 相同）。调用的语义受整数 *options* 的影响，常规操作下该值应为 0。

如果 *pid* 大于 0，则*waitpid()* 会获取该指定进程的状态信息。如果 *pid* 为 0，则获取当前进程所在进程组中的所有子进程的状态。如果 *pid* 为 -1，则获取当前进程的子进程状态。如果 *pid* 小于 -1，则获取进程组 -pid （*pid* 的绝对值）中所有进程的状态。

当系统调用返回 -1 时，将抛出带有错误码的*OSError* 异常。

在 Windows 上：等待句柄为 *pid* 的进程执行完毕，返回一个元组，内含 *pid* 以及左移 8 位后的退出状态（移位简化了跨平台使用本函数）。小于或等于 0 的 *pid* 在 Windows 上没有特殊含义，且会抛出异常。整数值 *options* 无效。*pid* 可以指向任何 ID 已知的进程，不一定是子进程。调用*spawn** 函数时传入*P_NOWAIT* 将返回合适的进程句柄。

os.**wait3**(*options*)
与*waitpid()* 相似，差别在于没有进程 ID 参数，且返回一个 3 元组，其中包括子进程 ID，退出状态指示和资源使用信息。关于资源使用信息的详情，请参考*resource*.*getrusage()*。option 参数与传入*waitpid()* 和*wait4()* 的相同。

Availability: Unix.

2.5 新版功能.

os.**wait4**(*pid*, *options*)
与*waitpid()* 相似，差别在本方法返回一个 3 元组，其中包括子进程 ID，退出状态指示和资源使用信息。关于资源使用信息的详情，请参考*resource*.*getrusage()*。*wait4()* 的参数与*waitpid()* 的参数相同。

Availability: Unix.

2.5 新版功能.

os.**WNOHANG**
用于*waitpid()* 的选项，如果没有立即可用的子进程状态，则立即返回。在这种情况下，函数返回 `(0, 0)`。

Availability: Unix.

os.**WCONTINUED**
被作业控制停止的子进程，如果自上次报告状态以来，已继续进行，则此选项将报告这些子进程。

Availability: Some Unix systems.

2.3 新版功能.

os.**WUNTRACED**
已停止的子进程，如果自停止以来尚未报告其当前状态，则此选项将报告这些子进程。

Availability: Unix.

2.3 新版功能.

下列函数采用进程状态码作为参数，状态码由*system()*、*wait()* 或*waitpid()* 返回。它们可用于确定进程上发生的操作。

os.**WCOREDUMP**(*status*)
如果为该进程生成了核心转储，返回 True，否则返回 False。

Availability: Unix.

2.3 新版功能.

os.**WIFCONTINUED**(*status*)
Return True if the process has been continued from a job control stop, otherwise return False.

Availability: Unix.

2.3 新版功能.

os.**WIFSTOPPED**(*status*)
Return True if the process has been stopped, otherwise return False.

Availability: Unix.

os.**WIFSIGNALED**(*status*)
Return True if the process exited due to a signal, otherwise return False.

Availability: Unix.

os.**WIFEXITED**(*status*)
Return True if the process exited using the *exit(2)* system call, otherwise return False.

Availability: Unix.

os.**WEXITSTATUS**(*status*)
If WIFEXITED(status) is true, return the integer parameter to the *exit(2)* system call. Otherwise, the return value is meaningless.

Availability: Unix.

os.**WSTOPSIG**(*status*)
返回导致进程停止的信号。

Availability: Unix.

os.**WTERMSIG**(*status*)
Return the signal which caused the process to exit.

Availability: Unix.

15.1.6 其他系统信息

os.**confstr**(*name*)
返回字符串格式的系统配置信息。*name* 指定要查找的配置名称，它可以是字符串，是一个系统已定义的名称，这些名称定义在不同标准（POSIX，Unix 95，Unix 98 等）中。一些平台还定义了额外的其他名称。当前操作系统已定义的名称在 confstr_names 字典的键中给出。对于未包含在该映射中的配置名称，也可以传递一个整数作为 *name*。

如果 *name* 指定的配置值未定义，返回 None。

如果 *name* 是一个字符串且不是已定义的名称，将抛出*ValueError* 异常。如果当前系统不支持 *name* 指定的配置名称，即使该名称存在于 confstr_names，也会抛出*OSError* 异常，错误码为*errno.EINVAL*。

Availability: Unix

os.**confstr_names**
字典，表示映射关系，为*confstr()* 可接受名称与操作系统为这些名称定义的整数值之间的映射。这可用于判断系统已定义了哪些名称。

Availability: Unix.

os.**getloadavg**()
返回系统运行队列中最近 1、5 和 15 分钟内的平均进程数。无法获得平均负载则抛出*OSError* 异常。

Availability: Unix.

2.3 新版功能.

os.**sysconf**(*name*)
返回整数格式的系统配置信息。如果 *name* 指定的配置值未定义，返回 -1。对*confstr()* 的 *name* 参数的注释在此处也适用。当前已知的配置名称在 sysconf_names 字典中提供。

Availability: Unix.

os.**sysconf_names**
字典，表示映射关系，为*sysconf()* 可接受名称与操作系统为这些名称定义的整数值之间的映射。这可用于判断系统已定义了哪些名称。

Availability: Unix.

以下数据值用于支持对路径本身的操作。所有平台都有定义。

对路径的高级操作在*os.path* 模块中定义。

os.**curdir**
操作系统用来表示当前目录的常量字符串。在 Windows 和 POSIX 上是 '.'。在*os.path* 中也可用。

os.**pardir**
操作系统用来表示父目录的常量字符串。在 Windows 和 POSIX 上是 '..'。在*os.path* 中也可用。

os.**sep**
操作系统用来分隔路径不同部分的字符。在 POSIX 上是 '/'，在 Windows 上是是 '\\'。注意，仅了解它不足以能解析或连接路径，请使用*os.path.split()* 和*os.path.join()*，但它有时是有用的。在*os.path* 中也可用。

os.**altsep**
操作系统用来分隔路径不同部分的替代字符。如果仅存在一个分隔符，则为 None。在 sep 是反斜杠的 Windows 系统上，该值被设为 '/'。在*os.path* 中也可用。

os.**extsep**
分隔基本文件名与扩展名的字符，如 os.py 中的 '.'。在*os.path* 中也可用。

2.2 新版功能.

os.**pathsep**
操作系统通常用于分隔搜索路径（如 PATH）中不同部分的字符，如 POSIX 上是 ':'，Windows 上是 ';'。在*os.path* 中也可用。

os.**defpath**
在环境变量没有 'PATH' 键的情况下，*exec*p** and *spawn*p** 使用的默认搜索路径。在*os.path* 中也可用。

os.**linesep**
当前平台用于分隔（或终止）行的字符串。它可以是单个字符，如 POSIX 上是 '\n'，也可以是多个字符，如 Windows 上是 '\r\n'。在写入以文本模式（默认模式）打开的文件时，请不要使用 *os.linesep* 作为行终止符，请在所有平台上都使用一个 '\n' 代替。

os.**devnull**
空设备的文件路径。如 POSIX 上为 '/dev/null'，Windows 上为 'nul'。在*os.path* 中也可用。

2.4 新版功能.

15.1.7 Miscellaneous Functions

os.**urandom**(*n*)

Return a string of *n* random bytes suitable for cryptographic use.

This function returns random bytes from an OS-specific randomness source. The returned data should be unpredictable enough for cryptographic applications, though its exact quality depends on the OS implementation. On a UNIX-like system this will query `/dev/urandom`, and on Windows it will use `CryptGenRandom()`. If a randomness source is not found, *NotImplementedError* will be raised.

For an easy-to-use interface to the random number generator provided by your platform, please see *random.SystemRandom*.

2.4 新版功能.

15.2 io —处理流的核心工具

2.6 新版功能.

The *io* module provides the Python interfaces to stream handling. Under Python 2.x, this is proposed as an alternative to the built-in *file* object, but in Python 3.x it is the default interface to access files and streams.

注解: Since this module has been designed primarily for Python 3.x, you have to be aware that all uses of "bytes" in this document refer to the *str* type (of which `bytes` is an alias), and all uses of "text" refer to the *unicode* type. Furthermore, those two types are not interchangeable in the *io* APIs.

At the top of the I/O hierarchy is the abstract base class *IOBase*. It defines the basic interface to a stream. Note, however, that there is no separation between reading and writing to streams; implementations are allowed to raise an *IOError* if they do not support a given operation.

Extending *IOBase* is *RawIOBase* which deals simply with the reading and writing of raw bytes to a stream. *FileIO* subclasses *RawIOBase* to provide an interface to files in the machine's file system.

BufferedIOBase deals with buffering on a raw byte stream (*RawIOBase*). Its subclasses, *BufferedWriter*, *BufferedReader*, and *BufferedRWPair* buffer streams that are readable, writable, and both readable and writable. *BufferedRandom* provides a buffered interface to random access streams. *BytesIO* is a simple stream of in-memory bytes.

Another *IOBase* subclass, *TextIOBase*, deals with streams whose bytes represent text, and handles encoding and decoding from and to *unicode* strings. *TextIOWrapper*, which extends it, is a buffered text interface to a buffered raw stream (*BufferedIOBase*). Finally, *StringIO* is an in-memory stream for unicode text.

Argument names are not part of the specification, and only the arguments of *open()* are intended to be used as keyword arguments.

15.2.1 Module Interface

io.**DEFAULT_BUFFER_SIZE**
An int containing the default buffer size used by the module's buffered I/O classes. *open()* uses the file's blksize (as obtained by *os.stat()*) if possible.

io.**open**(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*)
Open *file* and return a corresponding stream. If the file cannot be opened, an *IOError* is raised.

file is either a string giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless *closefd* is set to `False`.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode. Other common values are `'w'` for writing (truncating the file if it already exists), and `'a'` for appending (which on *some* Unix systems, means that *all* writes append to the end of the file regardless of the current seek position). In text mode, if *encoding* is not specified the encoding used is platform dependent. (For reading and writing raw bytes use binary mode and leave *encoding* unspecified.) The available modes are:

Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'a'`	open for writing, appending to the end of the file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open a disk file for updating (reading and writing)
`'U'`	universal newlines mode (for backwards compatibility; should not be used in new code)

The default mode is `'rt'` (open for reading text). For binary random access, the mode `'w+b'` opens and truncates the file to 0 bytes, while `'r+b'` opens the file without truncation.

Python distinguishes between files opened in binary and text modes, even when the underlying operating system doesn't. Files opened in binary mode (including `'b'` in the *mode* argument) return contents as `bytes` objects without any decoding. In text mode (the default, or when `'t'` is included in the *mode* argument), the contents of the file are returned as *unicode* strings, the bytes having been first decoded using a platform-dependent encoding or using the specified *encoding* if given.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no *buffering* argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on *DEFAULT_BUFFER_SIZE*. On many systems, the buffer will typically be 4096 or 8192 bytes long.
- "Interactive" text files (files for which `isatty()` returns True) use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent (whatever *locale.getpreferredencoding()* returns), but any encoding supported by Python can be used. See the *codecs* module for the list of supported encodings.

errors is an optional string that specifies how encoding and decoding errors are to be handled—this cannot be used in binary mode. Pass `'strict'` to raise a *ValueError* exception if there is an encoding error (the default of `None` has the same effect), or pass `'ignore'` to ignore errors. (Note that ignoring encoding errors can lead to data loss.) `'replace'` causes a replacement marker (such as `'?'`) to be inserted where there is malformed data. When writing, `'xmlcharrefreplace'` (replace with the appropriate XML character reference)

or 'backslashreplace' (replace with backslashed escape sequences) can be used. Any other error handling name that has been registered with *codecs.register_error()* is also valid.

newline controls how *universal newlines* works (it only applies to text mode). It can be None, '', '\n', '\r', and '\r\n'. It works as follows:

- On input, if *newline* is None, universal newlines mode is enabled. Lines in the input can end in '\n', '\r', or '\r\n', and these are translated into '\n' before being returned to the caller. If it is '', universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- On output, if *newline* is None, any '\n' characters written are translated to the system default line separator, *os.linesep*. If *newline* is '', no translation takes place. If *newline* is any of the other legal values, any '\n' characters written are translated to the given string.

If *closefd* is False and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given *closefd* has no effect and must be True (the default).

The type of file object returned by the *open()* function depends on the mode. When *open()* is used to open a file in a text mode ('w', 'r', 'wt', 'rt', etc.), it returns a subclass of *TextIOBase* (specifically *TextIOWrapper*). When used to open a file in a binary mode with buffering, the returned class is a subclass of *BufferedIOBase*. The exact class varies: in read binary mode, it returns a *BufferedReader*; in write binary and append binary modes, it returns a *BufferedWriter*, and in read/write mode, it returns a *BufferedRandom*. When buffering is disabled, the raw stream, a subclass of *RawIOBase*, *FileIO*, is returned.

It is also possible to use an *unicode* or bytes string as a file for both reading and writing. For *unicode* strings *StringIO* can be used like a file opened in text mode, and for bytes a *BytesIO* can be used like a file opened in a binary mode.

exception io.**BlockingIOError**
Error raised when blocking would occur on a non-blocking stream. It inherits *IOError*.

In addition to those of *IOError*, *BlockingIOError* has one attribute:

characters_written
An integer containing the number of characters written to the stream before it blocked.

exception io.**UnsupportedOperation**
An exception inheriting *IOError* and *ValueError* that is raised when an unsupported operation is called on a stream.

15.2.2 I/O 基类

class io.**IOBase**
所有 I/O 类的抽象基类，作用于字节流。没有公共构造函数。

此类为许多方法提供了空的抽象实现，派生类可以有选择地重写。默认实现无法读取、写入或查找的文件。

Even though *IOBase* does not declare read(), readinto(), or write() because their signatures will vary, implementations and clients should consider those methods part of the interface. Also, implementations may raise an *IOError* when operations they do not support are called.

The basic type used for binary data read from or written to a file is bytes (also known as *str*). Method arguments may also be *bytearray* or *memoryview* of arrays of bytes. In some cases, such as *readinto()*, a writable object such as *bytearray* is required. Text I/O classes work with *unicode* data.

在 2.7 版更改: Implementations should support *memoryview* arguments.

Note that calling any method (even inquiries) on a closed stream is undefined. Implementations may raise *IOError* in this case.

IOBase (and its subclasses) support the iterator protocol, meaning that an *IOBase* object can be iterated over yielding the lines in a stream. Lines are defined slightly differently depending on whether the stream is a binary stream (yielding bytes), or a text stream (yielding *unicode* strings). See *readline()* below.

IOBase is also a context manager and therefore supports the with statement. In this example, *file* is closed after the with statement's suite is finished—even if an exception occurs:

```
with io.open('spam.txt', 'w') as file:
    file.write(u'Spam and eggs!')
```

IOBase 提供以下数据属性和方法:

close()
刷新并关闭此流。如果文件已经关闭，则此方法无效。文件关闭后，对文件的任何操作（例如读取或写入）都会引发*ValueError* 。

为方便起见，允许多次调用此方法。但是，只有第一个调用才会生效。

closed
True if the stream is closed.

fileno()
Return the underlying file descriptor (an integer) of the stream if it exists. An *IOError* is raised if the IO object does not use a file descriptor.

flush()
刷新流的写入缓冲区（如果适用）。这对只读和非阻塞流不起作用。

isatty()
如果流是交互式的（即连接到终端/tty 设备），则返回 True 。

readable()
Return True if the stream can be read from. If False, read() will raise *IOError*.

readline(*limit=-1*)
Read and return one line from the stream. If *limit* is specified, at most *limit* bytes will be read.

The line terminator is always b'\n' for binary files; for text files, the *newline* argument to *open()* can be used to select the line terminator(s) recognized.

readlines(*hint=-1*)
从流中读取并返回包含多行的列表。可以指定 *hint* 来控制要读取的行数：如果（以字节/字符数表示的）所有行的总大小超出了 *hint* 则将不会读取更多的行。

请注意使用 for line in file: ... 就足够对文件对象进行迭代了，可以不必调用 file.readlines()。

seek(*offset*, *whence=SEEK_SET*)
将流位置修改到给定的字节 *offset*。*offset* 将相对于由 *whence* 指定的位置进行解析。*whence* 的默认值为 SEEK_SET。*whence* 的可用值有:

- SEEK_SET 或 0 –流的开头（默认值）；*offset* 应为零或正值
- SEEK_CUR or 1 –当前流位置；*offset* 可以为负值
- SEEK_END or 2 –流的末尾；*offset* 通常为负值

返回新的绝对位置。

2.7 新版功能: The SEEK_* constants

seekable()
Return `True` if the stream supports random access. If `False`, *seek()*, *tell()* and *truncate()* will raise *IOError*.

tell()
返回当前流的位置。

truncate(*size=None*)
Resize the stream to the given *size* in bytes (or the current position if *size* is not specified). The current stream position isn't changed. This resizing can extend or reduce the current file size. In case of extension, the contents of the new file area depend on the platform (on most systems, additional bytes are zero-filled, on Windows they're undetermined). The new file size is returned.

writable()
Return `True` if the stream supports writing. If `False`, `write()` and *truncate()* will raise *IOError*.

writelines(*lines*)
将行列表写入到流。不会添加行分隔符，因此通常所提供的每一行都带有末尾行分隔符。

__del__()
为对象销毁进行准备。*IOBase* 提供了此方法的默认实现，该实现会调用实例的*close()* 方法。

class io.**RawIOBase**
原始二进制 I/O 的基类。它继承自*IOBase*。没有公共构造器。

原始二进制 I/O 通常提供对下层 OS 设备或 API 的低层级访问，而不尝试将其封装到高层级的基元中（这是留给缓冲 I/O 和 Text I/O 的，将在下文中描述）。

In addition to the attributes and methods from *IOBase*, RawIOBase provides the following methods:

read(*n=-1*)
Read up to *n* bytes from the object and return them. As a convenience, if *n* is unspecified or -1, *readall()* is called. Otherwise, only one system call is ever made. Fewer than *n* bytes may be returned if the operating system call returns fewer than *n* bytes.

If 0 bytes are returned, and *n* was not 0, this indicates end of file. If the object is in non-blocking mode and no bytes are available, `None` is returned.

readall()
从流中读取并返回所有字节直到 EOF，如有必要将对流执行多次调用。

readinto(*b*)
Read up to len(b) bytes into *b*, and return the number of bytes read. The object *b* should be a pre-allocated, writable array of bytes, either *bytearray* or *memoryview*. If the object is in non-blocking mode and no bytes are available, `None` is returned.

write(*b*)
Write *b* to the underlying raw stream, and return the number of bytes written. The object *b* should be an array of bytes, either `bytes`, *bytearray*, or *memoryview*. The return value can be less than `len(b)`, depending on specifics of the underlying raw stream, and especially if it is in non-blocking mode. `None` is returned if the raw stream is set not to block and no single byte could be readily written to it. The caller may release or mutate *b* after this method returns, so the implementation should only access *b* during the method call.

class io.**BufferedIOBase**
支持某种缓冲的二进制流的基类。它继承自*IOBase*。没有公共构造器。

与*RawIOBase* 的主要差别在于*read()*, *readinto()* 和*write()* 等方法将（分别）尝试按照要求读取尽可能多的输入或是耗尽所有给定的输出，其代价是可能会执行一次以上的系统调用。

除此之外，那些方法还可能引发*BlockingIOError*，如果下层的原始数据流处于非阻塞模式并且无法接受或给出足够数据的话；不同于对应的*RawIOBase* 方法，它们将永远不会返回 `None`。

并且，*read()* 方法也没有转向*readinto()* 的默认实现。

典型的*BufferedIOBase* 实现不应当继承自*RawIOBase* 实现，而要包装一个该实现，正如*BufferedWriter* 和*BufferedReader* 所做的那样。

BufferedIOBase 在*IOBase* 的现有成员以外还提供或重载了下列方法和属性:

raw
由*BufferedIOBase* 处理的下层原始流 (*RawIOBase* 的实例)。它不是*BufferedIOBase* API 的组成部分并且不存在于某些实现中。

detach()
从缓冲区分离出下层原始流并将其返回。

在原始流被分离之后，缓冲区将处于不可用的状态。

某些缓冲区例如*BytesIO* 并无可从此方法返回的单独原始流的概念。它们将会引发*UnsupportedOperation*。

2.7 新版功能.

read(*n=-1*)
Read and return up to *n* bytes. If the argument is omitted, `None`, or negative, data is read and returned until EOF is reached. An empty bytes object is returned if the stream is already at EOF.

如果此参数为正值，并且下层原始流不可交互，则可能发起多个原始读取以满足字节计数（直至先遇到 EOF）。但对于可交互原始流，则将至多发起一个原始读取，并且简短的结果并不意味着已到达 EOF。

BlockingIOError 会在下层原始流不处于阻塞模式，并且当前没有可用数据时被引发。

read1(*n=-1*)
Read and return up to *n* bytes, with at most one call to the underlying raw stream’s *read()* method. This can be useful if you are implementing your own buffering on top of a *BufferedIOBase* object.

readinto(*b*)
Read up to len(b) bytes into *b*, and return the number of bytes read. The object *b* should be a pre-allocated, writable array of bytes, either *bytearray* or *memoryview*.

Like *read()*, multiple reads may be issued to the underlying raw stream, unless the latter is ‘interactive’.

BlockingIOError 会在下层原始流不处于阻塞模式，并且当前没有可用数据时被引发。

write(*b*)
Write *b*, and return the number of bytes written (always equal to `len(b)`, since if the write fails an *IOError* will be raised). The object *b* should be an array of bytes, either `bytes`, *bytearray*, or *memoryview*. Depending on the actual implementation, these bytes may be readily written to the underlying stream, or held in a buffer for performance and latency reasons.

当处于非阻塞模式时，如果需要将数据写入原始流但它无法在不阻塞的情况下接受所有数据则将引发*BlockingIOError*。

调用者可能会在此方法返回后释放或改变 *b*，因此该实现应当仅在方法调用期间访问 *b*。

15.2.3 原始文件 I/O

class io.**FileIO**(*name*, *mode='r'*, *closefd=True*)
FileIO 代表在 OS 层级上包含文件的字节数据。它实现了*RawIOBase* 接口（因而也实现了*IOBase* 接口）。

name 可以是以下两项之一:

- a string representing the path to the file which will be opened;
- an integer representing the number of an existing OS-level file descriptor to which the resulting *FileIO* object will give access.

The *mode* can be `'r'`, `'w'` or `'a'` for reading (default), writing, or appending. The file will be created if it doesn't exist when opened for writing or appending; it will be truncated when opened for writing. Add a `'+'` to the mode to allow simultaneous reading and writing.

该类的 `read()` (当附带正值参数调用时), `readinto()` 和 `write()` 方法将只执行一次系统调用。

In addition to the attributes and methods from *IOBase* and *RawIOBase*, *FileIO* provides the following data attributes and methods:

mode
构造函数中给定的模式。

name
文件名。当构造函数中没有给定名称时，这是文件的文件描述符。

15.2.4 缓冲流

相比原始 I/O，缓冲 I/O 流提供了针对 I/O 设备的更高层级接口.

class io.**BytesIO**([*initial_bytes*])
A stream implementation using an in-memory bytes buffer. It inherits *BufferedIOBase*.

The optional argument *initial_bytes* is a `bytes` object that contains initial data.

BytesIO 在继承自*BufferedIOBase* 和*IOBase* 的成员以外还提供或重载了下列方法:

getvalue()
Return `bytes` containing the entire contents of the buffer.

read1()
In *BytesIO*, this is the same as `read()`.

class io.**BufferedReader**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
一个提供对可读的序列型*RawIOBase* 对象更高层级访问的缓冲区。它继承自*BufferedIOBase*。当从此对象读取数据时，可能会从下层原始流请求更大量的数据，并存放到内部缓冲区中。接下来可以在后续读取时直接返回缓冲数据。

根据给定的可读 *raw* 流和 *buffer_size* 创建*BufferedReader* 的构造器。如果省略 *buffer_size*，则会使用*DEFAULT_BUFFER_SIZE*。

BufferedReader 在继承自*BufferedIOBase* 和*IOBase* 的成员以外还提供或重载了下列方法:

peek([*n*])
从流返回字节数据而不前移位置。完成此调用将至多读取一次原始流。返回的字节数量可能少于或多于请求的数量。

read([*n*])
Read and return *n* bytes, or if *n* is not given or negative, until EOF or if the read call would block in non-blocking mode.

read1(*n*)
Read and return up to *n* bytes with only one call on the raw stream. If at least one byte is buffered, only buffered bytes are returned. Otherwise, one raw stream read call is made.

class io.**BufferedWriter**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
A buffer providing higher-level access to a writeable, sequential *RawIOBase* object. It inherits *BufferedIOBase*. When writing to this object, data is normally held into an internal buffer. The buffer will be written out to the underlying *RawIOBase* object under various conditions, including:

- 当缓冲区对于所有挂起数据而言太小时；
- 当*flush()* 被调用时
- 当（为*BufferedRandom* 对象）请求 seek() 时；
- 当*BufferedWriter* 对象被关闭或销毁时。

该构造器会为给定的可写 *raw* 流创建一个*BufferedWriter*。如果未给定 *buffer_size*，则使用默认的*DEFAULT_BUFFER_SIZE*。

A third argument, *max_buffer_size*, is supported, but unused and deprecated.

BufferedWriter 在继承自*BufferedIOBase* 和*IOBase* 的成员以外还提供或重载了下列方法:

flush()
将缓冲区中保存的字节数据强制放入原始流。如果原始流发生阻塞则应当引发*BlockingIOError*。

write(*b*)
Write *b*, and return the number of bytes written. The object *b* should be an array of bytes, either bytes, *bytearray*, or *memoryview*. When in non-blocking mode, a *BlockingIOError* is raised if the buffer needs to be written out but the raw stream blocks.

class io.**BufferedRandom**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
A buffered interface to random access streams. It inherits *BufferedReader* and *BufferedWriter*, and further supports seek() and tell() functionality.

该构造器会为在第一个参数中给定的可查找原始流创建一个读取器和定稿器。如果省略 *buffer_size* 则使用默认的*DEFAULT_BUFFER_SIZE*。

A third argument, *max_buffer_size*, is supported, but unused and deprecated.

BufferedRandom is capable of anything *BufferedReader* or *BufferedWriter* can do.

class io.**BufferedRWPair**(*reader*, *writer*, *buffer_size=DEFAULT_BUFFER_SIZE*)
一个带缓冲的 I/O 对象，它将两个单向*RawIOBase* 对象–一个可读，另一个可写–组合为单个双向端点。它继承自*BufferedIOBase*。

reader 和 *writer* 分别是可读和可写的*RawIOBase* 对象。如果省略 *buffer_size* 则使用默认的*DEFAULT_BUFFER_SIZE*。

A fourth argument, *max_buffer_size*, is supported, but unused and deprecated.

BufferedRWPair 实现了*BufferedIOBase* 的所有方法，但*detach()* 除外，调用该方法将引发*UnsupportedOperation*。

> **警告:** *BufferedRWPair* 不会尝试同步访问其下层的原始流。你不应当将传给它与读取器和写入器相同的对象；而要改用*BufferedRandom*。

15.2.5 文本 I/O

class io.**TextIOBase**
Base class for text streams. This class provides a unicode character and line based interface to stream I/O. There is no `readinto()` method because Python's *unicode* strings are immutable. It inherits *IOBase*. There is no public constructor.

TextIOBase 在来自*IOBase* 的成员以外还提供或重载了以下数据属性和方法:

encoding
用于将流的字节串解码为字符串以及将字符串编码为字节串的编码格式名称。

errors
解码器或编码器的错误设置。

newlines
一个字符串、字符串元组或者 None，表示目前已经转写的新行。根据具体实现和初始构造器旗标的不同，此属性或许会不可用。

buffer
The underlying binary buffer (a *BufferedIOBase* instance) that *TextIOBase* deals with. This is not part of the *TextIOBase* API and may not exist on some implementations.

detach()
从*TextIOBase* 分离出下层二进制缓冲区并将其返回。

在下层缓冲区被分离后，*TextIOBase* 将处于不可用的状态。

Some *TextIOBase* implementations, like *StringIO*, may not have the concept of an underlying buffer and calling this method will raise *UnsupportedOperation*.

2.7 新版功能.

read(*n=-1*)
Read and return at most *n* characters from the stream as a single *unicode*. If *n* is negative or None, reads until EOF.

readline(*limit=-1*)
Read until newline or EOF and return a single `unicode`. If the stream is already at EOF, an empty string is returned.

If *limit* is specified, at most *limit* characters will be read.

seek(*offset*, *whence=SEEK_SET*)
将流位置改为给定的偏移位置 *offset*。具体行为取决于 *whence* 形参。*whence* 的默认值为 SEEK_SET。

- SEEK_SET 或 0: 从流的开始位置起查找（默认值）；*offset* 必须为*TextIOBase.tell()* 所返回的数值或为零。任何其他 *offset* 值都将导致未定义的行为。
- SEEK_CUR 或 1: "查找"到当前位置；*offset* 必须为零，表示无操作（所有其他值均不受支持）。
- SEEK_END 或 2: 查找到流的末尾；*offset* 必须为零（所有其他值均不受支持）。

以数字形式返回新的绝对位置。

2.7 新版功能: SEEK_* 常量.

tell()
以不透明数字形式返回当前流的位置。该数字通常并不代表下层二进制存储中对应的字节数。

write(*s*)
Write the *unicode* string *s* to the stream and return the number of characters written.

class io.**TextIOWrapper**(*buffer*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*)
一个基于*BufferedIOBase*二进制流的缓冲文本流。它继承自*TextIOBase*。

encoding gives the name of the encoding that the stream will be decoded or encoded with. It defaults to *locale.getpreferredencoding()*.

errors is an optional string that specifies how encoding and decoding errors are to be handled. Pass `'strict'` to raise a *ValueError* exception if there is an encoding error (the default of `None` has the same effect), or pass `'ignore'` to ignore errors. (Note that ignoring encoding errors can lead to data loss.) `'replace'` causes a replacement marker (such as `'?'`) to be inserted where there is malformed data. When writing, `'xmlcharrefreplace'` (replace with the appropriate XML character reference) or `'backslashreplace'` (replace with backslashed escape sequences) can be used. Any other error handling name that has been registered with *codecs.register_error()* is also valid.

newline 控制行结束符处理方式。它可以为 `None`, `''`, `'\n'`, `'\r'` 和 `'\r\n'`。其工作原理如下:

- On input, if *newline* is `None`, *universal newlines* mode is enabled. Lines in the input can end in `'\n'`, `'\r'`, or `'\r\n'`, and these are translated into `'\n'` before being returned to the caller. If it is `''`, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- On output, if *newline* is `None`, any `'\n'` characters written are translated to the system default line separator, *os.linesep*. If *newline* is `''`, no translation takes place. If *newline* is any of the other legal values, any `'\n'` characters written are translated to the given string.

如果 *line_buffering* 为 `True`，则当一个写入调用包含换行符或回车时将会应用 `flush()`。

TextIOWrapper provides one attribute in addition to those of *TextIOBase* and its parents:

line_buffering
是否启用行缓冲。

class io.**StringIO**(*initial_value=u''*, *newline=u'\n'*)
An in-memory stream for unicode text. It inherits *TextIOWrapper*.

缓冲区的初始值可通过提供 *initial_value* 来设置。如果启用了行结束符转写，换行将以*write()* 所用的方式被编码。数据流位置将被设为缓冲区的开头。

newline 参数的规则与*TextIOWrapper* 所用的一致。默认规则是仅将 `\n` 字符视为行结束符并且不执行换行符转写。如果 *newline* 设为 `None`，在所有平台上换行符都将被写入为 `\n`，但当读取时仍然会执行通用换行编码格式。

StringIO provides this method in addition to those from *TextIOWrapper* and its parents:

getvalue()
Return a `unicode` containing the entire contents of the buffer at any time before the *StringIO* object's `close()` method is called. Newlines are decoded as if by *read()*, although the stream position is not changed.

用法示例:

```
import io

output = io.StringIO()
output.write(u'First line.\n')
output.write(u'Second line.\n')

# Retrieve file contents -- this will be
# u'First line.\nSecond line.\n'
contents = output.getvalue()
```

(下页继续)

(续上页)

```
# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

class io.**IncrementalNewlineDecoder**

用于在*universal newlines*模式下解码换行符的辅助编解码器。它继承自*codecs.IncrementalDecoder*。

15.2.6 Advanced topics

Here we will discuss several advanced topics pertaining to the concrete I/O implementations described above.

性能

二进制 I/O

By reading and writing only large chunks of data even when the user asks for a single byte, buffered I/O is designed to hide any inefficiency in calling and executing the operating system's unbuffered I/O routines. The gain will vary very much depending on the OS and the kind of I/O which is performed (for example, on some contemporary OSes such as Linux, unbuffered disk I/O can be as fast as buffered I/O). The bottom line, however, is that buffered I/O will offer you predictable performance regardless of the platform and the backing device. Therefore, it is most always preferable to use buffered I/O rather than unbuffered I/O.

文本 I/O

Text I/O over a binary storage (such as a file) is significantly slower than binary I/O over the same storage, because it implies conversions from unicode to binary data using a character codec. This can become noticeable if you handle huge amounts of text data (for example very large log files). Also, `TextIOWrapper.tell()` and `TextIOWrapper.seek()` are both quite slow due to the reconstruction algorithm used.

StringIO, however, is a native in-memory unicode container and will exhibit similar speed to *BytesIO*.

多线程

FileIO objects are thread-safe to the extent that the operating system calls (such as `read(2)` under Unix) they are wrapping are thread-safe too.

二进制缓冲对象（例如*BufferedReader*, *BufferedWriter*, *BufferedRandom* 和*BufferedRWPair*）使用锁来保护其内部结构；因此，可以安全地一次从多个线程中调用它们。

TextIOWrapper 对象不再是线程安全的。

可重入性

Binary buffered objects (instances of *BufferedReader*, *BufferedWriter*, *BufferedRandom* and *BufferedRWPair*) are not reentrant. While reentrant calls will not happen in normal situations, they can arise if you are doing I/O in a *signal* handler. If it is attempted to enter a buffered object again while already being accessed *from the same thread*, then a *RuntimeError* is raised.

上面的内容隐含地扩展到文本文件，因为*open()* 函数会把缓冲对象包装在*TextIOWrapper* 中。这包括标准流，因此也会影响内置函数*print()* 。

15.3 time —时间的访问和转换

该模块提供了各种时间相关的函数。相关功能还可以参阅*datetime* 和*calendar* 模块。

尽管此模块始终可用，但并非所有平台上都提供所有功能。此模块中定义的大多数函数是调用了所在平台 C 语言库的同名函数。因为这些函数的语义因平台而异，所以使用时最好查阅平台相关文档。

下面是一些术语和惯例的解释.

- The *epoch* is the point where the time starts. On January 1st of that year, at 0 hours, the "time since the epoch" is zero. For Unix, the epoch is 1970. To find out what the epoch is, look at `gmtime(0)`.
- The functions in this module do not handle dates and times before the epoch or far in the future. The cut-off point in the future is determined by the C library; for Unix, it is typically in 2038.
- **Year 2000 (Y2K) issues**: Python depends on the platform's C library, which generally doesn't have year 2000 issues, since all dates and times are represented internally as seconds since the epoch. Functions accepting a *struct_time* (see below) generally require a 4-digit year. For backward compatibility, 2-digit years are supported if the module variable `accept2dyear` is a non-zero integer; this variable is initialized to `1` unless the environment variable `PYTHONY2K` is set to a non-empty string, in which case it is initialized to `0`. Thus, you can set `PYTHONY2K` to a non-empty string in the environment to require 4-digit years for all year input. When 2-digit years are accepted, they are converted according to the POSIX or X/Open standard: values 69-99 are mapped to 1969-1999, and values 0–68 are mapped to 2000–2068. Values 100–1899 are always illegal.
- UTC 是协调世界时（以前称为格林威治标准时间，或 GMT）。缩写 UTC 不是错误，而是英语和法语之间的妥协。
- DST 是夏令时，在一年中的一部分时间（通常）调整时区一小时。DST 规则很神奇（由当地法律确定），并且每年都会发生变化。C 库有一个包含本地规则的表（通常是从系统文件中读取以获得灵活性），并且在这方面是 True Wisdom 的唯一来源。
- 各种实时函数的精度可能低于表示其值或参数的单位所建议的精度。例如，在大多数 Unix 系统上，时钟"ticks"仅为每秒 50 或 100 次。
- 另一方面，*time()* 和*sleep()* 的精度优于它们的 Unix 等价物：时间表示为浮点数，*time()* 返回最准确的时间（使用 Unix `gettimeofday()` 如果可用），并且*sleep()* 将接受非零分数的时间（Unix `select()` 用于实现此功能，如果可用）。
- The time value as returned by *gmtime()*, *localtime()*, and *strptime()*, and accepted by *asctime()*, *mktime()* and *strftime()*, may be considered as a sequence of 9 integers. The return values of *gmtime()*, *localtime()*, and *strptime()* also offer attribute names for individual fields.

  请参阅*struct_time* 以获取这些对象的描述。

  在 2.2 版更改: The time value sequence was changed from a tuple to a *struct_time*, with the addition of attribute names for the fields.
- 使用以下函数在时间表示之间进行转换：

从	到	使用
seconds since the epoch	UTC 的*struct_time*	*gmtime()*
seconds since the epoch	本地时间的*struct_time*	*localtime()*
UTC 的*struct_time*	seconds since the epoch	*calendar.timegm()*
本地时间的*struct_time*	seconds since the epoch	*mktime()*

The module defines the following functions and data items:

time.**accept2dyear**
Boolean value indicating whether two-digit year values will be accepted. This is true by default, but will be set to false if the environment variable PYTHONY2K has been set to a non-empty string. It may also be modified at run time.

time.**altzone**
The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero.

time.**asctime**([*t*])
Convert a tuple or *struct_time* representing a time as returned by *gmtime()* or *localtime()* to a 24-character string of the following form: 'Sun Jun 20 23:21:05 1993'. If *t* is not provided, the current time as returned by *localtime()* is used. Locale information is not used by *asctime()*.

注解: Unlike the C function of the same name, there is no trailing newline.

在 2.1 版更改: Allowed *t* to be omitted.

time.**clock**()
On Unix, return the current processor time as a floating point number expressed in seconds. The precision, and in fact the very definition of the meaning of "processor time", depends on that of the C function of the same name, but in any case, this is the function to use for benchmarking Python or timing algorithms.

在 Windows 上，此函数返回自第一次调用此函数以来经过的 wallclock 秒数，作为浮点数，基于 Win32 函数 QueryPerformanceCounter()。分辨率通常优于 1 微秒。

time.**ctime**([*secs*])
将以自 epoch 开始的秒数表示的时间转换为表示本地时间的字符串。如果未提供 *secs* 或为*None*，则使用*time()* 所返回的当前时间。ctime(secs) 相当于 asctime(localtime(secs))。区域信息不会被*ctime()* 使用。

在 2.1 版更改: Allowed *secs* to be omitted.

在 2.4 版更改: If *secs* is *None*, the current time is used.

time.**daylight**
Nonzero if a DST timezone is defined.

time.**gmtime**([*secs*])
将以自 epoch 开始的秒数表示的时间转换为 UTC 的*struct_time*，其中 dst 标志始终为零。如果未提供 *secs* 或为*None*，则使用*time()* 所返回的当前时间。一秒以内的小数将被忽略。有关*struct_time* 对象的说明请参见上文。有关此函数的逆操作请参阅*calendar.timegm()*。

在 2.1 版更改: Allowed *secs* to be omitted.

在 2.4 版更改: If *secs* is *None*, the current time is used.

time.**localtime**([*secs*])
与*gmtime()* 相似但转换为当地时间。如果未提供 *secs* 或为*None*，则使用由*time()* 返回的当前时间。当 DST 适用于给定时间时，dst 标志设置为 1。

在 2.1 版更改: Allowed *secs* to be omitted.

在 2.4 版更改: If *secs* is `None`, the current time is used.

`time.`**`mktime`**`(`*t*`)`

这是`localtime()` 的反函数。它的参数是`struct_time` 或者完整的 9 元组（因为需要 dst 标志；如果它是未知的则使用 `-1` 作为 dst 标志），它表示 *local* 的时间，而不是 UTC 。它返回一个浮点数，以便与`time()` 兼容。如果输入值不能表示为有效时间，则`OverflowError` 或`ValueError` 将被引发（这取决于 Python 或底层 C 库是否捕获到无效值）。它可以生成时间的最早日期取决于平台。

`time.`**`sleep`**`(`*secs*`)`

Suspend execution of the current thread for the given number of seconds. The argument may be a floating point number to indicate a more precise sleep time. The actual suspension time may be less than that requested because any caught signal will terminate the `sleep()` following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount because of the scheduling of other activity in the system.

`time.`**`strftime`**`(`*format*`[`, *t*`]`)

Convert a tuple or `struct_time` representing a time as returned by `gmtime()` or `localtime()` to a string as specified by the *format* argument. If *t* is not provided, the current time as returned by `localtime()` is used. *format* must be a string. `ValueError` is raised if any field in *t* is outside of the allowed range. `strftime()` returns a locale dependent byte string; the result may be converted to unicode by doing `strftime(<myformat>).decode(locale.getlocale()[1])`.

在 2.1 版更改: Allowed *t* to be omitted.

在 2.4 版更改: `ValueError` raised if a field in *t* is out of range.

在 2.5 版更改: 0 is now a legal argument for any position in the time tuple; if it is normally illegal the value is forced to a correct one.

以下指令可以嵌入 *format* 字符串中。它们显示时没有可选的字段宽度和精度规范，并被`strftime()` 结果中的指示字符替换：

指令	含义	注释
%a	本地化的缩写星期中每日的名称。	
%A	本地化的星期中每日的完整名称。	
%b	本地化的月缩写名称。	
%B	本地化的月完整名称。	
%c	本地化的适当日期和时间表示。	
%d	十进制数 [01,31] 表示的月中日。	
%H	十进制数 [00,23] 表示的小时（24 小时制）。	
%I	十进制数 [01,12] 表示的小时（12 小时制）。	
%j	十进制数 [001,366] 表示的年中日。	
%m	十进制数 [01,12] 表示的月。	
%M	十进制数 [00,59] 表示的分钟。	
%p	本地化的 AM 或 PM 。	(1)
%S	十进制数 [00,61] 表示的秒。	(2)
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.	(3)
%w	十进制数 [0(星期日),6] 表示的周中日。	
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.	(3)
%x	本地化的适当日期表示。	
%X	本地化的适当时间表示。	
%y	十进制数 [00,99] 表示的没有世纪的年份。	
%Y	十进制数表示的带世纪的年份。	
%Z	时区名称（如果不存在时区，则不包含字符）。	
%%	字面的 '%' 字符。	

注释:

(1) 当与*strptime()* 函数一起使用时，如果使用 %I 指令来解析小时，%p 指令只影响输出小时字段。

(2) The range really is 0 to 61; this accounts for leap seconds and the (very rare) double leap seconds.

(3) 当与*strptime()* 函数一起使用时，%U 和 %W 仅用于指定星期几和年份的计算。

下面是一个示例，一个与 **RFC 2822** Internet 电子邮件标准以兼容的日期格式。[1]

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

某些平台可能支持其他指令，但只有此处列出的指令具有 ANSI C 标准化的含义。要查看平台支持的完整格式代码集，请参阅 *strftime(3)* 文档。

在某些平台上，可选的字段宽度和精度规范可以按照以下顺序紧跟在指令的初始 '%' 之后；这也不可移植。字段宽度通常为 2，除了 %j ，它是 3。

time.**strptime**(*string*[, *format*])

Parse a string representing a time according to a format. The return value is a *struct_time* as returned by *gmtime()* or *localtime()*.

The *format* parameter uses the same directives as those used by *strftime()*; it defaults to "%a %b %d %H:%M:%S %Y" which matches the formatting returned by *ctime()*. If *string* cannot be parsed according

[1] 现在不推荐使用 %Z ，但是所有 ANSI C 库都不支持扩展为首选小时/分钟偏移量的 “%z“转义符。此外，严格的 1982 年原始 **RFC 822** 标准要求两位数的年份（%y 而不是%Y），但是实际在 2000 年之前很久就转移到了 4 位数年。之后，**RFC 822** 已经废弃了，4 位数的年份首先被推荐 **RFC 1123** ，然后被 **RFC 2822** 强制执行。

to *format*, or if it has excess data after parsing, *ValueError* is raised. The default values used to fill in any missing data when more accurate values cannot be inferred are (1900, 1, 1, 0, 0, 0, 0, 1, -1).

例如:

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                 tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

支持 %Z 指令是基于 tzname 中包含的值以及 daylight 是否为真。因此，它是特定于平台的，除了识别始终已知的 UTC 和 GMT （并且被认为是非夏令时时区）。

仅支持文档中指定的指令。因为每个平台都实现了 strftime()，它有时会提供比列出的指令更多的指令。但是 strptime() 独立于任何平台，因此不一定支持所有未记录为支持的可用指令。

class time.**struct_time**

返回的时间值序列的类型为*gmtime()*、*localtime()* 和*strptime()*。它是一个带有*named tuple*接口的对象：可以通过索引和属性名访问值。存在以下值：

索引	属性	值
0	tm_year	（例如，1993）
1	tm_mon	range [1, 12]
2	tm_mday	range [1, 31]
3	tm_hour	range [0, 23]
4	tm_min	range [0, 59]
5	tm_sec	range [0, 61]；见*strftime()* 介绍中的 **(2)**
6	tm_wday	range [0, 6]，周一为 0
7	tm_yday	range [1, 366]
8	tm_isdst	0, 1 或 -1；如下所示

2.2 新版功能.

Note that unlike the C structure, the month value is a range of [1, 12], not [0, 11]. A year value will be handled as described under *Year 2000 (Y2K) issues* above.

在调用*mktime()* 时，tm_isdst 可以在夏令时生效时设置为 1，而在夏令时不生效时设置为 0。值-1 表示这是未知的，并且通常会导致填写正确的状态。

当一个长度不正确的元组被传递给期望*struct_time* 的函数，或者具有错误类型的元素时，会引发*TypeError*。

time.**time**()

Return the time in seconds since the epoch as a floating point number. Note that even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second. While this function normally returns non-decreasing values, it can return a lower value than a previous call if the system clock has been set back between the two calls.

time.**timezone**

The offset of the local (non-DST) timezone, in seconds west of UTC (negative in most of Western Europe, positive in the US, zero in the UK).

time.**tzname**

A tuple of two strings: the first is the name of the local non-DST timezone, the second is the name of the local DST timezone. If no DST timezone is defined, the second string should not be used.

time.**tzset**()

重置库例程使用的时间转换规则。环境变量 TZ 指定如何完成。它还将设置变量 tzname （来自 TZ 环

境变量），timezone （UTC 的西部非 DST 秒），altzone （UTC 以西的 DST 秒）和 daylight （如果此时区没有任何夏令时规则则为 0，如果有夏令时适用的时间，无论过去、现在或未来，则为非零）。

2.3 新版功能.

Availability: Unix.

注解： 虽然在很多情况下，更改 TZ 环境变量而不调用*tzset()* 可能会影响函数的输出，例如*localtime()* ，不应该依赖此行为。

TZ 不应该包含空格。

TZ 环境变量的标准格式是（为了清晰起见，添加了空格）：

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

组件的位置是：

std 和 dst 三个或更多字母数字，给出时区缩写。这些将传到 time.tzname

offset 偏移量的形式为：± hh[:mm[:ss]] 。这表示添加到达 UTC 的本地时间的值。如果前面有‘-‘，则时区位于本初子午线的东边；否则，在它是西边。如果 dst 之后没有偏移，则假设夏令时比标准时间提前一小时。

start[/time], end[/time] 指示何时更改为 DST 和从 DST 返回。开始日期和结束日期的格式为以下之一：

Jn Julian 日 *n* （1 <= *n* <= 365）。闰日不计算在内，因此在所有年份中，2 月 28 日是第 59 天，3 月 1 日是第 60 天。

n 从零开始的 Julian 日（0 <= *n* <= 365）。闰日计入，可以引用 2 月 29 日。

Mm.n.d The *d*’ th day (0 <= *d* <= 6) or week *n* of month *m* of the year (1 <= *n* <= 5, 1 <= *m* <= 12, where week 5 means “the last *d* day in month *m*” which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*’ th day occurs. Day zero is Sunday.

time 的格式与 offset 的格式相同，但不允许使用前导符号（‘-‘或‘+’）。如果没有给出时间，则默认值为 02:00:00。

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

在许多 Unix 系统（包括 *BSD ， Linux ， Solaris 和 Darwin 上），使用系统的区域信息（*tzfile(5)* ）数据库来指定时区规则会更方便。为此，将 TZ 环境变量设置为所需时区数据文件的路径，相对于系统‘zoneinfo’时区数据库的根目录，通常位于 /usr/share/zoneinfo 。例如，'US/Eastern' 、'Australia/Melbourne' 、'Egypt' 或 'Europe/Amsterdam'。

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
```

(下页继续)

(续上页)

```
>>> time.tzname
('EET', 'EEST')
```

参见:

模块*datetime* 更多面向对象的日期和时间接口。

模块*locale* 国际化服务。区域设置会影响*strftime()* 和*strptime()* 中许多格式说明符的解析。

模块*calendar* 一般日历相关功能。这个模块的 timegm() 是函数*gmtime()* 的反函数。

备注

15.4 argparse —命令行选项、参数和子命令解析器

2.7 新版功能.

源代码: Lib/argparse.py

教程

此页面包含该 API 的参考信息。有关 Python 命令行解析更细致的介绍，请参阅 argparse 教程。

argparse 模块可以让人轻松编写用户友好的命令行接口。程序定义它需要的参数，然后*argparse* 将弄清如何从*sys.argv* 解析出那些参数。*argparse* 模块还会自动生成帮助和使用手册，并在用户给程序传入无效参数时报出错误信息。

15.4.1 示例

以下代码是一个 Python 程序，它获取一个整数列表并计算总和或者最大值:

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print args.accumulate(args.integers)
```

假设上面的 Python 代码保存在名为 prog.py 的文件中，它可以在命令行运行并提供有用的帮助信息:

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
```

(下页继续)

(续上页)

```
 N           an integer for the accumulator

optional arguments:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

当使用适当的参数运行时，它会输出命令行传入整数的总和或者最大值：

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

如果传入无效参数，则会报出错误：

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

以下部分将引导你完成这个示例。

创建一个解析器

使用*argparse*的第一步是创建一个*ArgumentParser*对象：

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

*ArgumentParser*对象包含将命令行解析成 Python 数据类型所需的全部信息。

添加参数

给一个*ArgumentParser*添加程序参数信息是通过调用*add_argument()*方法完成的。通常，这些调用指定*ArgumentParser*如何获取命令行字符串并将其转换为对象。这些信息在*parse_args()*调用时被存储和使用。例如：

```
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', action='store_const',
...                     const=sum, default=max,
...                     help='sum the integers (default: find the max)')
```

然后，调用*parse_args()*将返回一个具有 integers 和 accumulate 两个属性的对象。integers 属性将是一个包含一个或多个整数的列表，而 accumulate 属性当命令行中指定了 --sum 参数时将是*sum()*函数，否则则是*max()*函数。

解析参数

ArgumentParser 通过*parse_args()* 方法解析参数。它将检查命令行，把每个参数转换为适当的类型然后调用相应的操作。在大多数情况下，这意味着一个简单的*Namespace* 对象将从命令行解析出的属性构建：

```
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

在脚本中，通常*parse_args()* 会被不带参数调用，而*ArgumentParser* 将自动从*sys.argv* 中确定命令行参数。

15.4.2 ArgumentParser 对象

class argparse.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=argparse.HelpFormatter*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*)

创建一个新的*ArgumentParser* 对象。所有的参数都应当作为关键字参数传入。每个参数在下面都有它更详细的描述，但简而言之，它们是：

- *prog* - 程序的名称（默认值：`sys.argv[0]`）
- *usage* - 描述程序用途的字符串（默认值：从添加到解析器的参数生成）
- *description* - 在参数帮助文档之前显示的文本（默认值：无）
- *epilog* - 在参数帮助文档之后显示的文本（默认值：无）
- *parents* - 一个*ArgumentParser* 对象的列表，它们的参数也应包含在内
- *formatter_class* - 用于自定义帮助文档输出格式的类
- *prefix_chars* - 可选参数的前缀字符集合（默认值：'-'）
- *fromfile_prefix_chars* - 当需要从文件中读取其他参数时，用于标识文件名的前缀字符集合（默认值：`None`）
- *argument_default* - 参数的全局默认值（默认值：`None`）
- *conflict_handler* - 解决冲突选项的策略（通常是不必要的）
- *add_help* - 为解析器添加一个 `-h/--help` 选项（默认值：`True`）

以下部分描述这些参数如何使用。

prog

默认情况下，*ArgumentParser* 对象使用 `sys.argv[0]` 来确定如何在帮助消息中显示程序名称。这一默认值几乎总是可取的，因为它将使帮助消息与从命令行调用此程序的方式相匹配。例如，对于有如下代码的名为 `myprogram.py` 的文件：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

该程序的帮助信息将显示 `myprogram.py` 作为程序名称（无论程序从何处被调用）：

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
$ cd ..
$ python subdir/myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

要更改这样的默认行为，可以使用 prog= 参数为*ArgumentParser* 指定另一个值：

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

optional arguments:
 -h, --help  show this help message and exit
```

需要注意的是，无论是从 sys.argv[0] 或是从 prog= 参数确定的程序名称，都可以在帮助消息里通过 %(prog)s 格式说明符来引用。

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo of the myprogram program
```

usage

默认情况下，*ArgumentParser* 根据它包含的参数来构建用法消息：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

可以通过 usage= 关键字参数覆盖这一默认消息：

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
```

(下页继续)

(续上页)

```
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
 bar           bar help

optional arguments:
 -h, --help    show this help message and exit
 --foo [FOO]   foo help
```

在用法消息中可以使用 `%(prog)s` 格式说明符来填入程序名称。

描述

大多数对*ArgumentParser* 构造方法的调用都会使用 `description=` 关键字参数。这个参数简要描述这个程度做什么以及怎么做。在帮助消息中，这个描述会显示在命令行用法字符串和各种参数的帮助消息之间:

```
>>> parser = argparse.ArgumentParser(description='A foo that bars')
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit
```

在默认情况下，description 将被换行以便适应给定的空间。如果想改变这种行为，见*formatter_class* 参数。

epilog

一些程序喜欢在 description 参数后显示额外的对程序的描述。这种文字能够通过给*ArgumentParser*:: 提供 `epilog=` 参数而被指定。

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit

And that's how you'd foo a bar
```

和*description* 参数一样，`epilog=` text 在默认情况下会换行，但是这种行为能够被调整通过提供*formatter_class* 参数给 `ArgumentParse`.

parents

有些时候，少数解析器会使用同一系列参数。单个解析器能够通过提供 parents= 参数给*ArgumentParser* 而使用相同的参数而不是重复这些参数的定义。parents= 参数使用*ArgumentParser* 对象的列表，从它们那里收集所有的位置和可选的行为，然后将这写行为加到正在构建的*ArgumentParser* 对象。

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

请注意大多数父解析器会指定 add_help=False . 否则，ArgumentParse 将会看到两个 -h/--help 选项（一个在父参数中一个在子参数中）并且产生一个错误。

注解： 你在传 “parents=“给那些解析器时必须完全初始化它们。如果你在子解析器之后改变父解析器是，这些改变不会反映在子解析器上。

formatter_class

ArgumentParser objects allow the help formatting to be customized by specifying an alternate formatting class. Currently, there are three such classes:

class argparse.**RawDescriptionHelpFormatter**
class argparse.**RawTextHelpFormatter**
class argparse.**ArgumentDefaultsHelpFormatter**

The first two allow more control over how textual descriptions are displayed, while the last automatically adds information about argument default values.

By default, *ArgumentParser* objects line-wrap the *description* and *epilog* texts in command-line help messages:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...             but that is okay''',
...     epilog='''
...             likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay

optional arguments:
 -h, --help  show this help message and exit
```

(下页继续)

(续上页)

```
likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

传*RawDescriptionHelpFormatter* 给 formatter_class= 表示*description* 和*epilog* 已经被正确的格式化了，不能在命令行中被自动换行:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         --------------------------------
...             I have indented it
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
   I have indented it
   exactly the way
   I want it

optional arguments:
 -h, --help  show this help message and exit
```

RawTextHelpFormatter 保留所有种类文字的空格，包括参数的描述。然而，多重的新行会被替换成一行。如果你想保留多重的空白行，可以在新行之间加空格。

The other formatter class available, *ArgumentDefaultsHelpFormatter*, will add information about the default value of each of the arguments:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar [bar ...]]

positional arguments:
 bar         BAR! (default: [1, 2, 3])

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   FOO! (default: 42)
```

prefix_chars

许多命令行会使用 - 当作前缀，比如 -f/--foo。如果解析器需要支持不同的或者额外的字符，比如像 +f 或者 /foo 的选项，可以在参数解析构建器中使用 prefix_chars= 参数。

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

The prefix_chars= 参数默认使用 '-'. 支持一系列字符，但是不包括 - ，这样会产生不被允许的 -f/--foo 选项。

fromfile_prefix_chars

有些时候，先举个例子，当处理一个特别长的参数列表的时候，把它存入一个文件中而不是在命令行打出来会很有意义。如果 fromfile_prefix_chars= 参数提供给*ArgumentParser* 构造函数，之后所有类型的字符的参数都会被当成文件处理，并且会被文件包含的参数替代。举个栗子:

```
>>> with open('args.txt', 'w') as fp:
...     fp.write('-f\nbar')
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

从文件读取的参数在默认情况下必须一个一行（但是可参见*convert_arg_line_to_args()*）并且它们被视为与命令行上的原始文件引用参数位于同一位置。所以在以上例子中，['-f', 'foo', '@args.txt'] 的表示和 ['-f', 'foo', '-f', 'bar'] 的表示相同。

fromfile_prefix_chars= 参数默认为 None，意味着参数不会被当作文件对待。

argument_default

一般情况下，参数默认会通过设置一个默认到*add_argument()* 或者调用带一组指定键值对的*ArgumentParser.set_defaults()* 方法。但是有些时候，为参数指定一个普遍适用的解析器会更有用。这能够通过传输 argument_default= 关键词参数给*ArgumentParser* 来完成。举个栗子，要全局禁止在*parse_args()* 中创建属性，我们提供 argument_default=SUPPRESS:

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

conflict_handler

ArgumentParser 对象不允许在相同选项字符串下有两种行为。默认情况下，*ArgumentParser* 对象会产生一个异常如果去创建一个正在使用的选项字符串参数。

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
 ..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

有些时候（例如：使用*parents*），重写旧的有相同选项字符串的参数会更有用。为了产生这种行为，'resolve' 值可以提供给*ArgumentParser* 的 conflict_handler= 参数：

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 -f FOO      old foo help
 --foo FOO   new foo help
```

注意*ArgumentParser* 对象只能移除一个行为如果它所有的选项字符串都被重写。所以，在上面的例子中，旧的 -f/--foo 行为回合 -f 行为保持一样，因为只有 --foo 选项字符串被重写。

add_help

默认情况下，ArgumentParser 对象添加一个简单的显示解析器帮助信息的选项。举个栗子，考虑一个名为 myprogram.py 的文件包含如下代码：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

如果 -h or --help 在命令行中被提供，参数解析器帮助信息会打印：

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

有时候可能会需要关闭额外的帮助信息。这可以通过在*ArgumentParser* 中设置 add_help= 参数为 False 来实现。

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]
```

(下页继续)

(续上页)

```
optional arguments:
 --foo FOO  foo help
```

帮助选项一般为 -h/--help。如果 prefix_chars= 被指定并且没有包含 - 字符，在这种情况下，-h --help 不是有效的选项。此时，prefix_chars 的第一个字符将用作帮助选项的前缀。

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

optional arguments:
  +h, ++help  show this help message and exit
```

15.4.3 add_argument() 方法

ArgumentParser.**add_argument**(*name or flags...*[, *action*][, *nargs*][, *const*][, *default*][, *type*][, *choices*][, *required*][, *help*][, *metavar*][, *dest*])

定义单个的命令行参数应当如何解析。每个形参都在下面有它自己更多的描述，长话短说有：

- *name or flags* - 一个命名或者一个选项字符串的列表，例如 foo 或 -f, --foo。
- *action* - 当参数在命令行中出现时使用的动作基本类型。
- *nargs* - 命令行参数应当消耗的数目。
- *const* - 被一些*action* 和*nargs* 选择所需求的常数。
- *default* - 当参数未在命令行中出现时使用的值。
- *type* - 命令行参数应当被转换成的类型。
- *choices* - 可用的参数的容器。
- *required* - 此命令行选项是否可省略（仅选项可用）。
- *help* - 一个此选项作用的简单描述。
- *metavar* - 在使用方法消息中使用的参数值示例。
- *dest* - 被添加到*parse_args()* 所返回对象上的属性名。

以下部分描述这些参数如何使用。

name or flags

add_argument() 方法必须知道它是否是一个选项，例如 -f 或 --foo，或是一个位置参数，例如一组文件名。第一个传递给*add_argument()* 的参数必须是一系列 flags 或者是一个简单的参数名。例如，可以选项可以被这样创建:

```
>>> parser.add_argument('-f', '--foo')
```

而位置参数可以这么创建:

```
>>> parser.add_argument('bar')
```

当*parse_args()* 被调用，选项会以 - 前缀识别，剩下的参数则会被假定为位置参数:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: too few arguments
```

action

ArgumentParser 对象将命令行参数与动作相关联。这些动作可以做与它们相关联的命令行参数的任何事，尽管大多数动作只是简单的向*parse_args()* 返回的对象上添加属性。action 命名参数指定了这个命令行参数应当如何处理。供应的动作有：

- 'store' - 存储参数的值。这是默认的动作。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args('--foo 1'.split())
Namespace(foo='1')
```

- 'store_const' - 存储被*const* 命名参数指定的值。'store_const' 动作通常用在选项中来指定一些标志。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_const', const=42)
>>> parser.parse_args(['--foo'])
Namespace(foo=42)
```

- 'store_true' and 'store_false' - These are special cases of 'store_const' using for storing the values True and False respectively. In addition, they create default values of False and True respectively. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('--bar', action='store_false')
>>> parser.add_argument('--baz', action='store_false')
>>> parser.parse_args('--foo --bar'.split())
Namespace(bar=False, baz=True, foo=True)
```

- 'append' - 存储一个列表，并且将每个参数值追加到列表中。在允许多次使用选项时很有用。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

- 'append_const' - 这存储一个列表，并将*const* 命名参数指定的值追加到列表中。（注意*const* 命名参数默认为 None。）“’ append_const’“ 动作一般在多个参数需要在同一列表中存储常数时会有用。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const', const=str)
```

(下页继续)

(续上页)

```
>>> parser.add_argument('--int', dest='types', action='append_const', const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<type 'str'>, <type 'int'>])
```

- `'count'` - 计算一个关键字参数出现的数目或次数。例如，对于一个增长的详情等级来说有用:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count')
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

- `'help'` - 打印所有当前解析器中的选项和参数的完整帮助信息，然后退出。默认情况下，一个 help 动作会被自动加入解析器。关于输出是如何创建的，参与*ArgumentParser*。
- `'version'` - 期望有一个 `version=` 命名参数在*add_argument()* 调用中，并打印版本信息并在调用后退出:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

您还可以通过传递 Action 子类或实现相同接口的其他对象来指定任意操作。建议的方法是扩展*Action*，覆盖 `__call__` 方法和可选的 `__init__` 方法。

一个自定义动作的例子:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super(FooAction, self).__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         print '%r %r %r' % (namespace, values, option_string)
...         setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

更多描述，见*Action*。

nargs

ArgumentParser 对象通常关联一个单独的命令行参数到一个单独的被执行的动作。nargs 命名参数关联不同数目的命令行参数到单一动作。支持的值有：

- N （一个整数）。命令行中的 N 个参数会被聚集到一个列表中。例如：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs=2)
>>> parser.add_argument('bar', nargs=1)
>>> parser.parse_args('c --foo a b'.split())
Namespace(bar=['c'], foo=['a', 'b'])
```

  注意 nargs=1 会产生一个单元素列表。这和默认的元素本身是不同的。

- '?'。如果可能的话，会从命令行中消耗一个参数，并产生一个单一项。如果当前没有命令行参数，则会产生*default* 值。注意，对于选项，有另外的用例 - 选项字符串出现但没有跟随命令行参数，则会产生*const* 值。一些说用用例：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='?', const='c', default='d')
>>> parser.add_argument('bar', nargs='?', default='d')
>>> parser.parse_args(['XX', '--foo', 'YY'])
Namespace(bar='XX', foo='YY')
>>> parser.parse_args(['XX', '--foo'])
Namespace(bar='XX', foo='c')
>>> parser.parse_args([])
Namespace(bar='d', foo='d')
```

  nargs='?' 的一个更普遍用法是允许可选的输入或输出文件：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', nargs='?', type=argparse.FileType('r'),
...                     default=sys.stdin)
>>> parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'),
...                     default=sys.stdout)
>>> parser.parse_args(['input.txt', 'output.txt'])
Namespace(infile=<open file 'input.txt', mode 'r' at 0x...>,
          outfile=<open file 'output.txt', mode 'w' at 0x...>)
>>> parser.parse_args([])
Namespace(infile=<open file '<stdin>', mode 'r' at 0x...>,
          outfile=<open file '<stdout>', mode 'w' at 0x...>)
```

- '*'。所有当前命令行参数被聚集到一个列表中。注意通过 nargs='*' 来实现多个位置参数通常没有意义，但是多个选项是可能的。例如：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- '+'。和 '*' 类似，所有当前命令行参数被聚集到一个列表中。另外，当前没有至少一个命令行参数时会产生一个错误信息。例如：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
```

(下页继续)

(续上页)

```
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: too few arguments
```

- argarse.REMAINDER。所有剩余的命令行参数被聚集到一个列表中。这通常在从一个命令行功能传递参数到另一个命令行功能中时有用:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo')
>>> parser.add_argument('command')
>>> parser.add_argument('args', nargs=argparse.REMAINDER)
>>> print parser.parse_args('--foo B cmd --arg1 XX ZZ'.split())
Namespace(args=['--arg1', 'XX', 'ZZ'], command='cmd', foo='B')
```

如果不提供 nargs 命名参数，则消耗参数的数目将被*action* 决定。通常这意味着单一项目（非列表）消耗单一命令行参数。

const

add_argument() 的 “const“ 参数用于保存不从命令行中读取但被各种*ArgumentParser* 动作需求的常数值。最常用的两例为:

- 当*add_argument()* 通过 action='store_const' 或 action='append_const 调用时。这些动作将 const 值添加到*parse_args()* 返回的对象的属性中。在*action* 的描述中查看案例。
- 当*add_argument()* 通过选项（例如 -f 或 --foo）调用并且 nargs='?' 时。这会创建一个可以跟随零个或一个命令行参数的选项。当解析命令行时，如果选项后没有参数，则将用 const 代替。在*nargs* 描述中查看案例。

对 'store_const' 和 'append_const' 动作，const 命名参数必须给出。对其他动作，默认为 None。

默认值

所有选项和一些位置参数可能在命令行中被忽略。*add_argument()* 的命名参数 default，默认值为 None，指定了在命令行参数未出现时应当使用的值。对于选项，default 值在选项未在命令行中出现时使用:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

如果 default 值是一个字符串，解析器解析此值就像一个命令行参数。特别是，在将属性设置在*Namespace* 的返回值之前，解析器应用任何提供的*type* 转换参数。否则解析器使用原值:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

对于*nargs* 等于 ? 或 * 的位置参数，default 值在没有命令行参数出现时使用。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

Providing default=argparse.SUPPRESS causes no attribute to be added if the command-line argument was not present.:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

type –类型

默认情况下，*ArgumentParser* 对象将命令行参数当作简单字符串读入。然而，命令行字符串经常需要被当作其它的类型，比如*float* 或者*int*。*add_argument()* 的 type 关键词参数允许任何的类型检查和类型转换。一般的内建类型和函数可以直接被 type 参数使用。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.add_argument('bar', type=file)
>>> parser.parse_args('2 temp.txt'.split())
Namespace(bar=<open file 'temp.txt', mode 'r' at 0x...>, foo=2)
```

当 type 参数被应用到默认参数时，请参考*default* 参数的部分。

To ease the use of various types of files, the argparse module provides the factory FileType which takes the mode= and bufsize= arguments of the file object. For example, FileType('w') can be used to create a writable file:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar', type=argparse.FileType('w'))
>>> parser.parse_args(['out.txt'])
Namespace(bar=<open file 'out.txt', mode 'w' at 0x...>)
```

type= 可接受任意可调用对象，该对象应传入单个字符串参数并返回转换后的值:

```
>>> def perfect_square(string):
...     value = int(string)
...     sqrt = math.sqrt(value)
...     if sqrt != int(sqrt):
...         msg = "%r is not a perfect square" % string
...         raise argparse.ArgumentTypeError(msg)
...     return value
...
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=perfect_square)
>>> parser.parse_args(['9'])
Namespace(foo=9)
>>> parser.parse_args(['7'])
```

(下页继续)

(续上页)

```
usage: PROG [-h] foo
PROG: error: argument foo: '7' is not a perfect square
```

choices 关键词参数可能会使类型检查者更方便的检查一个范围的值。

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=int, choices=xrange(5, 10))
>>> parser.parse_args(['7'])
Namespace(foo=7)
>>> parser.parse_args(['11'])
usage: PROG [-h] {5,6,7,8,9}
PROG: error: argument foo: invalid choice: 11 (choose from 5, 6, 7, 8, 9)
```

详情请查阅*choices* 段落。

choices

某些命令行参数应当从一组受限值中选择。这可通过将一个容器对象作为 *choices* 关键字参数传给*add_argument()* 来处理。当执行命令行解析时，参数值将被检查，如果参数不是可接受的值之一就将显示错误消息:

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

请注意 *choices* 容器包含的内容会在执行任意*type* 转换之后被检查，因此 *choices* 容器中对象的类型应当与指定的*type* 相匹配:

```
>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
>>> parser.parse_args(['4'])
usage: doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)
```

Any object that supports the `in` operator can be passed as the *choices* value, so *dict* objects, *set* objects, custom containers, etc. are all supported.

required

通常，*argparse* 模块会认为 `-f` 和 `--bar` 等旗标是指明 可选的参数，它们总是可以在命令行中被忽略。要让一个选项成为 必需的，则可以将 `True` 作为 `required=` 关键字参数传给*add_argument()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
```

(下页继续)

(续上页)

```
>>> parser.parse_args([])
usage: argparse.py [-h] [--foo FOO]
argparse.py: error: option --foo is required
```

如这个例子所示，如果一个选项被标记为 `required`，则当该选项未在命令行中出现时，*parse_args()* 将会报告一个错误。

注解： 必需的选项通常被认为是不适宜的，因为用户会预期 *options* 都是 可选的，因此在可能的情况下应当避免使用它们。

---

help

`help` 值是一个包含参数简短描述的字符串。当用户请求帮助时（一般是通过在命令行中使用 `-h` 或 `--help` 的方式），这些 `help` 描述将随每个参数一同显示:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true',
...                     help='foo the bars before frobbling')
>>> parser.add_argument('bar', nargs='+',
...                     help='one of the bars to be frobbled')
>>> parser.parse_args(['-h'])
usage: frobble [-h] [--foo] bar [bar ...]

positional arguments:
 bar     one of the bars to be frobbled

optional arguments:
 -h, --help  show this help message and exit
 --foo   foo the bars before frobbling
```

`help` 字符串可包括各种格式描述符以避免重复使用程序名称或参数*default* 等文本。有效的描述符包括程序名称 `%(prog)s` 和传给*add_argument()* 的大部分关键字参数，例如 `%(default)s`, `%(type)s` 等等:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                     help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
 bar     the bar to frobble (default: 42)

optional arguments:
 -h, --help  show this help message and exit
```

argparse 支持静默特定选项的帮助，具体做法是将 `help` 的值设为 `argparse.SUPPRESS`:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

optional arguments:
  -h, --help  show this help message and exit
```

metavar

当*ArgumentParser*生成帮助消息时，它需要用某种方式来引用每个预期的参数。默认情况下，ArgumentParser 对象使用*dest*值作为每个对象的“name”。默认情况下，对于位置参数动作，*dest*值将被直接使用，而对于可选参数动作，*dest*值将被转为大写形式。因此，一个位置参数 `dest='bar'` 的引用形式将为 `bar`。一个带有单独命令行参数的可选参数 `--foo` 的引用形式将为 `FOO`。示例如下:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo FOO] bar

positional arguments:
 bar

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO
```

可以使用 `metavar` 来指定一个替代名称:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo YYY] XXX

positional arguments:
 XXX

optional arguments:
 -h, --help  show this help message and exit
 --foo YYY
```

请注意 `metavar` 仅改变 *显示的名称* - *parse_args()* 对象的属性名称仍然会由*dest*值确定。

不同的 `nargs` 值可能导致 metavar 被多次使用。提供一个元组给 `metavar` 即为每个参数指定不同的显示信息:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

optional arguments:
 -h, --help     show this help message and exit
 -x X X
 --foo bar baz
```

dest

大多数*ArgumentParser* 动作会添加一些值作为*parse_args()* 所返回对象的一个属性。该属性的名称由*add_argument()* 的 dest 关键字参数确定。对于位置参数动作，dest 通常会作为*add_argument()* 的第一个参数提供:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

对于可选参数动作，dest 的值通常取自选项字符串。*ArgumentParser* 会通过接受第一个长选项字符串并去掉开头的 -- 字符串来生成 dest 的值。如果没有提供长选项字符串，则 dest 将通过接受第一个短选项字符串并去掉开头的 - 字符来获得。任何内部的 - 字符都将被转换为 _ 字符以确保字符串是有效的属性名称。下面的例子显示了这种行为:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

dest 允许提供自定义属性名称:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

Action 类

Action 类实现了 Action API，它是一个返回可调用对象的可调用对象，返回的可调用对象可处理来自命令行的参数。任何遵循此 API 的对象均可作为 action 形参传给 add_argument()。

class argparse.**Action**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Action 对象会被 ArgumentParser 用来表示解析从命令行中的一个或多个字符串中解析出单个参数所必须的信息。Action 类必须接受两个位置参数以及传给*ArgumentParser.add_argument()* 的任何关键字参数，除了 action 本身。

Action 的实例（或作为 or return value of any callable to the action 形参的任何可调用对象的返回值）应当定义“dest”，“option_strings”，“default”，“type”，“required”，“help”等属性。确保这些属性被定义的最容易方式是调用 Action.__init__。

Action 的实例应当为可调用对象，因此所有子类都必须重载 __call__ 方法，该方法应当接受四个形参:

- parser - 包含此动作的 ArgumentParser 对象。
- namespace - 将由*parse_args()* 返回的*Namespace* 对象。大多数动作会使用*setattr()* 为此对象添加属性。
- values - 已关联的命令行参数，并提供相应的类型转换。类型转换由*add_argument()* 的*type* 关键字参数来指定。
- option_string - 被用来发起调用此动作的选项字符串。option_string 参数是可选的，且此参数在动作关联到位置参数时将被略去。

__call__ 方法可以执行任意动作，但通常将基于 dest 和 values 来设置 namespace 的属性。

15.4.4 parse_args() 方法

ArgumentParser.**parse_args**(*args=None*, *namespace=None*)
将参数字符串转换为对象并将其设为命名空间的属性。返回带有成员的命名空间。

Previous calls to *add_argument()* determine exactly what objects are created and how they are assigned. See the documentation for *add_argument()* for details.

- *args* - List of strings to parse. The default is taken from *sys.argv*.
- *namespace* - An object to take the attributes. The default is a new empty *Namespace* object.

Option value syntax

The *parse_args()* method supports several ways of specifying the value of an option (if it takes one). In the simplest case, the option and its value are passed as two separate arguments:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

For long options (options with names longer than a single character), the option and value can also be passed as a single command-line argument, using = to separate them:

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

For short options (options only one character long), the option and its value can be concatenated:

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

Several short options can be joined together, using only a single - prefix, as long as only the last option (or none of them) requires a value:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

无效的参数

While parsing the command line, *parse_args()* checks for a variety of errors, including ambiguous options, invalid types, invalid options, wrong number of positional arguments, etc. When it encounters such an error, it exits and prints the error along with a usage message:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

包含 - 的参数

The *parse_args()* method attempts to give errors whenever the user has clearly made a mistake, but some situations are inherently ambiguous. For example, the command-line argument `-1` could either be an attempt to specify an option or an attempt to provide a positional argument. The *parse_args()* method is cautious here: positional arguments may only begin with `-` if they look like negative numbers and there are no options in the parser that look like negative numbers:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
```

(下页继续)

(续上页)

```
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

If you have positional arguments that must begin with - and don't look like negative numbers, you can insert the pseudo-argument `'--'` which tells *parse_args()* that everything after that is a positional argument:

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

参数缩写（前缀匹配）

The *parse_args()* method allows long options to be abbreviated to a prefix, if the abbreviation is unambiguous (the prefix matches a unique option):

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

An error is produced for arguments that could produce more than one options.

Beyond `sys.argv`

Sometimes it may be useful to have an ArgumentParser parse arguments other than those of *sys.argv*. This can be accomplished by passing a list of strings to *parse_args()*. This is useful for testing at the interactive prompt:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=xrange(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...     default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

命名空间对象

class argparse.**Namespace**
Simple class used by default by *parse_args()* to create an object holding attributes and return it.

This class is deliberately simple, just an *object* subclass with a readable string representation. If you prefer to have dict-like view of the attributes, you can use the standard Python idiom, *vars()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

It may also be useful to have an *ArgumentParser* assign attributes to an already existing object, rather than a new *Namespace* object. This can be achieved by specifying the namespace= keyword argument:

```
>>> class C(object):
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

15.4.5 其它实用工具

子命令

ArgumentParser.**add_subparsers**([*title*][, *description*][, *prog*][, *parser_class*][, *action*][, *option_string*][, *dest*][, *help*][, *metavar*])
Many programs split up their functionality into a number of sub-commands, for example, the svn program can invoke sub-commands like svn checkout, svn update, and svn commit. Splitting up functionality this way can be a particularly good idea when a program performs several different functions which require different kinds of command-line arguments. *ArgumentParser* supports the creation of such sub-commands with the *add_subparsers()* method. The *add_subparsers()* method is normally called with no arguments and returns a special action object. This object has a single method, add_parser(), which takes a command name and any *ArgumentParser* constructor arguments, and returns an *ArgumentParser* object that can be modified as usual.

形参的描述

- title - title for the sub-parser group in help output; by default "subcommands" if description is provided, otherwise uses title for positional arguments
- description - description for the sub-parser group in help output, by default None
- prog - usage information that will be displayed with sub-command help, by default the name of the program and any positional arguments before the subparser argument
- parser_class - class which will be used to create sub-parser instances, by default the class of the current parser (e.g. ArgumentParser)
- *action* - the basic type of action to be taken when this argument is encountered at the command line

- *dest* - name of the attribute under which sub-command name will be stored; by default `None` and no value is stored
- *help* - help for sub-parser group in help output, by default `None`
- *metavar* - string presenting available sub-commands in help; by default it is `None` and presents sub-commands in form {cmd1, cmd2, ..}

一些使用示例:

```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='sub-command help')
>>>
>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices='XYZ', help='baz help')
>>>
>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

Note that the object returned by *parse_args()* will only contain attributes for the main parser and the subparser that was selected by the command line (and not any other subparsers). So in the example above, when the `a` command is specified, only the `foo` and `bar` attributes are present, and when the `b` command is specified, only the `foo` and `baz` attributes are present.

Similarly, when a help message is requested from a subparser, only the help for that particular parser will be printed. The help message will not include parent parser or sibling parser messages. (A help message for each subparser command, however, can be given by supplying the `help=` argument to `add_parser()` as above.)

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   sub-command help
    a     a help
    b     b help

optional arguments:
  -h, --help  show this help message and exit
  --foo   foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar     bar help

optional arguments:
  -h, --help  show this help message and exit
```

(下页继续)

(续上页)

```
>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

optional arguments:
  -h, --help     show this help message and exit
  --baz {X,Y,Z}  baz help
```

The *add_subparsers()* method also supports `title` and `description` keyword arguments. When either is present, the subparser's commands will appear in their own group in the help output. For example:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                    description='valid subcommands',
...                                    help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

optional arguments:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}   additional help
```

One particularly effective way of handling sub-commands is to combine the use of the *add_subparsers()* method with calls to *set_defaults()* so that each subparser knows which Python function it should execute. For example:

```
>>> # sub-command functions
>>> def foo(args):
...     print args.x * args.y
...
>>> def bar(args):
...     print '((%s))' % args.z
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>>
>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
```

(下页继续)

(续上页)

```
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

This way, you can let *parse_args()* do the job of calling the appropriate function after argument parsing is complete. Associating functions with actions like this is typically the easiest way to handle the different actions for each of your subparsers. However, if it is necessary to check the name of the subparser that was invoked, the `dest` keyword argument to the *add_subparsers()* call will work:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

FileType 对象

class argparse.**FileType**(*mode='r'*, *bufsize=None*)

The *FileType* factory creates objects that can be passed to the type argument of *ArgumentParser.add_argument()*. Arguments that have *FileType* objects as their type will open command-line arguments as files with the requested modes and buffer sizes:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--output', type=argparse.FileType('wb', 0))
>>> parser.parse_args(['--output', 'out'])
Namespace(output=<open file 'out', mode 'wb' at 0x...>)
```

FileType objects understand the pseudo-argument `'-'` and automatically convert this into `sys.stdin` for readable *FileType* objects and `sys.stdout` for writable *FileType* objects:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<open file '<stdin>', mode 'r' at 0x...>)
```

参数组

ArgumentParser.**add_argument_group**(*title=None*, *description=None*)

By default, *ArgumentParser* groups command-line arguments into "positional arguments" and "optional arguments" when displaying help messages. When there is a better conceptual grouping of arguments than this default one, appropriate groups can be created using the *add_argument_group()* method:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
```

(下页继续)

(续上页)

```
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
  bar    bar help
  --foo FOO  foo help
```

The *add_argument_group()* method returns an argument group object which has an *add_argument()* method just like a regular *ArgumentParser*. When an argument is added to the group, the parser treats it just like a normal argument, but displays the argument in a separate group for help messages. The *add_argument_group()* method accepts *title* and *description* arguments which can be used to customize this display:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo    foo help

group2:
  group2 description

  --bar BAR  bar help
```

Note that any arguments not in your user-defined groups will end up back in the usual "positional arguments" and "optional arguments" sections.

Mutual exclusion

ArgumentParser.**add_mutually_exclusive_group**(*required=False*)

创建一个互斥组。*argparse* 将会确保互斥组中只有一个参数在命令行中可用:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

add_mutually_exclusive_group() 方法也接受一个 *required* 参数，表示在互斥组中至少有一个参数是需要的:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

注意，目前互斥参数组不支持*add_argument_group()* 的 *title* 和 *description* 参数。

Parser defaults

ArgumentParser.**set_defaults**(***kwargs*)

Most of the time, the attributes of the object returned by *parse_args()* will be fully determined by inspecting the command-line arguments and the argument actions. *set_defaults()* allows some additional attributes that are determined without any inspection of the command line to be added:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

Note that parser-level defaults always override argument-level defaults:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

Parser-level defaults can be particularly useful when working with multiple parsers. See the *add_subparsers()* method for an example of this type.

ArgumentParser.**get_default**(*dest*)

Get the default value for a namespace attribute, as set by either *add_argument()* or by *set_defaults()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

打印帮助

In most typical applications, *parse_args()* will take care of formatting and printing any usage or error messages. However, several formatting methods are available:

ArgumentParser.**print_usage**(*file=None*)

Print a brief description of how the *ArgumentParser* should be invoked on the command line. If *file* is None, *sys.stdout* is assumed.

ArgumentParser.**print_help**(*file=None*)

Print a help message, including the program usage and information about the arguments registered with the *ArgumentParser*. If *file* is None, *sys.stdout* is assumed.

There are also variants of these methods that simply return a string instead of printing it:

ArgumentParser.**format_usage**()
Return a string containing a brief description of how the *ArgumentParser* should be invoked on the command line.

ArgumentParser.**format_help**()
Return a string containing a help message, including the program usage and information about the arguments registered with the *ArgumentParser*.

Partial parsing

ArgumentParser.**parse_known_args**(*args=None*, *namespace=None*)

Sometimes a script may only parse a few of the command-line arguments, passing the remaining arguments on to another script or program. In these cases, the *parse_known_args()* method can be useful. It works much like *parse_args()* except that it does not produce an error when extra arguments are present. Instead, it returns a two item tuple containing the populated namespace and the list of remaining argument strings.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

警告: *Prefix matching* rules apply to `parse_known_args()`. The parser may consume an option even if it's just a prefix of one of its known options, instead of leaving it in the remaining arguments list.

自定义文件解析

ArgumentParser.**convert_arg_line_to_args**(*arg_line*)
Arguments that are read from a file (see the *fromfile_prefix_chars* keyword argument to the *ArgumentParser* constructor) are read one argument per line. *convert_arg_line_to_args()* can be overridden for fancier reading.

This method takes a single argument *arg_line* which is a string read from the argument file. It returns a list of arguments parsed from this string. The method is called once per line read from the argument file, in order.

A useful override of this method is one that treats each space-separated word as an argument:

```
def convert_arg_line_to_args(self, arg_line):
    return arg_line.split()
```

退出方法

ArgumentParser.**exit**(*status=0*, *message=None*)
This method terminates the program, exiting with the specified *status* and, if given, it prints a *message* before that.

ArgumentParser.**error**(*message*)
This method prints a usage message including the *message* to the standard error and terminates the program with a status code of 2.

15.4.6 升级 optparse 代码

Originally, the *argparse* module had attempted to maintain compatibility with *optparse*. However, *optparse* was difficult to extend transparently, particularly with the changes required to support the new `nargs=` specifiers and better usage messages. When most everything in *optparse* had either been copy-pasted over or monkey-patched, it no longer seemed practical to try to maintain the backwards compatibility.

The *argparse* module improves on the standard library *optparse* module in a number of ways including:

- 处理位置参数。
- 支持子命令。
- Allowing alternative option prefixes like + and /.
- Handling zero-or-more and one-or-more style arguments.
- Producing more informative usage messages.
- Providing a much simpler interface for custom `type` and `action`.

A partial upgrade path from *optparse* to *argparse*:

- Replace all *optparse.OptionParser.add_option()* calls with *ArgumentParser.add_argument()* calls.
- Replace `(options, args) = parser.parse_args()` with `args = parser.parse_args()` and add additional *ArgumentParser.add_argument()* calls for the positional arguments. Keep in mind that what was previously called `options`, now in the *argparse* context is called `args`.
- Replace *optparse.OptionParser.disable_interspersed_args()* by setting `nargs` of a positional argument to *argparse.REMAINDER*, or use *parse_known_args()* to collect unparsed argument strings in a separate list.
- Replace callback actions and the `callback_*` keyword arguments with `type` or `action` arguments.
- Replace string names for `type` keyword arguments with the corresponding type objects (e.g. int, float, complex, etc).
- Replace `optparse.Values` with *Namespace* and `optparse.OptionError` and `optparse.OptionValueError` with `ArgumentError`.
- Replace strings with implicit arguments such as `%default` or `%prog` with the standard Python syntax to use dictionaries to format strings, that is, `%(default)s` and `%(prog)s`.
- Replace the OptionParser constructor `version` argument with a call to `parser.add_argument('--version', action='version', version='<the version>')`.

15.5 optparse —解析器的命令行选项

2.3 新版功能.

2.7 版后已移除: The *optparse* module is deprecated and will not be developed further; development will continue with the *argparse* module.

源代码： Lib/optparse.py

---

optparse is a more convenient, flexible, and powerful library for parsing command-line options than the old *getopt* module. *optparse* uses a more declarative style of command-line parsing: you create an instance of *OptionParser*,

populate it with options, and parse the command line. *optparse* allows users to specify options in the conventional GNU/POSIX syntax, and additionally generates usage and help messages for you.

Here's an example of using *optparse* in a simple script:

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

With these few lines of code, users of your script can now do the "usual thing" on the command-line, for example:

```
<yourscript> --file=outfile -q
```

As it parses the command line, *optparse* sets attributes of the `options` object returned by `parse_args()` based on user-supplied command-line values. When `parse_args()` returns from parsing this command line, `options.filename` will be `"outfile"` and `options.verbose` will be `False`. *optparse* supports both long and short options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

Additionally, users can run one of

```
<yourscript> -h
<yourscript> --help
```

and *optparse* will print out a brief summary of your script's options:

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

where the value of *yourscript* is determined at runtime (normally from `sys.argv[0]`).

15.5.1 背景

optparse was explicitly designed to encourage the creation of programs with straightforward, conventional command-line interfaces. To that end, it supports only the most common command-line syntax and semantics conventionally used under Unix. If you are unfamiliar with these conventions, read this section to acquaint yourself with them.

术语

argument –参数 a string entered on the command-line, and passed by the shell to `execl()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]` (`sys.argv[0]` is the name of the program being executed). Unix shells also use the term "word".

It is occasionally desirable to substitute an argument list other than `sys.argv[1:]`, so you should read "argument" as "an element of `sys.argv[1:]`, or of some other list provided as a substitute for `sys.argv[1:]`".

选项 an argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options; the traditional Unix syntax is a hyphen ("-") followed by a single letter, e.g. `-x` or `-F`. Also, traditional Unix syntax allows multiple options to be merged into a single argument, e.g. `-x -F` is equivalent to `-xF`. The GNU project introduced `--` followed by a series of hyphen-separated words, e.g. `--file` or `--dry-run`. These are the only two option syntaxes provided by *optparse*.

Some other option syntaxes that the world has seen include:

- a hyphen followed by a few letters, e.g. `-pf` (this is *not* the same as multiple options merged into a single argument)
- a hyphen followed by a whole word, e.g. `-file` (this is technically equivalent to the previous syntax, but they aren't usually seen in the same program)
- a plus sign followed by a single letter, or a few letters, or a word, e.g. `+f`, `+rgb`
- a slash followed by a letter, or a few letters, or a word, e.g. `/f`, `/file`

These option syntaxes are not supported by *optparse*, and they never will be. This is deliberate: the first three are non-standard on any environment, and the last only makes sense if you're exclusively targeting VMS, MS-DOS, and/or Windows.

可选参数: an argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With *optparse*, option arguments may either be in a separate argument from their option:

```
-f foo
--file foo
```

or included in the same argument:

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn't. Lots of people want an "optional option arguments" feature, meaning that some options will take an argument if they see it, and won't if they don't. This is somewhat controversial, because it makes parsing ambiguous: if `-a` takes an optional argument and `-b` is another option entirely, how do we interpret `-ab`? Because of this ambiguity, *optparse* does not support this feature.

positional argument –位置参数 something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

必选选项 an option that must be supplied on the command-line; note that the phrase "required option" is self-contradictory in English. *optparse* doesn't prevent you from implementing required options, but doesn't give you much help at it either.

For example, consider this hypothetical command-line:

```
prog -v --report report.txt foo bar
```

`-v` and `--report` are both options. Assuming that `--report` takes one argument, `report.txt` is an option argument. `foo` and `bar` are positional arguments.

What are options for?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn't clear, options are usually *optional*. A program should be able to run just fine with no options whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense? The main exceptions are `find`, `tar`, and `dd`—all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces.)

Lots of people want their programs to have "required options". Think about it. If it's required, then it's *not optional*! If there is a piece of information that your program absolutely requires in order to run successfully, that's what positional arguments are for.

As an example of good command-line interface design, consider the humble `cp` utility, for copying files. It doesn't make much sense to try to copy files without supplying a destination and at least one source. Hence, `cp` fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. Most `cp` implementations provide a bunch of options to tweak exactly how the files are copied: you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of `cp`, which is to copy either one file to another, or several files to another directory.

位置位置

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in order to run successfully, it doesn't much matter *how* you get that information from the user—most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI: if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply—use sensible defaults whenever possible. Of course, you also want to make your programs reasonably flexible. That's what options are for. Again, it doesn't matter if they are entries in a config file, widgets in the "Preferences" dialog of a GUI, or command-line options—the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course; too many options can overwhelm users and make your code much harder to maintain.

15.5.2 教程

While *optparse* is quite flexible and powerful, it's also straightforward to use in most cases. This section covers the code patterns that are common to any *optparse*-based program.

First, you need to import the OptionParser class; then, early in the main program, create an OptionParser instance:

```
from optparse import OptionParser
...
parser = OptionParser()
```

Then you can start defining options. The basic syntax is:

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

Each option has one or more option strings, such as `-f` or `--file`, and several option attributes that tell *optparse* what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g.:

```
parser.add_option("-f", "--file", ...)
```

You're free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to *OptionParser.add_option()* are effectively labels for the option defined by that call. For brevity, we will frequently refer to *encountering an option* on the command line; in reality, *optparse* encounters *option strings* and looks up options from them.

Once all of your options are defined, instruct *optparse* to parse your program's command line:

```
(options, args) = parser.parse_args()
```

(If you like, you can pass a custom argument list to `parse_args()`, but that's rarely necessary: by default it uses `sys.argv[1:]`.)

`parse_args()` 返回两个值:

- `options`, an object containing values for all of your options—e.g. if `--file` takes a single string argument, then `options.file` will be the filename supplied by the user, or `None` if the user did not supply that option
- `args`, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes: *action*, *type*, *dest* (destination), and *help*. Of these, *action* is the most fundamental.

Understanding option actions

Actions tell *optparse* what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into *optparse*; adding new actions is an advanced topic covered in section *Extending optparse*. Most actions tell *optparse* to store a value in some variable—for example, take a string from the command line and store it in an attribute of `options`.

If you don't specify an option action, *optparse* defaults to `store`.

The store action

The most common option action is `store`, which tells *optparse* to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

例如

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

Now let's make up a fake command line and ask *optparse* to parse it:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When *optparse* sees the option string `-f`, it consumes the next argument, `foo.txt`, and stores it in `options.filename`. So, after this call to `parse_args()`, `options.filename` is `"foo.txt"`.

Some other option types supported by *optparse* are `int` and `float`. Here's an option that expects an integer argument:

```
parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly acceptable. Also, there's no explicit action, since the default is `store`.

Let's parse another fake command-line. This time, we'll jam the option argument right up against the option: since `-n42` (one argument) is equivalent to `-n 42` (two arguments), the code

```
(options, args) = parser.parse_args(["-n42"])
print options.num
```

will print `42`.

If you don't specify a type, *optparse* assumes `string`. Combined with the fact that the default action is `store`, that means our first example can be a lot shorter:

```
parser.add_option("-f", "--file", dest="filename")
```

If you don't supply a destination, *optparse* figures out a sensible default from the option strings: if the first long option string is `--foo-bar`, then the default destination is `foo_bar`. If there are no long option strings, *optparse* looks at the first short option string: the default destination for `-f` is `f`.

optparse also includes built-in `long` and `complex` types. Adding types is covered in section *Extending optparse*.

Handling boolean (flag) options

Flag options—set a variable to true or false when a particular option is seen —are quite common. *optparse* supports them with two separate actions, `store_true` and `store_false`. For example, you might have a `verbose` flag that is turned on with `-v` and off with `-q`:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values—see below.)

When *optparse* encounters `-v` on the command line, it sets `options.verbose` to `True`; when it encounters `-q`, `options.verbose` is set to `False`.

Other actions

Some other actions supported by *optparse* are:

"store_const" store a constant value

"append" append this option's argument to a list

"count" increment a counter by one

"callback" 调用指定函数

These are covered in section *参考指南*, Reference Guide and section *Option Callbacks*.

默认值

All of the above examples involve setting some variable (the "destination") when certain command-line options are seen. What happens if those options are never seen? Since we didn't supply any defaults, they are all set to `None`. This is usually fine, but sometimes you want more control. *optparse* lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want *optparse* to set `verbose` to `True` unless `-q` is seen, then we can do this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Since default values apply to the *destination* rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

考虑一下:

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Again, the default value for `verbose` will be `True`: the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the `set_defaults()` method of OptionParser, which you can call at any time before calling `parse_args()`:

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

As before, the last value specified for a given option destination is the one that counts. For clarity, try to use one method or the other of setting default values, not both.

Generating help

optparse's ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a *help* value for each option, and optionally a short usage message for your whole program. Here's an OptionParser populated with user-friendly (documented) options:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                       "or expert [default: %default]")
```

If *optparse* encounters either `-h` or `--help` on the command-line, or if you just call `parser.print_help()`, it prints the following to standard output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

(If the help output is triggered by a help option, *optparse* exits after printing the help text.)

There's a lot going on here to help *optparse* generate the best possible help message:

- the script defines its own usage message:

  ```
  usage = "usage: %prog [options] arg1 arg2"
  ```

  optparse expands `%prog` in the usage string to the name of the current program, i.e. `os.path.basename(sys.argv[0])`. The expanded string is then printed before the detailed option help.

  If you don't supply a usage string, *optparse* uses a bland but sensible default: `"Usage: %prog [options]"`, which is fine if your script doesn't take any positional arguments.

- every option defines a help string, and doesn't worry about line-wrapping—*optparse* takes care of wrapping lines and making the help output look good.

- options that take a value indicate this fact in their automatically-generated help message, e.g. for the "mode" option:

  ```
  -m MODE, --mode=MODE
  ```

  Here, "MODE" is called the meta-variable: it stands for the argument that the user is expected to supply to `-m`/`--mode`. By default, *optparse* converts the destination variable name to uppercase and uses that for the

meta-variable. Sometimes, that's not what you want—for example, the `--filename` option explicitly sets `metavar="FILE"`, resulting in this automatically-generated option description:

```
-f FILE, --filename=FILE
```

This is important for more than just saving space, though: the manually written help text uses the meta-variable `FILE` to clue the user in that there's a connection between the semi-formal syntax `-f FILE` and the informal semantic description "write output to FILE". This is a simple but effective way to make your help text a lot clearer and more useful for end users.

2.4 新版功能: Options that have a default value can include `%default` in the help string—*optparse* will replace it with *str()* of the option's default value. If an option has no default value (or the default value is `None`), `%default` expands to `none`.

Grouping Options

When dealing with many options, it is convenient to group these options for better help output. An *OptionParser* can contain several option groups, each of which can contain several options.

An option group is obtained using the class *OptionGroup*:

class optparse.**OptionGroup**(*parser*, *title*, *description=None*)

where

- parser is the *OptionParser* instance the group will be insterted in to
- title is the group title
- description, optional, is a long description of the group

OptionGroup inherits from `OptionContainer` (like *OptionParser*) and so the `add_option()` method can be used to add an option to the group.

Once all the options are declared, using the *OptionParser* method `add_option_group()` the group is added to the previously defined parser.

Continuing with the parser defined in the previous section, adding an *OptionGroup* to a parser is easy:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

This would result in the following help output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
```

(下页继续)

(续上页)

```
    of them bite.

    -g                  Group option.
```

A bit more complete example might involve using more than one group: still extending the previous example:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                 help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                 help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```

that results in the following output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or expert
                        [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.

  Debug Options:
    -d, --debug         Print debug information
    -s, --sql           Print all SQL statements executed
    -e                  Print every action done
```

Another interesting method, in particular when working programmatically with option groups is:

`OptionParser.`**`get_option_group`**`(`*opt_str*`)`

Return the *OptionGroup* to which the short or long option string *opt_str* (e.g. `'-o'` or `'--option'`) belongs. If there's no such *OptionGroup*, return `None`.

Printing a version string

Similar to the brief usage string, *optparse* can also print a version string for your program. You have to supply the string as the `version` argument to OptionParser:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

`%prog` is expanded just like it is in `usage`. Apart from that, `version` can contain anything you like. When you supply it, *optparse* automatically adds a `--version` option to your parser. If it encounters this option on the command line, it expands your `version` string (by replacing `%prog`), prints it to stdout, and exits.

For example, if your script is called `/usr/bin/foo`:

```
$ /usr/bin/foo --version
foo 1.0
```

The following two methods can be used to print and get the `version` string:

`OptionParser.`**`print_version`**`(`*file=None*`)`
: Print the version message for the current program (`self.version`) to *file* (default stdout). As with *print_usage()*, any occurrence of `%prog` in `self.version` is replaced with the name of the current program. Does nothing if `self.version` is empty or undefined.

`OptionParser.`**`get_version`**`()`
: Same as *print_version()* but returns the version string instead of printing it.

How `optparse` handles errors

There are two broad classes of errors that *optparse* has to worry about: programmer errors and user errors. Programmer errors are usually erroneous calls to *OptionParser.add_option()*, e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way: raise an exception (either `optparse.OptionError` or *TypeError*) and let the program crash.

Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. *optparse* can automatically detect some user errors, such as bad option arguments (passing `-n 4x` where `-n` takes an integer argument), missing arguments (`-n` at the end of the command line, where `-n` takes an argument of any type). Also, you can call `OptionParser.error()` to signal an application-defined error condition:

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, *optparse* handles the error the same way: it prints the program's usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes `4x` to an option that takes an integer:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

Or, where the user fails to pass a value at all:

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

optparse-generated error messages take care always to mention the option involved in the error; be sure to do the same when calling `OptionParser.error()` from your application code.

If *optparse*'s default error-handling behaviour does not suit your needs, you'll need to subclass OptionParser and override its `exit()` and/or `error()` methods.

Putting it all together

Here's what *optparse*-based scripts usually look like:

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print "reading %s..." % options.filename
    ...

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

15.5.3 参考指南

创建解析器

The first step in using *optparse* is to create an OptionParser instance.

class optparse.**OptionParser**(...)

The OptionParser constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

usage (默认: **"%prog [options]"**) The usage summary to print when your program is run incorrectly or with a help option. When *optparse* prints the usage string, it expands `%prog` to `os.path.basename(sys.argv[0])` (or to `prog` if you passed that keyword argument). To suppress a usage message, pass the special value `optparse.SUPPRESS_USAGE`.

option_list (默认: **[]**) A list of Option objects to populate the parser with. The options in `option_list` are added after any options in `standard_option_list` (a class attribute that may be set by OptionParser

subclasses), but before any version or help options. Deprecated; use *add_option()* after creating the parser instead.

option_class (默认: optparse.Option) Class to use when adding options to the parser in *add_option()*.

version (默认: None) A version string to print when the user supplies a version option. If you supply a true value for `version`, *optparse* automatically adds a version option with the single option string `--version`. The substring `%prog` is expanded the same as for `usage`.

conflict_handler (默认: "error") Specifies what to do when options with conflicting option strings are added to the parser; see section *Conflicts between options*.

description (默认: None) A paragraph of text giving a brief overview of your program. *optparse* reformats this paragraph to fit the current terminal width and prints it when the user requests help (after `usage`, but before the list of options).

formatter (default: a new IndentedHelpFormatter) An instance of optparse.HelpFormatter that will be used for printing help text. *optparse* provides two concrete classes for this purpose: IndentedHelpFormatter and TitledHelpFormatter.

add_help_option (默认: True) If true, *optparse* will add a help option (with option strings `-h` and `--help`) to the parser.

prog The string to use when expanding `%prog` in `usage` and `version` instead of `os.path.basename(sys.argv[0])`.

epilog (默认: None) A paragraph of help text to print after the option help.

填充解析器

There are several ways to populate the parser with options. The preferred way is by using *OptionParser.add_option()*, as shown in section 教程. `add_option()` can be called in one of two ways:

- pass it an Option instance (as returned by `make_option()`)
- pass it any combination of positional and keyword arguments that are acceptable to `make_option()` (i.e., to the Option constructor), and it will create the Option instance for you

The other alternative is to pass a list of pre-constructed Option instances to the OptionParser constructor, as in:

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
    ]
parser = OptionParser(option_list=option_list)
```

(`make_option()` is a factory function for creating Option instances; currently it is an alias for the Option constructor. A future version of *optparse* may split Option into several classes, and `make_option()` will pick the right class to instantiate. Do not instantiate Option directly.)

定义选项

Each Option instance represents a set of synonymous command-line option strings, e.g. `-f` and `--file`. You can specify any number of short or long option strings, but you must specify at least one overall option string.

The canonical way to create an `Option` instance is with the `add_option()` method of *`OptionParser`*.

`OptionParser.`**`add_option`**(*option*)
`OptionParser.`**`add_option`**(**opt_str*, *attr=value*, ...)

To define an option with only a short option string:

```
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string:

```
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is *`action`*, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, *`optparse`* raises an `OptionError` exception explaining your mistake.

An option's *action* determines what *`optparse`* does when it encounters this option on the command-line. The standard option actions hard-coded into *`optparse`* are:

`"store"` 存储此选项的参数（默认）

`"store_const"` store a constant value

`"store_true"` store a true value

`"store_false"` store a false value

`"append"` append this option's argument to a list

`"append_const"` 将常量值附加到列表

`"count"` increment a counter by one

`"callback"` 调用指定函数

`"help"` 打印用法消息，包括所有选项和文档

(If you don't supply an action, the default is `"store"`. For this action, you may also supply *`type`* and *`dest`* option attributes; see *Standard option actions*.)

As you can see, most actions involve storing or updating a value somewhere. *`optparse`* always creates a special object for this, conventionally called `options` (it happens to be an instance of `optparse.Values`). Option arguments (and various other values) are stored as attributes of this object, according to the *`dest`* (destination) option attribute.

For example, when you call

```
parser.parse_args()
```

one of the first things *`optparse`* does is create the `options` object:

```
options = Values()
```

If one of the options in this parser is defined with

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

and the command-line being parsed includes any of the following:

```
-ffoo
-f foo
--file=foo
--file foo
```

then *optparse*, on seeing this option, will do the equivalent of

```
options.filename = "foo"
```

The *type* and *dest* option attributes are almost as important as *action*, but *action* is the only one that makes sense for *all* options.

Option attributes

The following option attributes may be passed as keyword arguments to *OptionParser.add_option()*. If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, *optparse* raises OptionError.

Option.**action**
(默认: "store")

Determines *optparse*'s behaviour when this option is seen on the command line; the available options are documented *here*.

Option.**type**
(默认: "string")

The argument type expected by this option (e.g., "string" or "int"); the available option types are documented *here*.

Option.**dest**
(default: derived from option strings)

If the option's action implies writing or modifying a value somewhere, this tells *optparse* where to write it: *dest* names an attribute of the options object that *optparse* builds as it parses the command line.

Option.**default**
The value to use for this option's destination if the option is not seen on the command line. See also *OptionParser.set_defaults()*.

Option.**nargs**
(默认: 1)

How many arguments of type *type* should be consumed when this option is seen. If > 1, *optparse* will store a tuple of values to *dest*.

Option.**const**
For actions that store a constant value, the constant value to store.

Option.**choices**
For options of type "choice", the list of strings the user may choose from.

Option.**callback**
For options with action "callback", the callable to call when this option is seen. See section *Option Callbacks* for detail on the arguments passed to the callable.

Option.**callback_args**
Option.**callback_kwargs**
Additional positional and keyword arguments to pass to callback after the four standard callback arguments.

Option.**help**
Help text to print for this option when listing all available options after the user supplies a *help* option (such as `--help`). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value `optparse.SUPPRESS_HELP`.

Option.**metavar**
(default: derived from option strings)

Stand-in for the option argument(s) to use when printing help text. See section 教程 for an example.

Standard option actions

The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide *optparse*'s behaviour; a few have required attributes, which you must specify for any option using that action.

- `"store"` [relevant: *type*, *dest*, *nargs*, *choices*]

  The option must be followed by an argument, which is converted to a value according to *type* and stored in *dest*. If *nargs* > 1, multiple arguments will be consumed from the command line; all will be converted according to *type* and stored to *dest* as a tuple. See the *Standard option types* section.

  If *choices* is supplied (a list or tuple of strings), the type defaults to `"choice"`.

  If *type* is not supplied, it defaults to `"string"`.

  If *dest* is not supplied, *optparse* derives a destination from the first long option string (e.g., `--foo-bar` implies `foo_bar`). If there are no long option strings, *optparse* derives a destination from the first short option string (e.g., `-f` implies `f`).

  示例:

```
parser.add_option("-f")
parser.add_option("-p", type="float", nargs=3, dest="point")
```

  As it parses the command line

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

  optparse will set

```
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

- `"store_const"` [required: *const*; relevant: *dest*]

  The value *const* is stored in *dest*.

  示例:

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0, dest="verbose")
parser.add_option("-v", "--verbose",
                  action="store_const", const=1, dest="verbose")
parser.add_option("--noisy",
                  action="store_const", const=2, dest="verbose")
```

  If `--noisy` is seen, *optparse* will set

```
options.verbose = 2
```

- "store_true" [relevant: *dest*]

  A special case of "store_const" that stores a true value to *dest*.

- "store_false" [relevant: *dest*]

  Like "store_true", but stores a false value.

  示例:

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- "append" [relevant: *type*, *dest*, *nargs*, *choices*]

  The option must be followed by an argument, which is appended to the list in *dest*. If no default value for *dest* is supplied, an empty list is automatically created when *optparse* first encounters this option on the command-line. If *nargs* > 1, multiple arguments are consumed, and a tuple of length *nargs* is appended to *dest*.

  The defaults for *type* and *dest* are the same as for the "store" action.

  示例:

```
parser.add_option("-t", "--tracks", action="append", type="int")
```

  If -t3 is seen on the command-line, *optparse* does the equivalent of:

```
options.tracks = []
options.tracks.append(int("3"))
```

  If, a little later on, --tracks=4 is seen, it does:

```
options.tracks.append(int("4"))
```

  The append action calls the append method on the current value of the option. This means that any default value specified must have an append method. It also means that if the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values:

```
>>> parser.add_option("--files", action="append", default=['~/.mypkg/defaults'])
>>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
>>> opts.files
['~/.mypkg/defaults', 'overrides.mypkg']
```

- "append_const" [required: *const*; relevant: *dest*]

  Like "store_const", but the value *const* is appended to *dest*; as with "append", *dest* defaults to None, and an empty list is automatically created the first time the option is encountered.

- "count" [relevant: *dest*]

  Increment the integer stored at *dest*. If no default value is supplied, *dest* is set to zero before being incremented the first time.

  示例:

```
parser.add_option("-v", action="count", dest="verbosity")
```

  The first time -v is seen on the command line, *optparse* does the equivalent of:

```
options.verbosity = 0
options.verbosity += 1
```

Every subsequent occurrence of `-v` results in

```
options.verbosity += 1
```

- `"callback"` [required: *callback*; relevant: *type*, *nargs*, *callback_args*, *callback_kwargs*]

  Call the function specified by *callback*, which is called as

```
func(option, opt_str, value, parser, *args, **kwargs)
```

  See section *Option Callbacks* for more detail.

- `"help"`

  Prints a complete help message for all the options in the current option parser. The help message is constructed from the `usage` string passed to OptionParser's constructor and the *help* string passed to every option.

  If no *help* string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value `optparse.SUPPRESS_HELP`.

  optparse automatically adds a *help* option to all OptionParsers, so you do not normally need to create one.

  示例:

```
from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, but that can
# be suppressed using the add_help_option argument
parser = OptionParser(add_help_option=False)

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true", dest="verbose",
                  help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                  help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)
```

  If *optparse* sees either `-h` or `--help` on the command line, it will print something like the following help message to stdout (assuming `sys.argv[0]` is `"foo.py"`):

```
Usage: foo.py [options]

Options:
  -h, --help        Show this help message and exit
  -v                Be moderately verbose
  --file=FILENAME   Input file to read data from
```

  After printing the help message, *optparse* terminates your process with `sys.exit(0)`.

- `"version"`

  Prints the version number supplied to the OptionParser to stdout and exits. The version number is actually formatted and printed by the `print_version()` method of OptionParser. Generally only relevant if the `version` argument is supplied to the OptionParser constructor. As with *help* options, you will rarely create `version` options, since *optparse* automatically adds them when needed.

Standard option types

optparse has six built-in option types: "string", "int", "long", "choice", "float" and "complex". If you need to add new option types, see section *Extending optparse*.

Arguments to string options are not checked or converted in any way: the text on the command line is stored in the destination (or passed to the callback) as-is.

Integer arguments (type "int" or "long") are parsed as follows:

- if the number starts with 0x, it is parsed as a hexadecimal number
- if the number starts with 0, it is parsed as an octal number
- if the number starts with 0b, it is parsed as a binary number
- otherwise, the number is parsed as a decimal number

The conversion is done by calling either *int()* or *long()* with the appropriate base (2, 8, 10, or 16). If this fails, so will *optparse*, although with a more useful error message.

"float" and "complex" option arguments are converted directly with *float()* and *complex()*, with similar error-handling.

"choice" options are a subtype of "string" options. The *choices* option attribute (a sequence of strings) defines the set of allowed option arguments. optparse.check_choice() compares user-supplied option arguments against this master list and raises OptionValueError if an invalid string is given.

解析参数

The whole point of creating and populating an OptionParser is to call its parse_args() method:

```
(options, args) = parser.parse_args(args=None, values=None)
```

输入参数的位置

args the list of arguments to process (default: sys.argv[1:])

values an optparse.Values object to store option arguments in (default: a new instance of Values) –if you give an existing object, the option defaults will not be initialized on it

and the return values are

options the same object that was passed in as values, or the optparse.Values instance created by *optparse*

args the leftover positional arguments after all options have been processed

The most common usage is to supply neither keyword argument. If you supply values, it will be modified with repeated *setattr()* calls (roughly one for every option argument stored to an option destination) and returned by parse_args().

If parse_args() encounters any errors in the argument list, it calls the OptionParser's error() method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional Unix exit status for command-line errors).

Querying and manipulating your option parser

The default behavior of the option parser can be customized slightly, and you can also poke around your option parser and see what's there. OptionParser provides several methods to help you out:

`OptionParser.`**`disable_interspersed_args`**`()`
Set parsing to stop on the first non-option. For example, if `-a` and `-b` are both simple options that take no arguments, *optparse* normally accepts this syntax:

```
prog -a arg1 -b arg2
```

and treats it as equivalent to

```
prog -a -b arg1 arg2
```

To disable this feature, call *disable_interspersed_args()*. This restores traditional Unix syntax, where option parsing stops with the first non-option argument.

Use this if you have a command processor which runs another command which has options of its own and you want to make sure these options don't get confused. For example, each command might have a different set of options.

`OptionParser.`**`enable_interspersed_args`**`()`
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior.

`OptionParser.`**`get_option`**`(`*opt_str*`)`
Returns the Option instance with the option string *opt_str*, or `None` if no options have that option string.

`OptionParser.`**`has_option`**`(`*opt_str*`)`
Return true if the OptionParser has an option with option string *opt_str* (e.g., `-q` or `--verbose`).

`OptionParser.`**`remove_option`**`(`*opt_str*`)`
If the *OptionParser* has an option corresponding to *opt_str*, that option is removed. If that option provided any other option strings, all of those option strings become invalid. If *opt_str* does not occur in any option belonging to this *OptionParser*, raises *ValueError*.

Conflicts between options

If you're not careful, it's easy to define options with conflicting option strings:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(This is particularly true if you've defined your own OptionParser subclass with some standard options.)

Every time you add an option, *optparse* checks for conflicts with existing options. If it finds any, it invokes the current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor:

```
parser = OptionParser(..., conflict_handler=handler)
```

or with a separate call:

```
parser.set_conflict_handler(handler)
```

The available conflict handlers are:

`"error"` (默认) assume option conflicts are a programming error and raise `OptionConflictError`

"resolve" resolve option conflicts intelligently (see below)

As an example, let's define an *OptionParser* that resolves conflicts intelligently and add conflicting options to it:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

At this point, *optparse* detects that a previously-added option is already using the `-n` option string. Since `conflict_handler` is `"resolve"`, it resolves the situation by removing `-n` from the earlier option's list of option strings. Now `--dry-run` is the only way for the user to activate that option. If the user asks for help, the help message will reflect that:

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

It's possible to whittle away the option strings for a previously-added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, *optparse* removes that option completely, so it doesn't show up in help text or anywhere else. Carrying on with our existing OptionParser:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the original `-n`/`--dry-run` option is no longer accessible, so *optparse* removes it, leaving this help text:

```
Options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

清理

OptionParser instances have several cyclic references. This should not be a problem for Python's garbage collector, but you may wish to break the cyclic references explicitly by calling `destroy()` on your OptionParser once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your OptionParser.

Other methods

OptionParser supports several other public methods:

`OptionParser.`**`set_usage`**`(`*usage*`)`
Set the usage string according to the rules described above for the `usage` constructor keyword argument. Passing `None` sets the default usage string; use `optparse.SUPPRESS_USAGE` to suppress a usage message.

`OptionParser.`**`print_usage`**`(`*file=None*`)`
Print the usage message for the current program (`self.usage`) to *file* (default stdout). Any occurrence of the string `%prog` in `self.usage` is replaced with the name of the current program. Does nothing if `self.usage` is empty or not defined.

`OptionParser.`**`get_usage`**`()`
Same as *print_usage()* but returns the usage string instead of printing it.

OptionParser.**set_defaults**(*dest=value*, ...)

Set default values for several option destinations at once. Using *set_defaults()* is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several "mode" options all set the same destination, any one of them can set the default, and the last one wins:

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")    # overridden below
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")  # overrides above setting
```

To avoid this confusion, use *set_defaults()*:

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

15.5.4 Option Callbacks

When *optparse*'s built-in actions and types aren't quite enough for your needs, you have two choices: extend *optparse* or define a callback option. Extending *optparse* is more general, but overkill for a lot of simple cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option:

- define the option itself using the `"callback"` action
- write the callback; this is a function (or method) that takes at least four arguments, as described below

Defining a callback option

As always, the easiest way to define a callback option is by using the *OptionParser.add_option()* method. Apart from *action*, the only option attribute you must specify is `callback`, the function to call:

```
parser.add_option("-c", action="callback", callback=my_callback)
```

`callback` is a function (or other callable object), so you must have already defined `my_callback()` when you create this callback option. In this simple case, *optparse* doesn't even know if `-c` takes any arguments, which usually means that the option takes no arguments—the mere presence of `-c` on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing callbacks gets tricky; it's covered later in this section.

optparse always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via *callback_args* and *callback_kwargs*. Thus, the minimal callback function signature is:

```
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option:

type has its usual meaning: as with the `"store"` or `"append"` actions, it instructs *optparse* to consume one argument and convert it to *type*. Rather than storing the converted value(s) anywhere, though, *optparse* passes it to your callback function.

nargs also has its usual meaning: if it is supplied and > 1, *optparse* will consume *nargs* arguments, each of which must be convertible to *type*. It then passes a tuple of converted values to your callback.

callback_args a tuple of extra positional arguments to pass to the callback

callback_kwargs a dictionary of extra keyword arguments to pass to the callback

How callbacks are called

All callbacks are called as follows:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

where

option is the Option instance that's calling the callback

opt_str is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, `opt_str` will be the full, canonical option string—e.g. if the user puts `--foo` on the command-line as an abbreviation for `--foobar`, then `opt_str` will be `"--foobar"`.)

value is the argument to this option seen on the command-line. *optparse* will only expect an argument if *type* is set; the type of `value` will be the type implied by the option's type. If *type* for this option is `None` (no argument expected), then `value` will be `None`. If *nargs* > 1, `value` will be a tuple of values of the appropriate type.

parser is the OptionParser instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes:

parser.largs the current list of leftover arguments, ie. arguments that have been consumed but are neither options nor option arguments. Feel free to modify `parser.largs`, e.g. by adding more arguments to it. (This list will become `args`, the second return value of `parse_args()`.)

parser.rargs the current list of remaining arguments, ie. with `opt_str` and `value` (if applicable) removed, and only the arguments following them still there. Feel free to modify `parser.rargs`, e.g. by consuming more arguments.

parser.values the object where option values are by default stored (an instance of optparse.OptionValues). This lets callbacks use the same mechanism as the rest of *optparse* for storing option values; you don't need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

args is a tuple of arbitrary positional arguments supplied via the *callback_args* option attribute.

kwargs is a dictionary of arbitrary keyword arguments supplied via *callback_kwargs*.

Raising errors in a callback

The callback function should raise `OptionValueError` if there are any problems with the option or its argument(s). *optparse* catches this and terminates the program, printing the error message you supply to stderr. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what they did wrong.

Callback example 1: trivial callback

Here's an example of a callback option that takes no arguments, and simply records that the option was seen:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

Of course, you could do that with the `"store_true"` action.

Callback example 2: check option order

Here's a slightly more interesting example: record the fact that `-a` is seen, but blow up if it comes after `-b` in the command-line.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

Callback example 3: check option order (generalized)

If you want to re-use this callback for several similar options (set a flag, but blow up if `-b` has already been seen), it needs a bit of work: the error message and the flag that it sets must be generalized.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

Callback example 4: check arbitrary condition

Of course, you could put any condition in there—you're not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this:

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full"
                               % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(The definition of `is_moon_full()` is left as an exercise for the reader.)

Callback example 5: fixed arguments

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a "store" or "append" option: if you define *type*, then the option takes one argument that must be convertible to that type; if you further define *nargs*, then the option takes *nargs* arguments.

Here's an example that just emulates the standard "store" action:

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

Note that *optparse* takes care of consuming 3 arguments and converting them to integers for you; all you have to do is store them. (Or whatever; obviously you don't need a callback for this example.)

Callback example 6: variable arguments

Things get hairy when you want an option to take a variable number of arguments. For this case, you must write a callback, as *optparse* doesn't provide any built-in capabilities for it. And you have to deal with certain intricacies of conventional Unix command-line parsing that *optparse* normally handles for you. In particular, callbacks should implement the conventional rules for bare -- and - arguments:

- either -- or - can be option arguments
- bare -- (if not the argument to some option): halt command-line processing and discard the --
- bare - (if not the argument to some option): halt command-line processing but keep the - (append it to `parser.largs`)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you're willing to make for your application (which is why *optparse* doesn't support this sort of thing directly).

Nevertheless, here's a stab at a callback for an option with variable arguments:

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

    def floatable(str):
        try:
            float(str)
            return True
        except ValueError:
            return False

    for arg in parser.rargs:
        # stop on --foo like options
        if arg[:2] == "--" and len(arg) > 2:
            break
        # stop on -a, but not on -3 or -3.0
        if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
            break
        value.append(arg)
```

(下页继续)

(续上页)

```
    del parser.rargs[:len(value)]
    setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)
```

15.5.5 Extending `optparse`

Since the two major controlling factors in how *optparse* interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

Adding new types

To add new types, you need to define your own subclass of *optparse*'s `Option` class. This class has a couple of attributes that define *optparse*'s types: *TYPES* and *TYPE_CHECKER*.

`Option.`**`TYPES`**
: A tuple of type names; in your subclass, simply define a new tuple *TYPES* that builds on the standard one.

`Option.`**`TYPE_CHECKER`**
: A dictionary mapping type names to type-checking functions. A type-checking function has the following signature:

```
def check_mytype(option, opt, value)
```

  where `option` is an `Option` instance, `opt` is an option string (e.g., `-f`), and `value` is the string from the command line that must be checked and converted to your desired type. `check_mytype()` should return an object of the hypothetical type `mytype`. The value returned by a type-checking function will wind up in the OptionValues instance returned by `OptionParser.parse_args()`, or be passed to a callback as the `value` parameter.

  Your type-checking function should raise `OptionValueError` if it encounters any problems. `OptionValueError` takes a single string argument, which is passed as-is to *OptionParser*'s `error()` method, which in turn prepends the program name and the string `"error:"` and prints everything to stderr before terminating the process.

Here's a silly example that demonstrates adding a `"complex"` option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because *optparse* 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports:

```
from copy import copy
from optparse import Option, OptionValueError
```

You need to define your type-checker first, since it's referred to later (in the *TYPE_CHECKER* class attribute of your Option subclass):

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

Finally, the Option subclass:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(If we didn't make a *copy()* of *Option.TYPE_CHECKER*, we would end up modifying the *TYPE_CHECKER* attribute of *optparse*'s Option class. This being Python, nothing stops you from doing that except good manners and common sense.)

That's it! Now you can write a script that uses the new option type just like any other *optparse*-based script, except you have to instruct your OptionParser to use MyOption instead of Option:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

Alternately, you can build your own option list and pass it to OptionParser; if you don't use `add_option()` in the above way, you don't need to tell OptionParser which option class to use:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

Adding new actions

Adding new actions is a bit trickier, because you have to understand that *optparse* has a couple of classifications for actions:

"store" actions actions that result in *optparse* storing a value to an attribute of the current OptionValues instance; these options require a *dest* attribute to be supplied to the Option constructor.

"typed" actions actions that take a value from the command line and expect it to be of a certain type; or rather, a string that can be converted to a certain type. These options require a *type* attribute to the Option constructor.

These are overlapping sets: some default "store"actions are `"store"`, `"store_const"`, `"append"`, and `"count"`, while the default "typed" actions are `"store"`, `"append"`, and `"callback"`.

When you add an action, you need to categorize it by listing it in at least one of the following class attributes of Option (all are lists of strings):

`Option.`**`ACTIONS`**
All actions must be listed in ACTIONS.

`Option.`**`STORE_ACTIONS`**
"store" actions are additionally listed here.

`Option.`**`TYPED_ACTIONS`**
"typed" actions are additionally listed here.

`Option.`**`ALWAYS_TYPED_ACTIONS`**
Actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that *optparse* assigns the default type, `"string"`, to options with no explicit type whose action is listed in *ALWAYS_TYPED_ACTIONS*.

In order to actually implement your new action, you must override Option's `take_action()` method and add a case that recognizes your action.

For example, let's add an `"extend"` action. This is similar to the standard `"append"` action, but instead of taking a single value from the command-line and appending it to an existing list, `"extend"` will take multiple values in a single

comma-delimited string, and extend an existing list with them. That is, if `--names` is an `"extend"` option of type `"string"`, the command line

```
--names=foo,bar --names blah --names ding,dong
```

would result in a list

```
["foo", "bar", "blah", "ding", "dong"]
```

Again we define a subclass of Option:

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

Features of note:

- `"extend"` both expects a value on the command-line and stores that value somewhere, so it goes in both *STORE_ACTIONS* and *TYPED_ACTIONS*.
- to ensure that *optparse* assigns the default type of `"string"` to `"extend"` actions, we put the `"extend"` action in *ALWAYS_TYPED_ACTIONS* as well.
- `MyOption.take_action()` implements just this one new action, and passes control back to `Option.take_action()` for the standard *optparse* actions.
- `values` is an instance of the optparse_parser.Values class, which provides the very useful `ensure_value()` method. `ensure_value()` is essentially *getattr()* with a safety valve; it is called as

  ```
  values.ensure_value(attr, value)
  ```

  If the `attr` attribute of `values` doesn't exist or is `None`, then ensure_value() first sets it to `value`, and then returns 'value. This is very handy for actions like `"extend"`, `"append"`, and `"count"`, all of which accumulate data in a variable and expect that variable to be of a certain type (a list for the first two, an integer for the latter). Using `ensure_value()` means that scripts using your action don't have to worry about setting a default value for the option destinations in question; they can just leave the default as `None` and `ensure_value()` will take care of getting it right when it's needed.

15.6 getopt —C-style parser for command line options

Source code: Lib/getopt.py

注解: The *getopt* module is a parser for command line options whose API is designed to be familiar to users of the C `getopt()` function. Users who are unfamiliar with the C `getopt()` function or who would like to write less code and get better help and error messages should consider using the *argparse* module instead.

This module helps scripts to parse the command line arguments in `sys.argv`. It supports the same conventions as the Unix `getopt()` function (including the special meanings of arguments of the form '`-`' and '`--`'). Long options similar to those supported by GNU software may be used as well via an optional third argument.

This module provides two functions and an exception:

`getopt.`**`getopt`**(*args*, *options*[, *long_options*])

Parses command line options and parameter list. *args* is the argument list to be parsed, without the leading reference to the running program. Typically, this means `sys.argv[1:]`. *options* is the string of option letters that the script wants to recognize, with options that require an argument followed by a colon (`':'`; i.e., the same format that Unix `getopt()` uses).

注解: Unlike GNU `getopt()`, after a non-option argument, all further arguments are considered also non-options. This is similar to the way non-GNU Unix systems work.

---

long_options, if specified, must be a list of strings with the names of the long options which should be supported. The leading `'--'` characters should not be included in the option name. Long options which require an argument should be followed by an equal sign (`'='`). Optional arguments are not supported. To accept only long options, *options* should be an empty string. Long options on the command line can be recognized so long as they provide a prefix of the option name that matches exactly one of the accepted options. For example, if *long_options* is `['foo', 'frob']`, the option `--fo` will match as `--foo`, but `--f` will not match uniquely, so *GetoptError* will be raised.

The return value consists of two elements: the first is a list of `(option, value)` pairs; the second is the list of program arguments left after the option list was stripped (this is a trailing slice of *args*). Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., `'-x'`) or two hyphens for long options (e.g., `'--long-option'`), and the option argument as its second element, or an empty string if the option has no argument. The options occur in the list in the same order in which they were found, thus allowing multiple occurrences. Long and short options may be mixed.

`getopt.`**`gnu_getopt`**(*args*, *options*[, *long_options*])

This function works like *getopt()*, except that GNU style scanning mode is used by default. This means that option and non-option arguments may be intermixed. The *getopt()* function stops processing options as soon as a non-option argument is encountered.

If the first character of the option string is `'+'`, or if the environment variable `POSIXLY_CORRECT` is set, then option processing stops as soon as a non-option argument is encountered.

2.3 新版功能.

exception `getopt.`**`GetoptError`**

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none. The argument to the exception is a string indicating the cause of the error. For long options, an argument given to an option which does not require one will also cause this exception to be raised. The attributes `msg` and `opt` give the error message and related option; if there is no specific option to which the exception relates, `opt` is an empty string.

在 1.6 版更改: Introduced *GetoptError* as a synonym for *error*.

exception getopt.**error**
Alias for *GetoptError*; for backward compatibility.

An example using only Unix style options:

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

Using long option names is equally easy:

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x', '')]
>>> args
['a1', 'a2']
```

In a script, typical usage is something like this:

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
    except getopt.GetoptError as err:
        # print help information and exit:
        print str(err)  # will print something like "option -a not recognized"
        usage()
        sys.exit(2)
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"
    # ...

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

Note that an equivalent command line interface could be produced with less code and more informative help and error messages by using the *argparse* module:

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    args = parser.parse_args()
    # ... do something with args.output ...
    # ... do something with args.verbose ..
```

参见:

Module *argparse* Alternative command line option and argument parsing library.

15.7 模块 logging —Python 的日志记录工具

Important

此页面仅包含 API 参考信息。有关更多高级主题的教程信息和讨论，请参阅

- 基础教程
- 进阶教程
- Logging Cookbook

源代码: Lib/logging/__init__.py

2.3 新版功能.

这个模块为应用与库定义了实现灵活的事件日志系统的函数与类.

使用标准库提提供的 logging API 最主要的好处是，所有的 Python 模块都可能参与日志输出，包括你的日志消息和第三方模块的日志消息。

这个模块提供许多强大而灵活的功能。如果你对 logging 不太熟悉的话，掌握它最好的方式就是查看它对应的教程（详见右侧的链接）。

该模块定义的基础类和函数都列在下面。

- 记录器暴露了应用程序代码直接使用的接口。
- 处理程序将日志记录（由记录器创建）发送到适当的目标。
- 过滤器提供了更精细的附加功能，用于确定要输出的日志记录。
- 格式化程序指定最终输出中日志记录的样式。

15.7.1 Logger 对象

Loggers have the following attributes and methods. Note that Loggers are never instantiated directly, but always through the module-level function `logging.getLogger(name)`. Multiple calls to *getLogger()* with the same name will always return a reference to the same Logger object.

`name` 是潜在的周期分割层级值, 像 "foo.bar.baz" (例如, 抛出的可以只是明文的 "foo")。Loggers 是进一步在子层次列表的更高 loggers 列表。例如，有个名叫 "foo"的 logger，名叫 "foo.bar"，`foo.bar.baz`, 和 `foo.bam` 都是 `foo``的衍生 logger. logger 的名字分级类似 Python 包的层级，并且相同的如果你组织你的 loggers 在每模块级别基本上使用推荐的结构 ``logging.getLogger(__name__)`。这是因为在模块里，在 Python 包的命名空间的模块名为 "__name__"。

class logging.**Logger**

Logger.**propagate**
: If this evaluates to true, events logged to this logger will be passed to the handlers of higher level (ancestor) loggers, in addition to any handlers attached to this logger. Messages are passed directly to the ancestor loggers' handlers - neither the level nor filters of the ancestor loggers in question are considered.

  如果等于假，记录消息将不会传递给这个原型记录器的管理器。

  构造器将这个属性初始化为 `True`。

注解: 如果你关联了一个管理器 * 并且 * 到它自己的一个或多个记录器，它可能发出多次相同的记录。总体来说，你不需要关联管理器到一个或多个记录器 - 如果你只是关联它到一个合适的记录器等级中的最高级别记录器，它将会看到子记录器所有记录的事件，他们的传播剩下的设置为 "True"。一个通用场景是只关联管理器到根记录器，并且让传播照顾剩下的.

Logger.**setLevel**(*level*)
: Sets the threshold for this logger to *level*. Logging messages which are less severe than *level* will be ignored. When a logger is created, the level is set to `NOTSET` (which causes all messages to be processed when the logger is the root logger, or delegation to the parent when the logger is a non-root logger). Note that the root logger is created with level `WARNING`.

  委派给父级的意思是如果一个记录器的级别设置为 NOTSET，遍历其祖先记录器链，直到找到另一个 NOTSET 级别的祖先或到达根为止。

  如果发现某个父级的级别不是 NOTSET ，那么该父级的级别将被视为发起搜索的记录器的有效级别，并用于确定如何处理日志事件。

  如果到达根 logger ，并且其级别为 NOTSET，则将处理所有消息。否则，将使用根记录器的级别作为有效级别。

  参见日志级别 级别列表。

Logger.**isEnabledFor**(*lvl*)
: Indicates if a message of severity *lvl* would be processed by this logger. This method checks first the module-level level set by `logging.disable(lvl)` and then the logger's effective level as determined by *getEffectiveLevel()*.

Logger.**getEffectiveLevel**()
: 指示此记录器的有效级别。如果通过*setLevel()* 设置了除 `NOTSET` 以外的值，则返回该值。否则，将层次结构遍历到根，直到找到除 `NOTSET` 以外的其他值，然后返回该值。返回的值是一个整数，通常为 `logging.DEBUG`、`logging.INFO` 等等。

Logger.**getChild**(*suffix*)
: 返回由后缀确定的，是该记录器的后代的记录器。因此，`logging.getLogger('abc').getChild('def.ghi')` 与 `logging.getLogger('abc.def.ghi')` 将返回相同的记录器。这是一个便捷方法，当使用如 `__name__` 而不是字符串字面值命名父记录器时很有用。

2.7 新版功能.

Logger.**debug**(*msg*, **args*, ***kwargs*)

Logs a message with level DEBUG on this logger. The *msg* is the message format string, and the *args* are the arguments which are merged into *msg* using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

There are two keyword arguments in *kwargs* which are inspected: *exc_info* which, if it does not evaluate as false, causes exception information to be added to the logging message. If an exception tuple (in the format returned by *sys.exc_info()*) is provided, it is used; otherwise, *sys.exc_info()* is called to get the exception information.

The second keyword argument is *extra* which can be used to pass a dictionary which is used to populate the __dict__ of the LogRecord created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', extra=d)
```

would print something like

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

The keys in the dictionary passed in *extra* should not clash with the keys used by the logging system. (See the *Formatter* documentation for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the *Formatter* has been set up with a format string which expects 'clientip' and 'user' in the attribute dictionary of the LogRecord. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always need to pass the *extra* dictionary with these keys.

尽管这可能很烦人，但此功能旨在用于特殊情况，例如在多个上下文中执行相同代码的多线程服务器，并且出现的有趣条件取决于此上下文（例如在上面的示例中就是远程客户端 IP 地址和已验证用户名）。在这种情况下，很可能将专门的*Formatter* 与特定的 Handler 一起使用。

Logger.**info**(*msg*, **args*, ***kwargs*)

在此记录器上记录 INFO 级别的消息。参数解释同*debug()*。

Logger.**warning**(*msg*, **args*, ***kwargs*)

在此记录器上记录 WARNING 级别的消息。参数解释同*debug()*。

Logger.**error**(*msg*, **args*, ***kwargs*)

在此记录器上记录 ERROR 级别的消息。参数解释同*debug()*。

Logger.**critical**(*msg*, **args*, ***kwargs*)

在此记录器上记录 CRITICAL 级别的消息。参数解释同*debug()*。

Logger.**log**(*lvl*, *msg*, **args*, ***kwargs*)

Logs a message with integer level *lvl* on this logger. The other arguments are interpreted as for *debug()*.

Logger.**exception**(*msg*, **args*, ***kwargs*)

Logs a message with level ERROR on this logger. The arguments are interpreted as for *debug()*, except that any passed *exc_info* is not inspected. Exception info is always added to the logging message. This method should only be called from an exception handler.

Logger.**addFilter**(*filter*)

将指定的过滤器 *filter* 添加到此记录器。

Logger.**removeFilter**(*filter*)
从此记录器中删除指定的处理程序 *filter*。

Logger.**filter**(*record*)
Applies this logger's filters to the record and returns a true value if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be processed (passed to handlers). If one returns a false value, no further processing of the record occurs.

Logger.**addHandler**(*hdlr*)
将指定的处理程序 *hdlr* 添加到此记录器。

Logger.**removeHandler**(*hdlr*)
从此记录器中删除指定的处理器 *hdlr*。

Logger.**findCaller**()
Finds the caller's source filename and line number. Returns the filename, line number and function name as a 3-element tuple.

在 2.4 版更改: The function name was added. In earlier versions, the filename and line number were returned as a 2-element tuple.

Logger.**handle**(*record*)
通过将记录传递给与此记录器及其祖先关联的所有处理器来处理（直到某个 *propagate* 值为 false）。此方法用于从套接字接收的未序列化的以及在本地创建的记录。使用*filter()* 进行记录程序级别过滤。

Logger.**makeRecord**(*name*, *lvl*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*)
这是一种工厂方法，可以在子类中对其进行重写以创建专门的*LogRecord* 实例。

在 2.5 版更改: *func* and *extra* were added.

15.7.2 日志级别

日志记录级别的数值在下表中给出。如果你想要定义自己的级别，并且需要它们具有相对于预定义级别的特定值，那么这些内容可能是你感兴趣的。如果你定义具有相同数值的级别，它将覆盖预定义的值; 预定义的名称丢失。

级别	数值
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

15.7.3 处理器对象

Handler 有以下属性和方法。注意不要直接实例化 Handler；这个类用来派生其他更有用的子类。但是，子类的 __init__() 方法需要调用*Handler.__init__()* 。

Handler.**__init__**(*level=NOTSET*)
初始化 Handler 实例时，需要设置它的级别，将过滤列表置为空，并且创建锁（通过*createLock()* ）来序列化对 I/O 的访问。

Handler.**createLock**()
初始化一个线程锁，用来序列化对底层的 I/O 功能的访问，底层的 I/O 功能可能不是线程安全的。

Handler.**acquire**()
使用*createLock()* 获取线程锁。

Handler.**release**()
使用*acquire()* 来释放线程锁。

Handler.**setLevel**(*level*)
给处理器设置阈值为 *level* 。日志级别小于 *level* 将被忽略。创建处理器时，日志级别被设置为 NOTSET（所有的消息都会被处理）。

参见日志级别 级别列表。

Handler.**setFormatter**(*fmt*)
将此处理器的*Formatter* 设置为 *fmt*。

Handler.**addFilter**(*filter*)
将指定的过滤器 *filter* 添加到此处理器。

Handler.**removeFilter**(*filter*)
从此处理器中删除指定的过滤器 *filter* 。

Handler.**filter**(*record*)
Applies this handler's filters to the record and returns a true value if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be emitted. If one returns a false value, the handler will not emit the record.

Handler.**flush**()
确保所有日志记录从缓存输出。此版本不执行任何操作，并且应由子类实现。

Handler.**close**()
整理处理器使用的所有资源。此版本不输出，但从内部处理器列表中删除处理器，内部处理器在*shutdown()* 被调用时关闭。子类应确保从重写的*close()* 方法中调用此方法。

Handler.**handle**(*record*)
经已添加到处理器的过滤器过滤后，有条件地发出指定的日志记录。用获取/释放 I/O 线程锁包装记录的实际发出行为。

Handler.**handleError**(*record*)
调用*emit()* 期间遇到异常时，应从处理器中调用此方法。如果模块级属性 raiseExceptions 是 False，则异常将被静默忽略。这是大多数情况下日志系统需要的——大多数用户不会关心日志系统中的错误，他们对应用程序错误更感兴趣。但是，你可以根据需要将其替换为自定义处理器。指定的记录是发生异常时正在处理的记录。（raiseExceptions 的默认值是 True，因为这在开发过程中是比较有用的）。

Handler.**format**(*record*)
如果设置了格式器则用其对记录进行格式化。否则，使用模块的默认格式器。

Handler.**emit**(*record*)
执行实际记录给定日志记录所需的操作。这个版本应由子类实现，因此这里直接引发*NotImplementedError* 异常。

有关作为标准随附的处理程序，请参见*logging.handlers*。

15.7.4 格式器对象

Formatter 对象拥有以下的属性和方法。一般情况下，它们负责将*LogRecord* 转换为可由人或外部系统解释的字符串。基础的*Formatter* 允许指定格式字符串。如果未提供任何值，则使用默认值 `'%(message)s'`，它仅将消息包括在日志记录调用中。要在格式化输出中包含其他信息（如时间戳），请阅读下文。

A Formatter can be initialized with a format string which makes use of knowledge of the *LogRecord* attributes - such as the default value mentioned above making use of the fact that the user's message and arguments are pre-formatted into a *LogRecord*'s *message* attribute. This format string contains standard Python %-style mapping keys. See section *String Formatting Operations* for more information on string formatting.

The useful mapping keys in a *LogRecord* are given in the section on *LogRecord 属性*.

class logging.**Formatter**(*fmt=None*, *datefmt=None*)

Returns a new instance of the *Formatter* class. The instance is initialized with a format string for the message as a whole, as well as a format string for the date/time portion of a message. If no *fmt* is specified, `'%(message)s'` is used. If no *datefmt* is specified, the ISO8601 date format is used.

format(*record*)

The record's attribute dictionary is used as the operand to a string formatting operation. Returns the resulting string. Before formatting the dictionary, a couple of preparatory steps are carried out. The *message* attribute of the record is computed using *msg % args*. If the formatting string contains `'(asctime)'`, *formatTime()* is called to format the event time. If there is exception information, it is formatted using *formatException()* and appended to the message. Note that the formatted exception information is cached in attribute *exc_text*. This is useful because the exception information can be pickled and sent across the wire, but you should be careful if you have more than one *Formatter* subclass which customizes the formatting of exception information. In this case, you will have to clear the cached value after a formatter has done its formatting, so that the next formatter to handle the event doesn't use the cached value but recalculates it afresh.

formatTime(*record*, *datefmt=None*)

This method should be called from *format()* by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behavior is as follows: if *datefmt* (a string) is specified, it is used with *time.strftime()* to format the creation time of the record. Otherwise, the ISO8601 format is used. The resulting string is returned.

This function uses a user-configurable function to convert the creation time to a tuple. By default, *time.localtime()* is used; to change this for a particular formatter instance, set the `converter` attribute to a function with the same signature as *time.localtime()* or *time.gmtime()*. To change it for all formatters, for example if you want all logging times to be shown in GMT, set the `converter` attribute in the `Formatter` class.

formatException(*exc_info*)

Formats the specified exception information (a standard exception tuple as returned by *sys.exc_info()*) as a string. This default implementation just uses *traceback.print_exception()*. The resulting string is returned.

15.7.5 Filter Objects

`Filters` can be used by `Handlers` and `Loggers` for more sophisticated filtering than is provided by levels. The base filter class only allows events which are below a certain point in the logger hierarchy. For example, a filter initialized with 'A.B' will allow events logged by loggers 'A.B', 'A.B.C', 'A.B.C.D', 'A.B.D' etc. but not 'A.BB', 'B.A.B' etc. If initialized with the empty string, all events are passed.

class `logging.`**Filter**(*name=''*)

Returns an instance of the *Filter* class. If *name* is specified, it names a logger which, together with its children, will have its events allowed through the filter. If *name* is the empty string, allows every event.

filter(*record*)

Is the specified record to be logged? Returns zero for no, nonzero for yes. If deemed appropriate, the record may be modified in-place by this method.

Note that filters attached to handlers are consulted before an event is emitted by the handler, whereas filters attached to loggers are consulted whenever an event is logged (using *debug()*, *info()*, etc.), before sending an event to handlers. This means that events which have been generated by descendant loggers will not be filtered by a logger's filter setting, unless the filter has also been applied to those descendant loggers.

You don't actually need to subclass `Filter`: you can pass any instance which has a `filter` method with the same semantics.

Although filters are used primarily to filter records based on more sophisticated criteria than levels, they get to see every record which is processed by the handler or logger they're attached to: this can be useful if you want to do things like counting how many records were processed by a particular logger or handler, or adding, changing or removing attributes in the LogRecord being processed. Obviously changing the LogRecord needs to be done with some care, but it does allow the injection of contextual information into logs (see filters-contextual).

15.7.6 LogRecord Objects

LogRecord instances are created automatically by the *Logger* every time something is logged, and can be created manually via *makeLogRecord()* (for example, from a pickled event received over the wire).

class `logging.`**LogRecord**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*)

Contains all the information pertinent to the event being logged.

The primary information is passed in `msg` and `args`, which are combined using `msg % args` to create the `message` field of the record.

参数

- **name** –The name of the logger used to log the event represented by this LogRecord. Note that this name will always have this value, even though it may be emitted by a handler attached to a different (ancestor) logger.
- **level** –The numeric level of the logging event (one of DEBUG, INFO etc.) Note that this is converted to *two* attributes of the LogRecord: `levelno` for the numeric value and `levelname` for the corresponding level name.
- **pathname** –The full pathname of the source file where the logging call was made.
- **lineno** –The line number in the source file where the logging call was made.
- **msg** –The event description message, possibly a format string with placeholders for variable data.
- **args** –Variable data to merge into the *msg* argument to obtain the event description.

- **exc_info** –An exception tuple with the current exception information, or `None` if no exception information is available.
- **func** –The name of the function or method from which the logging call was invoked.

在 2.5 版更改: *func* was added.

getMessage()

Returns the message for this *LogRecord* instance after merging any user-supplied arguments with the message. If the user-supplied message argument to the logging call is not a string, *str()* is called on it to convert it to a string. This allows use of user-defined classes as messages, whose `__str__` method can return the actual format string to be used.

15.7.7 LogRecord 属性

The LogRecord has a number of attributes, most of which are derived from the parameters to the constructor. (Note that the names do not always correspond exactly between the LogRecord constructor parameters and the LogRecord attributes.) These attributes can be used to merge data from the record into the format string. The following table lists (in alphabetical order) the attribute names, their meanings and the corresponding placeholder in a %-style format string.

属性名称	格式	描述
args	不需要格式化。	The tuple of arguments merged into `msg` to produce `message`, or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).
asctime	`%(asctime)s`	Human-readable time when the *`LogRecord`* was created. By default this is of the form '2003-07-08 16:49:45,896' (the numbers after the comma are millisecond portion of the time).
created	`%(created)f`	Time when the *`LogRecord`* was created (as returned by *`time.time()`*).
exc_info	不需要格式化。	Exception tuple (à la `sys.exc_info`) or, if no exception has occurred, `None`.
filename	`%(filename)s`	Filename portion of `pathname`.
funcName	`%(funcName)s`	Name of function containing the logging call.
levelname	`%(levelname)s`	Text logging level for the message (`'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`).
levelno	`%(levelno)s`	Numeric logging level for the message (`DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`).
lineno	`%(lineno)d`	Source line number where the logging call was issued (if available).
module 模块	`%(module)s`	模块 (`filename` 的名称部分)。
msecs	`%(msecs)d`	Millisecond portion of the time when the *`LogRecord`* was created.
message	`%(message)s`	The logged message, computed as `msg % args`. This is set when *`Formatter.format()`* is invoked.
msg	不需要格式化。	The format string passed in the original logging call. Merged with `args` to produce `message`, or an arbitrary object (see arbitrary-object-messages).
名称	`%(name)s`	Name of the logger used to log the call.
pathname	`%(pathname)s`	Full pathname of the source file where the logging call was issued (if available).
process	`%(process)d`	进程 ID（如果可用）
processName	`%(processName)s`	进程名（如果可用）
relativeCreated	`%(relativeCreated)d`	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded.
thread	`%(thread)d`	线程 ID（如果可用）
threadName	`%(threadName)s`	线程名（如果可用）

在 2.5 版更改: *funcName* was added.

在 2.6 版更改: 添加了 *processName*

15.7.8 LoggerAdapter 对象

LoggerAdapter instances are used to conveniently pass contextual information into logging calls. For a usage example, see the section on adding contextual information to your logging output.

2.6 新版功能.

class logging.**LoggerAdapter**(*logger*, *extra*)

Returns an instance of *LoggerAdapter* initialized with an underlying *Logger* instance and a dict-like object.

process(*msg*, *kwargs*)

Modifies the message and/or keyword arguments passed to a logging call in order to insert contextual information. This implementation takes the object passed as *extra* to the constructor and adds it to *kwargs* using key 'extra'. The return value is a (*msg*, *kwargs*) tuple which has the (possibly modified) versions of the arguments passed in.

In addition to the above, *LoggerAdapter* supports the following methods of *Logger*: *debug()*, *info()*, *warning()*, *error()*, *exception()*, *critical()*, *log()* and *isEnabledFor()*. These methods have the same signatures as their counterparts in *Logger*, so you can use the two types of instances interchangeably for these calls.

在 2.7 版更改: The *isEnabledFor()* method was added to *LoggerAdapter*. This method delegates to the underlying logger.

15.7.9 线程安全

The logging module is intended to be thread-safe without any special work needing to be done by its clients. It achieves this though using threading locks; there is one lock to serialize access to the module's shared data, and each handler also creates a lock to serialize access to its underlying I/O.

If you are implementing asynchronous signal handlers using the *signal* module, you may not be able to use logging from within such handlers. This is because lock implementations in the *threading* module are not always re-entrant, and so cannot be invoked from such signal handlers.

15.7.10 模块级别函数

In addition to the classes described above, there are a number of module- level functions.

logging.**getLogger**([*name*])

Return a logger with the specified name or, if no name is specified, return a logger which is the root logger of the hierarchy. If specified, the name is typically a dot-separated hierarchical name like "*a*", "*a.b*" or "*a.b.c.d*". Choice of these names is entirely up to the developer who is using logging.

All calls to this function with a given name return the same logger instance. This means that logger instances never need to be passed between different parts of an application.

logging.**getLoggerClass**()

Return either the standard *Logger* class, or the last class passed to *setLoggerClass()*. This function may be called from within a new class definition, to ensure that installing a customized *Logger* class will not undo customizations already applied by other code. For example:

```
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

logging.**debug**(*msg*[, **args*[, ***kwargs*]])

Logs a message with level DEBUG on the root logger. The *msg* is the message format string, and the *args* are the

arguments which are merged into *msg* using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

There are two keyword arguments in *kwargs* which are inspected: *exc_info* which, if it does not evaluate as false, causes exception information to be added to the logging message. If an exception tuple (in the format returned by *sys.exc_info()*) is provided, it is used; otherwise, *sys.exc_info()* is called to get the exception information.

The other optional keyword argument is *extra* which can be used to pass a dictionary which is used to populate the __dict__ of the LogRecord created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```
FORMAT = "%(asctime)-15s %(clientip)s %(user)-8s %(message)s"
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logging.warning("Protocol problem: %s", "connection reset", extra=d)
```

would print something like:

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

The keys in the dictionary passed in *extra* should not clash with the keys used by the logging system. (See the *Formatter* documentation for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the *Formatter* has been set up with a format string which expects 'clientip' and 'user' in the attribute dictionary of the LogRecord. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always need to pass the *extra* dictionary with these keys.

尽管这可能很烦人，但此功能旨在用于特殊情况，例如在多个上下文中执行相同代码的多线程服务器，并且出现的有趣条件取决于此上下文（例如在上面的示例中就是远程客户端 IP 地址和已验证用户名）。在这种情况下，很可能将专门的*Formatter* 与特定的 Handler 一起使用。

在 2.5 版更改: *extra* was added.

logging.**info**(*msg*[, **args*[, ***kwargs*]])
Logs a message with level INFO on the root logger. The arguments are interpreted as for *debug()*.

logging.**warning**(*msg*[, **args*[, ***kwargs*]])
Logs a message with level WARNING on the root logger. The arguments are interpreted as for *debug()*.

logging.**error**(*msg*[, **args*[, ***kwargs*]])
Logs a message with level ERROR on the root logger. The arguments are interpreted as for *debug()*.

logging.**critical**(*msg*[, **args*[, ***kwargs*]])
Logs a message with level CRITICAL on the root logger. The arguments are interpreted as for *debug()*.

logging.**exception**(*msg*[, **args*[, ***kwargs*]])
Logs a message with level ERROR on the root logger. The arguments are interpreted as for *debug()*, except that any passed *exc_info* is not inspected. Exception info is always added to the logging message. This function should only be called from an exception handler.

logging.**log**(*level*, *msg*[, **args*[, ***kwargs*]])
Logs a message with level *level* on the root logger. The other arguments are interpreted as for *debug()*.

注解: The above module-level convenience functions, which delegate to the root logger, call *basicConfig()* to ensure that at least one handler is available. Because of this, they should *not* be used in threads, in versions of Python earlier than 2.7.1 and 3.2, unless at least one handler has been added to the root logger *before* the threads are started. In earlier versions of Python, due to a thread safety shortcoming in *basicConfig()*, this can (under

rare circumstances) lead to handlers being added multiple times to the root logger, which can in turn lead to multiple messages for the same event.

`logging.`**`disable`**(*lvl*)

Provides an overriding level *lvl* for all loggers which takes precedence over the logger's own level. When the need arises to temporarily throttle logging output down across the whole application, this function can be useful. Its effect is to disable all logging calls of severity *lvl* and below, so that if you call it with a value of INFO, then all INFO and DEBUG events would be discarded, whereas those of severity WARNING and above would be processed according to the logger's effective level. If `logging.disable(logging.NOTSET)` is called, it effectively removes this overriding level, so that logging output again depends on the effective levels of individual loggers.

`logging.`**`addLevelName`**(*lvl*, *levelName*)

Associates level *lvl* with text *levelName* in an internal dictionary, which is used to map numeric levels to a textual representation, for example when a *Formatter* formats a message. This function can also be used to define your own levels. The only constraints are that all levels used must be registered using this function, levels should be positive integers and they should increase in increasing order of severity.

注解: If you are thinking of defining your own levels, please see the section on custom-levels.

`logging.`**`getLevelName`**(*lvl*)

Returns the textual representation of logging level *lvl*. If the level is one of the predefined levels `CRITICAL`, `ERROR`, `WARNING`, `INFO` or `DEBUG` then you get the corresponding string. If you have associated levels with names using *addLevelName()* then the name you have associated with *lvl* is returned. If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned. Otherwise, the string "Level %s" % lvl is returned.

注解: Integer levels should be used when e.g. setting levels on instances of *Logger* and handlers. This function is used to convert between an integer level and the level name displayed in the formatted log output by means of the `%(levelname)s` format specifier (see *LogRecord 属性*).

`logging.`**`makeLogRecord`**(*attrdict*)

Creates and returns a new *LogRecord* instance whose attributes are defined by *attrdict*. This function is useful for taking a pickled *LogRecord* attribute dictionary, sent over a socket, and reconstituting it as a *LogRecord* instance at the receiving end.

`logging.`**`basicConfig`**([****kwargs**])

Does basic configuration for the logging system by creating a *StreamHandler* with a default *Formatter* and adding it to the root logger. The functions *debug()*, *info()*, *warning()*, *error()* and *critical()* will call *basicConfig()* automatically if no handlers are defined for the root logger.

This function does nothing if the root logger already has handlers configured for it.

在 2.4 版更改: Formerly, *basicConfig()* did not take any keyword arguments.

注解: This function should be called from the main thread before other threads are started. In versions of Python prior to 2.7.1 and 3.2, if this function is called from multiple threads, it is possible (in rare circumstances) that a handler will be added to the root logger more than once, leading to unexpected results such as messages being duplicated in the log.

支持以下关键字参数。

格式	描述
filename	使用指定的文件名而不是 StreamHandler 创建 FileHandler。
filemode	If *filename* is specified, open the file in this mode. Defaults to `'a'`.
format	处理器使用的指定格式字符串。
datefmt	Use the specified date/time format, as accepted by *time.strftime()*.
level	Set the root logger level to the specified *level*.
stream	Use the specified stream to initialize the StreamHandler. Note that this argument is incompatible with *filename* - if both are present, *stream* is ignored.

`logging.`**`shutdown`**`()`
Informs the logging system to perform an orderly shutdown by flushing and closing all handlers. This should be called at application exit and no further use of the logging system should be made after this call.

`logging.`**`setLoggerClass`**`(`*klass*`)`
Tells the logging system to use the class *klass* when instantiating a logger. The class should define `__init__()` such that only a name argument is required, and the `__init__()` should call `Logger.__init__()`. This function is typically called before any loggers are instantiated by applications which need to use custom logger behavior.

15.7.11 与警告模块集成

captureWarnings() 函数可用来将*logging* 和*warnings* 模块集成。

`logging.`**`captureWarnings`**`(`*capture*`)`
此函数用于打开和关闭日志系统对警告的捕获。

如果 *capture* 是 `True`，则*warnings* 模块发出的警告将重定向到日志记录系统。具体来说，将使用*warnings.formatwarning()* 格式化警告信息，并将结果字符串使用 `WARNING` 等级记录到名为 `'py.warnings'` 的记录器中。

如果 *capture* 是 `False`，则将停止将警告重定向到日志记录系统，并且将警告重定向到其原始目标（即在 `captureWarnings(True)` 调用之前的有效目标）。

参见:

模块*logging.config* 日志记录模块的配置 API 。

模块*logging.handlers* 日志记录模块附带的有用处理程序。

PEP 282 - Logging 系统 该提案描述了 Python 标准库中包含的这个特性。

Original Python logging package 这是该*logging* 包的原始来源。该站点提供的软件包版本适用于 Python 1.5.2、2.1.x 和 2.2.x，它们不被*logging* 包含在标准库中。

15.8 `logging.config` —日志记录配置

Important

此页面仅包含参考信息。有关教程，请参阅

- 基础教程
- 进阶教程

- Logging Cookbook

源代码： Lib/logging/config.py

This section describes the API for configuring the logging module.

15.8.1 Configuration functions

The following functions configure the logging module. They are located in the *logging.config* module. Their use is optional —you can configure the logging module using these functions or by making calls to the main API (defined in *logging* itself) and defining handlers which are declared either in *logging* or *logging.handlers*.

logging.config.**dictConfig**(*config*)

Takes the logging configuration from a dictionary. The contents of this dictionary are described in *Configuration dictionary schema* below.

If an error is encountered during configuration, this function will raise a *ValueError*, *TypeError*, *AttributeError* or *ImportError* with a suitably descriptive message. The following is a (possibly incomplete) list of conditions which will raise an error:

- A `level` which is not a string or which is a string not corresponding to an actual logging level.
- A `propagate` value which is not a boolean.
- An id which does not have a corresponding destination.
- A non-existent handler id found during an incremental call.
- An invalid logger name.
- Inability to resolve to an internal or external object.

Parsing is performed by the `DictConfigurator` class, whose constructor is passed the dictionary used for configuration, and has a `configure()` method. The *logging.config* module has a callable attribute `dictConfigClass` which is initially set to `DictConfigurator`. You can replace the value of `dictConfigClass` with a suitable implementation of your own.

dictConfig() calls `dictConfigClass` passing the specified dictionary, and then calls the `configure()` method on the returned object to put the configuration into effect:

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

For example, a subclass of `DictConfigurator` could call `DictConfigurator.__init__()` in its own `__init__()`, then set up custom prefixes which would be usable in the subsequent `configure()` call. `dictConfigClass` would be bound to this new subclass, and then *dictConfig()* could be called exactly as in the default, uncustomized state.

2.7 新版功能.

logging.config.**fileConfig**(*fname*, *defaults=None*, *disable_existing_loggers=True*)

Reads the logging configuration from a `configparser`-format file named *fname*. The format of the file should be as described in *Configuration file format*. This function can be called several times from an application, allowing an end user to select from various pre-canned configurations (if the developer provides a mechanism to present the choices and load the chosen configuration).

参数

- **defaults** –Defaults to be passed to the ConfigParser can be specified in this argument.
- **disable_existing_loggers** –If specified as `False`, loggers which exist when this call is made are left enabled. The default is `True` because this enables old behaviour in a backward-compatible way. This behaviour is to disable any existing loggers unless they or their ancestors are explicitly named in the logging configuration.

在 2.6 版更改: The `disable_existing_loggers` keyword argument was added. Previously, existing loggers were *always* disabled.

`logging.config.`**`listen`**(*port=DEFAULT_LOGGING_CONFIG_PORT*)

Starts up a socket server on the specified port, and listens for new configurations. If no port is specified, the module's default `DEFAULT_LOGGING_CONFIG_PORT` is used. Logging configurations will be sent as a file suitable for processing by *`fileConfig()`*. Returns a *`Thread`* instance on which you can call *`start()`* to start the server, and which you can *`join()`* when appropriate. To stop the server, call *`stopListening()`*.

To send a configuration to the socket, read in the configuration file and send it to the socket as a string of bytes preceded by a four-byte length string packed in binary using `struct.pack('>L', n)`.

注解: Because portions of the configuration are passed through *`eval()`*, use of this function may open its users to a security risk. While the function only binds to a socket on `localhost`, and so does not accept connections from remote machines, there are scenarios where untrusted code could be run under the account of the process which calls *`listen()`*. Specifically, if the process calling *`listen()`* runs on a multi-user machine where users cannot trust each other, then a malicious user could arrange to run essentially arbitrary code in a victim user's process, simply by connecting to the victim's *`listen()`* socket and sending a configuration which runs whatever code the attacker wants to have executed in the victim's process. This is especially easy to do if the default port is used, but not hard even if a different port is used).

`logging.config.`**`stopListening`**()

Stops the listening server which was created with a call to *`listen()`*. This is typically called before calling `join()` on the return value from *`listen()`*.

15.8.2 Configuration dictionary schema

Describing a logging configuration requires listing the various objects to create and the connections between them; for example, you may create a handler named 'console' and then say that the logger named 'startup' will send its messages to the 'console' handler. These objects aren't limited to those provided by the *`logging`* module because you might write your own formatter or handler class. The parameters to these classes may also need to include external objects such as `sys.stderr`. The syntax for describing these objects and connections is defined in *Object connections* below.

Dictionary Schema Details

The dictionary passed to *`dictConfig()`* must contain the following keys:

- *version* - to be set to an integer value representing the schema version. The only valid value at present is 1, but having this key allows the schema to evolve while still preserving backwards compatibility.

All other keys are optional, but if present they will be interpreted as described below. In all cases below where a 'configuring dict' is mentioned, it will be checked for the special `'()'` key to see if a custom instantiation is required. If so, the mechanism described in *User-defined objects* below is used to create an instance; otherwise, the context is used to determine what to instantiate.

- *formatters* - the corresponding value will be a dict in which each key is a formatter id and each value is a dict describing how to configure the corresponding *`Formatter`* instance.

The configuring dict is searched for keys `format` and `datefmt` (with defaults of `None`) and these are used to construct a *Formatter* instance.

- *filters* - the corresponding value will be a dict in which each key is a filter id and each value is a dict describing how to configure the corresponding Filter instance.

  The configuring dict is searched for the key `name` (defaulting to the empty string) and this is used to construct a *logging.Filter* instance.

- *handlers* - the corresponding value will be a dict in which each key is a handler id and each value is a dict describing how to configure the corresponding Handler instance.

  The configuring dict is searched for the following keys:

  - `class` (mandatory). This is the fully qualified name of the handler class.
  - `level` (optional). The level of the handler.
  - `formatter` (optional). The id of the formatter for this handler.
  - `filters` (optional). A list of ids of the filters for this handler.

  All *other* keys are passed through as keyword arguments to the handler's constructor. For example, given the snippet:

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

  the handler with id `console` is instantiated as a *logging.StreamHandler*, using `sys.stdout` as the underlying stream. The handler with id `file` is instantiated as a *logging.handlers.RotatingFileHandler* with the keyword arguments `filename='logconfig.log', maxBytes=1024, backupCount=3`.

- *loggers* - the corresponding value will be a dict in which each key is a logger name and each value is a dict describing how to configure the corresponding Logger instance.

  The configuring dict is searched for the following keys:

  - `level` (optional). The level of the logger.
  - `propagate` (optional). The propagation setting of the logger.
  - `filters` (optional). A list of ids of the filters for this logger.
  - `handlers` (optional). A list of ids of the handlers for this logger.

  The specified loggers will be configured according to the level, propagation, filters and handlers specified.

- *root* - this will be the configuration for the root logger. Processing of the configuration will be as for any logger, except that the `propagate` setting will not be applicable.

- *incremental* - whether the configuration is to be interpreted as incremental to the existing configuration. This value defaults to `False`, which means that the specified configuration replaces the existing configuration with the same semantics as used by the existing *`fileConfig()`* API.

  If the specified value is `True`, the configuration is processed as described in the section on *Incremental Configuration*.

- *disable_existing_loggers* - whether any existing loggers are to be disabled. This setting mirrors the parameter of the same name in *`fileConfig()`*. If absent, this parameter defaults to `True`. This value is ignored if *incremental* is `True`.

Incremental Configuration

It is difficult to provide complete flexibility for incremental configuration. For example, because objects such as filters and formatters are anonymous, once a configuration is set up, it is not possible to refer to such anonymous objects when augmenting a configuration.

Furthermore, there is not a compelling case for arbitrarily altering the object graph of loggers, handlers, filters, formatters at run-time, once a configuration is set up; the verbosity of loggers and handlers can be controlled just by setting levels (and, in the case of loggers, propagation flags). Changing the object graph arbitrarily in a safe way is problematic in a multi-threaded environment; while not impossible, the benefits are not worth the complexity it adds to the implementation.

Thus, when the `incremental` key of a configuration dict is present and is `True`, the system will completely ignore any `formatters` and `filters` entries, and process only the `level` settings in the `handlers` entries, and the `level` and `propagate` settings in the `loggers` and `root` entries.

Using a value in the configuration dict lets configurations to be sent over the wire as pickled dicts to a socket listener. Thus, the logging verbosity of a long-running application can be altered over time with no need to stop and restart the application.

Object connections

The schema describes a set of logging objects - loggers, handlers, formatters, filters - which are connected to each other in an object graph. Thus, the schema needs to represent connections between the objects. For example, say that, once configured, a particular logger has attached to it a particular handler. For the purposes of this discussion, we can say that the logger represents the source, and the handler the destination, of a connection between the two. Of course in the configured objects this is represented by the logger holding a reference to the handler. In the configuration dict, this is done by giving each destination object an id which identifies it unambiguously, and then using the id in the source object's configuration to indicate that a connection exists between the source and the destination object with that id.

So, for example, consider the following YAML snippet:

```
formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
handlers:
  h1: #This is an id
   # configuration of handler with id 'h1' goes here
   formatter: brief
  h2: #This is another id
   # configuration of handler with id 'h2' goes here
   formatter: precise
loggers:
  foo.bar.baz:
```

(下页继续)

(续上页)

```
    # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]
```

(Note: YAML used here because it's a little more readable than the equivalent Python source form for the dictionary.)

The ids for loggers are the logger names which would be used programmatically to obtain a reference to those loggers, e.g. `foo.bar.baz`. The ids for Formatters and Filters can be any string value (such as `brief`, `precise` above) and they are transient, in that they are only meaningful for processing the configuration dictionary and used to determine connections between objects, and are not persisted anywhere when the configuration call is complete.

The above snippet indicates that logger named `foo.bar.baz` should have two handlers attached to it, which are described by the handler ids `h1` and `h2`. The formatter for `h1` is that described by id `brief`, and the formatter for `h2` is that described by id `precise`.

User-defined objects

The schema supports user-defined objects for handlers, filters and formatters. (Loggers do not need to have different types for different instances, so there is no support in this configuration schema for user-defined logger classes.)

Objects to be configured are described by dictionaries which detail their configuration. In some places, the logging system will be able to infer from the context how an object is to be instantiated, but when a user-defined object is to be instantiated, the system will not know how to do this. In order to provide complete flexibility for user-defined object instantiation, the user needs to provide a 'factory' - a callable which is called with a configuration dictionary and which returns the instantiated object. This is signalled by an absolute import path to the factory being made available under the special key `'()'`. Here's a concrete example:

```
formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  custom:
      (): my.package.customFormatterFactory
      bar: baz
      spam: 99.9
      answer: 42
```

The above YAML snippet defines three formatters. The first, with id `brief`, is a standard *`logging.Formatter`* instance with the specified format string. The second, with id `default`, has a longer format and also defines the time format explicitly, and will result in a *`logging.Formatter`* initialized with those two format strings. Shown in Python source form, the `brief` and `default` formatters have configuration sub-dictionaries:

```
{
  'format' : '%(message)s'
}
```

和:

```
{
  'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
  'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

respectively, and as these dictionaries do not contain the special key `'()'`, the instantiation is inferred from the context: as a result, standard *logging.Formatter* instances are created. The configuration sub-dictionary for the third formatter, with id `custom`, is:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42
}
```

and this contains the special key `'()'`, which means that user-defined instantiation is wanted. In this case, the specified factory callable will be used. If it is an actual callable it will be used directly - otherwise, if you specify a string (as in the example) the actual callable will be located using normal import mechanisms. The callable will be called with the **remaining** items in the configuration sub-dictionary as keyword arguments. In the above example, the formatter with id `custom` will be assumed to be returned by the call:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

The key `'()'` has been used as the special key because it is not a valid keyword parameter name, and so will not clash with the names of the keyword arguments used in the call. The `'()'` also serves as a mnemonic that the corresponding value is a callable.

Access to external objects

There are times where a configuration needs to refer to objects external to the configuration, for example `sys.stderr`. If the configuration dict is constructed using Python code, this is straightforward, but a problem arises when the configuration is provided via a text file (e.g. JSON, YAML). In a text file, there is no standard way to distinguish `sys.stderr` from the literal string `'sys.stderr'`. To facilitate this distinction, the configuration system looks for certain special prefixes in string values and treat them specially. For example, if the literal string `'ext://sys.stderr'` is provided as a value in the configuration, then the `ext://` will be stripped off and the remainder of the value processed using normal import mechanisms.

The handling of such prefixes is done in a way analogous to protocol handling: there is a generic mechanism to look for prefixes which match the regular expression `^(?P<prefix>[a-z]+)://(?P<suffix>.*)$` whereby, if the `prefix` is recognised, the `suffix` is processed in a prefix-dependent manner and the result of the processing replaces the string value. If the prefix is not recognised, then the string value will be left as-is.

Access to internal objects

As well as external objects, there is sometimes also a need to refer to objects in the configuration. This will be done implicitly by the configuration system for things that it knows about. For example, the string value `'DEBUG'` for a `level` in a logger or handler will automatically be converted to the value `logging.DEBUG`, and the `handlers`, `filters` and `formatter` entries will take an object id and resolve to the appropriate destination object.

However, a more generic mechanism is needed for user-defined objects which are not known to the *logging* module. For example, consider *logging.handlers.MemoryHandler*, which takes a `target` argument which is another handler to delegate to. Since the system already knows about this class, then in the configuration, the given `target` just needs to be the object id of the relevant target handler, and the system will resolve to the handler from the id. If, however, a user defines a `my.package.MyHandler` which has an `alternate` handler, the configuration system would not know that the `alternate` referred to a handler. To cater for this, a generic resolution system allows the user to specify:

```
handlers:
  file:
```

(下页继续)

(续上页)

```
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

The literal string `'cfg://handlers.file'` will be resolved in an analogous way to strings with the `ext://` prefix, but looking in the configuration itself rather than the import namespace. The mechanism allows access by dot or by index, in a similar way to that provided by `str.format`. Thus, given the following snippet:

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

in the configuration, the string `'cfg://handlers'` would resolve to the dict with key `handlers`, the string `'cfg://handlers.email` would resolve to the dict with key `email` in the `handlers` dict, and so on. The string `'cfg://handlers.email.toaddrs[1]` would resolve to `'dev_team.domain.tld'` and the string `'cfg://handlers.email.toaddrs[0]'` would resolve to the value `'support_team@domain.tld'`. The `subject` value could be accessed using either `'cfg://handlers.email.subject'` or, equivalently, `'cfg://handlers.email[subject]'`. The latter form only needs to be used if the key contains spaces or non-alphanumeric characters. If an index value consists only of decimal digits, access will be attempted using the corresponding integer value, falling back to the string value if needed.

Given a string `cfg://handlers.myhandler.mykey.123`, this will resolve to `config_dict['handlers']['myhandler']['mykey']['123']`. If the string is specified as `cfg://handlers.myhandler.mykey[123]`, the system will attempt to retrieve the value from `config_dict['handlers']['myhandler']['mykey'][123]`, and fall back to `config_dict['handlers']['myhandler']['mykey']['123']` if that fails.

Import resolution and custom importers

Import resolution, by default, uses the builtin *`__import__()`* function to do its importing. You may want to replace this with your own importing mechanism: if so, you can replace the `importer` attribute of the `DictConfigurator` or its superclass, the `BaseConfigurator` class. However, you need to be careful because of the way functions are accessed from classes via descriptors. If you are using a Python callable to do your imports, and you want to define it at class level rather than instance level, you need to wrap it with *`staticmethod()`*. For example:

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

You don't need to wrap with *`staticmethod()`* if you're setting the import callable on a configurator *instance*.

15.8.3 Configuration file format

The configuration file format understood by *fileConfig()* is based on configparser functionality. The file must contain sections called [loggers], [handlers] and [formatters] which identify by name the entities of each type which are defined in the file. For each such entity, there is a separate section which identifies how that entity is configured. Thus, for a logger named log01 in the [loggers] section, the relevant configuration details are held in a section [logger_log01]. Similarly, a handler called hand01 in the [handlers] section will have its configuration held in a section called [handler_hand01], while a formatter called form01 in the [formatters] section will have its configuration specified in a section called [formatter_form01]. The root logger configuration must be specified in a section called [logger_root].

注解： The *fileConfig()* API is older than the *dictConfig()* API and does not provide functionality to cover certain aspects of logging. For example, you cannot configure *Filter* objects, which provide for filtering of messages beyond simple integer levels, using *fileConfig()*. If you need to have instances of *Filter* in your logging configuration, you will need to use *dictConfig()*. Note that future enhancements to configuration functionality will be added to *dictConfig()*, so it's worth considering transitioning to this newer API when it's convenient to do so.

Examples of these sections in the file are given below.

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

The root logger must specify a level and a list of handlers. An example of a root logger section is given below.

```
[logger_root]
level=NOTSET
handlers=hand01
```

The level entry can be one of DEBUG, INFO, WARNING, ERROR, CRITICAL or NOTSET. For the root logger only, NOTSET means that all messages will be logged. Level values are *eval()*uated in the context of the logging package's namespace.

The handlers entry is a comma-separated list of handler names, which must appear in the [handlers] section. These names must appear in the [handlers] section and have corresponding sections in the configuration file.

For loggers other than the root logger, some additional information is required. This is illustrated by the following example.

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

The level and handlers entries are interpreted as for the root logger, except that if a non-root logger's level is specified as NOTSET, the system consults loggers higher up the hierarchy to determine the effective level of the logger. The propagate entry is set to 1 to indicate that messages must propagate to handlers higher up the logger hierarchy from this logger, or 0 to indicate that messages are **not** propagated to handlers up the hierarchy. The qualname entry is the hierarchical channel name of the logger, that is to say the name used by the application to get the logger.

Sections which specify handler configuration are exemplified by the following.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

The `class` entry indicates the handler's class (as determined by *eval()* in the `logging` package's namespace). The `level` is interpreted as for loggers, and `NOTSET` is taken to mean 'log everything'.

在 2.6 版更改: Added support for resolving the handler's class as a dotted module and class name.

The `formatter` entry indicates the key name of the formatter for this handler. If blank, a default formatter (`logging._defaultFormatter`) is used. If a name is specified, it must appear in the `[formatters]` section and have a corresponding section in the configuration file.

The `args` entry, when *eval()*uated in the context of the `logging` package's namespace, is the list of arguments to the constructor for the handler class. Refer to the constructors for the relevant handlers, or to the examples below, to see how typical entries are constructed.

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
```

(下页继续)

(续上页)

```
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
```

Sections which specify formatter configuration are typified by the following.

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
datefmt=
class=logging.Formatter
```

The `format` entry is the overall format string, and the `datefmt` entry is the `strftime()`-compatible date/time format string. If empty, the package substitutes ISO8601 format date/times, which is almost equivalent to specifying the date format string `'%Y-%m-%d %H:%M:%S'`. The ISO8601 format also specifies milliseconds, which are appended to the result of using the above format string, with a comma separator. An example time in ISO8601 format is `2003-01-23 00:29:50,411`.

The `class` entry is optional. It indicates the name of the formatter's class (as a dotted module and class name.) This option is useful for instantiating a *`Formatter`* subclass. Subclasses of *`Formatter`* can present exception tracebacks in an expanded or condensed format.

注解: Due to the use of *`eval()`* as described above, there are potential security risks which result from using the *`listen()`* to send and receive configurations via sockets. The risks are limited to where multiple users with no mutual trust run code on the same machine; see the *`listen()`* documentation for more information.

参见:

模块*`logging`* 日志记录模块的 API 参考。

模块*`logging.handlers`* 日志记录模块附带的有用处理程序。

15.9 `logging.handlers` —日志处理

Important

此页面仅包含参考信息。有关教程，请参阅

- 基础教程
- 进阶教程
- Logging Cookbook

源代码: Lib/logging/handlers.py

The following useful handlers are provided in the package. Note that three of the handlers (*StreamHandler*, *FileHandler* and *NullHandler*) are actually defined in the *logging* module itself, but have been documented here along with the other handlers.

15.9.1 StreamHandler

The *StreamHandler* class, located in the core *logging* package, sends logging output to streams such as *sys.stdout*, *sys.stderr* or any file-like object (or, more precisely, any object which supports `write()` and `flush()` methods).

class logging.**StreamHandler**(*stream=None*)

Returns a new instance of the *StreamHandler* class. If *stream* is specified, the instance will use it for logging output; otherwise, *sys.stderr* will be used.

emit(*record*)

If a formatter is specified, it is used to format the record. The record is then written to the stream with a newline terminator. If exception information is present, it is formatted using *traceback.print_exception()* and appended to the stream.

flush()

Flushes the stream by calling its *flush()* method. Note that the `close()` method is inherited from `Handler` and so does no output, so an explicit *flush()* call may be needed at times.

15.9.2 FileHandler

The *FileHandler* class, located in the core *logging* package, sends logging output to a disk file. It inherits the output functionality from *StreamHandler*.

class logging.**FileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*)

Returns a new instance of the *FileHandler* class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to *emit()*. By default, the file grows indefinitely.

在 2.6 版更改: *delay* was added.

close()

关闭文件。

emit(*record*)

将记录输出到文件。

15.9.3 NullHandler

2.7 新版功能.

The *NullHandler* class, located in the core *logging* package, does not do any formatting or output. It is essentially a 'no-op' handler for use by library developers.

class logging.**NullHandler**

Returns a new instance of the *NullHandler* class.

emit(*record*)

This method does nothing.

handle(*record*)

This method does nothing.

createLock()
This method returns None for the lock, since there is no underlying I/O to which access needs to be serialized.

See library-config for more information on how to use *NullHandler*.

15.9.4 WatchedFileHandler

2.6 新版功能.

The *WatchedFileHandler* class, located in the *logging.handlers* module, is a FileHandler which watches the file it is logging to. If the file changes, it is closed and reopened using the file name.

A file change can happen because of usage of programs such as *newsyslog* and *logrotate* which perform log file rotation. This handler, intended for use under Unix/Linux, watches the file to see if it has changed since the last emit. (A file is deemed to have changed if its device or inode have changed.) If the file has changed, the old file stream is closed, and the file opened to get a new stream.

This handler is not appropriate for use under Windows, because under Windows open log files cannot be moved or renamed - logging opens the files with exclusive locks - and so there is no need for such a handler. Furthermore, *ST_INO* is not supported under Windows; *stat()* always returns zero for this value.

class logging.handlers.**WatchedFileHandler**(*filename*[, *mode*[, *encoding*[, *delay*]]])
Returns a new instance of the *WatchedFileHandler* class. The specified file is opened and used as the stream for logging. If *mode* is not specified, 'a' is used. If *encoding* is not None, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to *emit()*. By default, the file grows indefinitely.

emit(*record*)
Outputs the record to the file, but first checks to see if the file has changed. If it has, the existing stream is flushed and closed and the file opened again, before outputting the record to the file.

15.9.5 RotatingFileHandler

The *RotatingFileHandler* class, located in the *logging.handlers* module, supports rotation of disk log files.

class logging.handlers.**RotatingFileHandler**(*filename*, *mode='a'*, *maxBytes=0*, *backupCount=0*, *encoding=None*, *delay=0*)
Returns a new instance of the *RotatingFileHandler* class. The specified file is opened and used as the stream for logging. If *mode* is not specified, 'a' is used. If *encoding* is not None, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to *emit()*. By default, the file grows indefinitely.

You can use the *maxBytes* and *backupCount* values to allow the file to *rollover* at a predetermined size. When the size is about to be exceeded, the file is closed and a new file is silently opened for output. Rollover occurs whenever the current log file is nearly *maxBytes* in length; if either of *maxBytes* or *backupCount* is zero, rollover never occurs. If *backupCount* is non-zero, the system will save old log files by appending the extensions '.1', '.2' etc., to the filename. For example, with a *backupCount* of 5 and a base file name of app.log, you would get app.log, app.log.1, app.log.2, up to app.log.5. The file being written to is always app.log. When this file is filled, it is closed and renamed to app.log.1, and if files app.log.1, app.log.2, etc. exist, then they are renamed to app.log.2, app.log.3 etc. respectively.

在 2.6 版更改: *delay* was added.

doRollover()
Does a rollover, as described above.

emit(*record*)
Outputs the record to the file, catering for rollover as described previously.

15.9.6 TimedRotatingFileHandler

The *TimedRotatingFileHandler* class, located in the *logging.handlers* module, supports rotation of disk log files at certain timed intervals.

class logging.handlers.**TimedRotatingFileHandler**(*filename*, *when='h'*, *interval=1*, *backupCount=0*, *encoding=None*, *delay=False*, *utc=False*)
Returns a new instance of the *TimedRotatingFileHandler* class. The specified file is opened and used as the stream for logging. On rotating it also sets the filename suffix. Rotating happens based on the product of *when* and *interval*.

You can use the *when* to specify the type of *interval*. The list of possible values is below. Note that they are not case sensitive.

值	间隔类型
'S'	秒
'M'	分钟
'H'	小时
'D'	天
'W0'-'W6'	工作日 (0= 星期一)
'midnight'	Roll over at midnight

When using weekday-based rotation, specify ‘W0’ for Monday, ‘W1’ for Tuesday, and so on up to ‘W6’ for Sunday. In this case, the value passed for *interval* isn’t used.

The system will save old log files by appending extensions to the filename. The extensions are date-and-time based, using the strftime format `%Y-%m-%d_%H-%M-%S` or a leading portion thereof, depending on the rollover interval.

When computing the next rollover time for the first time (when the handler is created), the last modification time of an existing log file, or else the current time, is used to compute when the next rotation will occur.

If the *utc* argument is true, times in UTC will be used; otherwise local time is used.

If *backupCount* is nonzero, at most *backupCount* files will be kept, and if more would be created when rollover occurs, the oldest one is deleted. The deletion logic uses the interval to determine which files to delete, so changing the interval may leave old files lying around.

If *delay* is true, then file opening is deferred until the first call to *emit()*.

在 2.6 版更改: *delay* and *utc* were added.

doRollover()
Does a rollover, as described above.

emit(*record*)
Outputs the record to the file, catering for rollover as described above.

15.9.7 SocketHandler

The *SocketHandler* class, located in the *logging.handlers* module, sends logging output to a network socket. The base class uses a TCP socket.

class logging.handlers.**SocketHandler**(*host*, *port*)

Returns a new instance of the *SocketHandler* class intended to communicate with a remote machine whose address is given by *host* and *port*.

close()

Closes the socket.

emit()

Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. If the connection was previously lost, re-establishes the connection. To unpickle the record at the receiving end into a *LogRecord*, use the *makeLogRecord()* function.

handleError()

Handles an error which has occurred during *emit()*. The most likely cause is a lost connection. Closes the socket so that we can retry on the next event.

makeSocket()

This is a factory method which allows subclasses to define the precise type of socket they want. The default implementation creates a TCP socket (*socket.SOCK_STREAM*).

makePickle(*record*)

Pickles the record's attribute dictionary in binary format with a length prefix, and returns it ready for transmission across the socket.

Note that pickles aren't completely secure. If you are concerned about security, you may want to override this method to implement a more secure mechanism. For example, you can sign pickles using HMAC and then verify them on the receiving end, or alternatively you can disable unpickling of global objects on the receiving end.

send(*packet*)

Send a pickled string *packet* to the socket. This function allows for partial sends which can happen when the network is busy.

createSocket()

Tries to create a socket; on failure, uses an exponential back-off algorithm. On initial failure, the handler will drop the message it was trying to send. When subsequent messages are handled by the same instance, it will not try connecting until some time has passed. The default parameters are such that the initial delay is one second, and if after that delay the connection still can't be made, the handler will double the delay each time up to a maximum of 30 seconds.

This behaviour is controlled by the following handler attributes:

- `retryStart` (initial delay, defaulting to 1.0 seconds).
- `retryFactor` (multiplier, defaulting to 2.0).
- `retryMax` (maximum delay, defaulting to 30.0 seconds).

This means that if the remote listener starts up *after* the handler has been used, you could lose messages (since the handler won't even attempt a connection until the delay has elapsed, but just silently drop messages during the delay period).

15.9.8 DatagramHandler

The *DatagramHandler* class, located in the *logging.handlers* module, inherits from *SocketHandler* to support sending logging messages over UDP sockets.

class logging.handlers.**DatagramHandler**(*host*, *port*)
Returns a new instance of the *DatagramHandler* class intended to communicate with a remote machine whose address is given by *host* and *port*.

emit()
Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. To unpickle the record at the receiving end into a *LogRecord*, use the *makeLogRecord()* function.

makeSocket()
The factory method of *SocketHandler* is here overridden to create a UDP socket (*socket.SOCK_DGRAM*).

send(*s*)
Send a pickled string to a socket.

15.9.9 SysLogHandler

The *SysLogHandler* class, located in the *logging.handlers* module, supports sending logging messages to a remote or local Unix syslog.

class logging.handlers.**SysLogHandler**(*address=('localhost', SYSLOG_UDP_PORT)*, *facility=LOG_USER*, *socktype=socket.SOCK_DGRAM*)
Returns a new instance of the *SysLogHandler* class intended to communicate with a remote Unix machine whose address is given by *address* in the form of a `(host, port)` tuple. If *address* is not specified, `('localhost', 514)` is used. The address is used to open a socket. An alternative to providing a `(host, port)` tuple is providing an address as a string, for example '/dev/log'. In this case, a Unix domain socket is used to send the message to the syslog. If *facility* is not specified, `LOG_USER` is used. The type of socket opened depends on the *socktype* argument, which defaults to *socket.SOCK_DGRAM* and thus opens a UDP socket. To open a TCP socket (for use with the newer syslog daemons such as rsyslog), specify a value of *socket.SOCK_STREAM*.

Note that if your server is not listening on UDP port 514, *SysLogHandler* may appear not to work. In that case, check what address you should be using for a domain socket - it's system dependent. For example, on Linux it's usually '/dev/log' but on OS/X it's '/var/run/syslog'. You'll need to check your platform and use the appropriate address (you may need to do this check at runtime if your application needs to run on several platforms). On Windows, you pretty much have to use the UDP option.

在 2.7 版更改: *socktype* was added.

close()
Closes the socket to the remote host.

emit(*record*)
The record is formatted, and then sent to the syslog server. If exception information is present, it is *not* sent to the server.

encodePriority(*facility*, *priority*)
Encodes the facility and priority into an integer. You can pass in strings or integers - if strings are passed, internal mapping dictionaries are used to convert them to integers.

The symbolic `LOG_` values are defined in *SysLogHandler* and mirror the values defined in the `sys/syslog.h` header file.

优先级

名称（字符串）	符号值
alert	LOG_ALERT
crit 或 critical	LOG_CRIT
debug	LOG_DEBUG
emerg 或 panic	LOG_EMERG
err 或 error	LOG_ERR
info	LOG_INFO
notice	LOG_NOTICE
warn 或 warning	LOG_WARNING

设备

名称（字符串）	符号值
auth	LOG_AUTH
authpriv	LOG_AUTHPRIV
cron	LOG_CRON
daemon	LOG_DAEMON
ftp	LOG_FTP
kern	LOG_KERN
lpr	LOG_LPR
mail	LOG_MAIL
news	LOG_NEWS
syslog	LOG_SYSLOG
user	LOG_USER
uucp	LOG_UUCP
local0	LOG_LOCAL0
local1	LOG_LOCAL1
local2	LOG_LOCAL2
local3	LOG_LOCAL3
local4	LOG_LOCAL4
local5	LOG_LOCAL5
local6	LOG_LOCAL6
local7	LOG_LOCAL7

mapPriority(*levelname*)
Maps a logging level name to a syslog priority name. You may need to override this if you are using custom levels, or if the default algorithm is not suitable for your needs. The default algorithm maps DEBUG, INFO, WARNING, ERROR and CRITICAL to the equivalent syslog names, and all other level names to 'warning'.

15.9.10 NTEventLogHandler

The *NTEventLogHandler* class, located in the *logging.handlers* module, supports sending logging messages to a local Windows NT, Windows 2000 or Windows XP event log. Before you can use it, you need Mark Hammond's Win32 extensions for Python installed.

class logging.handlers.**NTEventLogHandler**(*appname*, *dllname=None*, *logtype='Application'*)
Returns a new instance of the *NTEventLogHandler* class. The *appname* is used to define the application name as it appears in the event log. An appropriate registry entry is created using this name. The *dllname* should give the fully qualified pathname of a .dll or .exe which contains message definitions to hold in the log (if not specified, 'win32service.pyd' is used - this is installed with the Win32 extensions and contains some basic placeholder

message definitions. Note that use of these placeholders will make your event logs big, as the entire message source is held in the log. If you want slimmer logs, you have to pass in the name of your own .dll or .exe which contains the message definitions you want to use in the event log). The *logtype* is one of `'Application'`, `'System'` or `'Security'`, and defaults to `'Application'`.

close()
At this point, you can remove the application name from the registry as a source of event log entries. However, if you do this, you will not be able to see the events as you intended in the Event Log Viewer - it needs to be able to access the registry to get the .dll name. The current version does not do this.

emit(*record*)
Determines the message ID, event category and event type, and then logs the message in the NT event log.

getEventCategory(*record*)
Returns the event category for the record. Override this if you want to specify your own categories. This version returns 0.

getEventType(*record*)
Returns the event type for the record. Override this if you want to specify your own types. This version does a mapping using the handler's typemap attribute, which is set up in `__init__()` to a dictionary which contains mappings for `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL`. If you are using your own levels, you will either need to override this method or place a suitable dictionary in the handler's *typemap* attribute.

getMessageID(*record*)
Returns the message ID for the record. If you are using your own messages, you could do this by having the *msg* passed to the logger being an ID rather than a format string. Then, in here, you could use a dictionary lookup to get the message ID. This version returns 1, which is the base message ID in `win32service.pyd`.

15.9.11 SMTPHandler

The *SMTPHandler* class, located in the *logging.handlers* module, supports sending logging messages to an email address via SMTP.

class logging.handlers.**SMTPHandler**(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*)
Returns a new instance of the *SMTPHandler* class. The instance is initialized with the from and to addresses and subject line of the email. The *toaddrs* should be a list of strings. To specify a non-standard SMTP port, use the (host, port) tuple format for the *mailhost* argument. If you use a string, the standard SMTP port is used. If your SMTP server requires authentication, you can specify a (username, password) tuple for the *credentials* argument.

To specify the use of a secure protocol (TLS), pass in a tuple to the *secure* argument. This will only be used when authentication credentials are supplied. The tuple should be either an empty tuple, or a single-value tuple with the name of a keyfile, or a 2-value tuple with the names of the keyfile and certificate file. (This tuple is passed to the *smtplib.SMTP.starttls()* method.)

在 2.6 版更改: *credentials* was added.

在 2.7 版更改: *secure* was added.

emit(*record*)
Formats the record and sends it to the specified addressees.

getSubject(*record*)
If you want to specify a subject line which is record-dependent, override this method.

15.9.12 MemoryHandler

The *MemoryHandler* class, located in the *logging.handlers* module, supports buffering of logging records in memory, periodically flushing them to a *target* handler. Flushing occurs whenever the buffer is full, or when an event of a certain severity or greater is seen.

MemoryHandler is a subclass of the more general *BufferingHandler*, which is an abstract class. This buffers logging records in memory. Whenever each record is added to the buffer, a check is made by calling `shouldFlush()` to see if the buffer should be flushed. If it should, then `flush()` is expected to do the flushing.

class logging.handlers.**BufferingHandler**(*capacity*)
Initializes the handler with a buffer of the specified capacity.

emit(*record*)
Appends the record to the buffer. If *shouldFlush()* returns true, calls *flush()* to process the buffer.

flush()
You can override this to implement custom flushing behavior. This version just zaps the buffer to empty.

shouldFlush(*record*)
Returns true if the buffer is up to capacity. This method can be overridden to implement custom flushing strategies.

class logging.handlers.**MemoryHandler**(*capacity*, *flushLevel=ERROR*, *target=None*)
Returns a new instance of the *MemoryHandler* class. The instance is initialized with a buffer size of *capacity*. If *flushLevel* is not specified, `ERROR` is used. If no *target* is specified, the target will need to be set using *setTarget()* before this handler does anything useful.

close()
Calls *flush()*, sets the target to *None* and clears the buffer.

flush()
For a *MemoryHandler*, flushing means just sending the buffered records to the target, if there is one. The buffer is also cleared when this happens. Override if you want different behavior.

setTarget(*target*)
Sets the target handler for this handler.

shouldFlush(*record*)
Checks for buffer full or a record at the *flushLevel* or higher.

15.9.13 HTTPHandler

The *HTTPHandler* class, located in the *logging.handlers* module, supports sending logging messages to a Web server, using either `GET` or `POST` semantics.

class logging.handlers.**HTTPHandler**(*host*, *url*, *method='GET'*)
Returns a new instance of the *HTTPHandler* class. The `host` can be of the form `host:port`, should you need to use a specific port number.

mapLogRecord(*record*)
Provides a dictionary, based on `record`, which is to be URL-encoded and sent to the web server. The default implementation just returns `record.__dict__`. This method can be overridden if e.g. only a subset of *LogRecord* is to be sent to the web server, or if more specific customization of what's sent to the server is required.

emit(*record*)
Sends the record to the Web server as a URL-encoded dictionary. The *mapLogRecord()* method is used to convert the record to the dictionary to be sent.

注解: Since preparing a record for sending it to a Web server is not the same as a generic formatting operation, using `setFormatter()` to specify a `Formatter` for a *HTTPHandler* has no effect. Instead of calling *format()*, this handler calls *mapLogRecord()* and then *urllib.urlencode()* to encode the dictionary in a form suitable for sending to a Web server.

参见:

模块*logging* 日志记录模块的 API 参考。

模块*logging.config* 日志记录模块的配置 API 。

15.10 getpass —便携式密码输入工具

getpass 模块提供了两个函数:

getpass.**getpass**([*prompt*[, *stream*]])
Prompt the user for a password without echoing. The user is prompted using the string *prompt*, which defaults to `'Password: '`. On Unix, the prompt is written to the file-like object *stream*. *stream* defaults to the controlling terminal (/dev/tty) or if that is unavailable to `sys.stderr` (this argument is ignored on Windows).

如果回显自由输入不可用则 getpass() 将回退为打印一条警告消息到 *stream* 并且从 `sys.stdin` 读取同时发出*GetPassWarning*。

在 2.5 版更改: The *stream* parameter was added.

在 2.6 版更改: On Unix it defaults to using /dev/tty before falling back to `sys.stdin` and `sys.stderr`.

注解: 如果你从 IDLE 内部调用 getpass，输入可能是在你启动 IDLE 的终端中而非在 IDEL 窗口本身中完成。

exception getpass.**GetPassWarning**
一个当密码输入可能被回显时发出的*UserWarning* 子类。

getpass.**getuser**()
返回用户的“登录名称”。

此函数会按顺序检查环境变量 `LOGNAME`, `USER`, `LNAME` 和 `USERNAME`，并返回其中第一个被设置为非空字符串的值。如果均未设置，则在支持*pwd* 模块的系统上将返回来自密码数据库的登录名，否则将引发一个异常。

15.11 curses —终端字符单元显示的处理

在 1.6 版更改: Added support for the `ncurses` library and converted to a package.

curses 模块提供了 curses 库的接口，这是可移植高级终端处理的事实标准。

While curses is most widely used in the Unix environment, versions are available for DOS, OS/2, and possibly other systems as well. This extension module is designed to match the API of ncurses, an open-source curses library hosted on Linux and the BSD variants of Unix.

注解: 从 5.4 版本开始，ncurses 库使用 `nl_langinfo` 函数来决定如何解释非 ASCII 数据。这意味着你需要在程序中调用*locale.setlocale()* 函数，并使用一种系统中可用的编码方法来编码 Unicode 字符串。这个例子使用了系统默认的编码:

```
import locale
locale.setlocale(locale.LC_ALL, '')
code = locale.getpreferredencoding()
```

然后使用 *code* 作为*str.encode()* 调用的编码。

参见:

模块*curses.ascii* 在 ASCII 字符上工作的工具，无论你的区域设置是什么。

模块*curses.panel* A panel stack extension that adds depth to curses windows.

Module *curses.textpad* Editable text widget for curses supporting **Emacs**-like bindings.

curses-howto Tutorial material on using curses with Python, by Andrew Kuchling and Eric Raymond.

The Demo/curses/ directory in the Python source distribution contains some example programs using the curses bindings provided by this module.

15.11.1 函数

curses 模块定义了以下异常:

exception curses.**error**
当 curses 库中函数返回一个错误时引发的异常。

注解: Whenever *x* or *y* arguments to a function or a method are optional, they default to the current cursor location. Whenever *attr* is optional, it defaults to `A_NORMAL`.

curses 模块定义了以下函数:

curses.**baudrate**()
Return the output speed of the terminal in bits per second. On software terminal emulators it will have a fixed high value. Included for historical reasons; in former times, it was used to write output loops for time delays and occasionally to change interfaces depending on the line speed.

curses.**beep**()
Emit a short attention sound.

curses.**can_change_color**()
Return `True` or `False`, depending on whether the programmer can change the colors displayed by the terminal.

curses.**cbreak**()
Enter cbreak mode. In cbreak mode (sometimes called "rare" mode) normal tty line buffering is turned off and characters are available to be read one by one. However, unlike raw mode, special characters (interrupt, quit, suspend, and flow control) retain their effects on the tty driver and calling program. Calling first *raw()* then *cbreak()* leaves the terminal in cbreak mode.

curses.**color_content**(*color_number*)
Return the intensity of the red, green, and blue (RGB) components in the color *color_number*, which must be between `0` and `COLORS`. A 3-tuple is returned, containing the R,G,B values for the given color, which will be between `0` (no component) and `1000` (maximum amount of component).

curses.**color_pair**(*color_number*)
Return the attribute value for displaying text in the specified color. This attribute value can be combined with A_STANDOUT, A_REVERSE, and the other A_* attributes. *pair_number()* is the counterpart to this function.

curses.**curs_set**(*visibility*)
Set the cursor state. *visibility* can be set to 0, 1, or 2, for invisible, normal, or very visible. If the terminal supports the visibility requested, the previous cursor state is returned; otherwise, an exception is raised. On many terminals, the "visible" mode is an underline cursor and the "very visible" mode is a block cursor.

curses.**def_prog_mode**()
Save the current terminal mode as the "program" mode, the mode when the running program is using curses. (Its counterpart is the "shell" mode, for when the program is not in curses.) Subsequent calls to *reset_prog_mode()* will restore this mode.

curses.**def_shell_mode**()
Save the current terminal mode as the "shell" mode, the mode when the running program is not using curses. (Its counterpart is the "program" mode, when the program is using curses capabilities.) Subsequent calls to *reset_shell_mode()* will restore this mode.

curses.**delay_output**(*ms*)
Insert an *ms* millisecond pause in output.

curses.**doupdate**()
Update the physical screen. The curses library keeps two data structures, one representing the current physical screen contents and a virtual screen representing the desired next state. The *doupdate()* ground updates the physical screen to match the virtual screen.

The virtual screen may be updated by a noutrefresh() call after write operations such as addstr() have been performed on a window. The normal refresh() call is simply noutrefresh() followed by *doupdate()*; if you have to update multiple windows, you can speed performance and perhaps reduce screen flicker by issuing noutrefresh() calls on all windows, followed by a single *doupdate()*.

curses.**echo**()
Enter echo mode. In echo mode, each character input is echoed to the screen as it is entered.

curses.**endwin**()
De-initialize the library, and return terminal to normal status.

curses.**erasechar**()
Return the user's current erase character. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.**filter**()
The *filter()* routine, if used, must be called before *initscr()* is called. The effect is that, during those calls, LINES is set to 1; the capabilities clear, cup, cud, cud1, cuu1, cuu, vpa are disabled; and the home string is set to the value of cr. The effect is that the cursor is confined to the current line, and so are screen updates. This may be used for enabling character-at-a-time line editing without touching the rest of the screen.

curses.**flash**()
Flash the screen. That is, change it to reverse-video and then change it back in a short interval. Some people prefer such as 'visible bell' to the audible attention signal produced by *beep()*.

curses.**flushinp**()
Flush all input buffers. This throws away any typeahead that has been typed by the user and has not yet been processed by the program.

curses.**getmouse**()
After getch() returns KEY_MOUSE to signal a mouse event, this method should be call to retrieve the queued mouse event, represented as a 5-tuple (id, x, y, z, bstate). *id* is an ID value used to distinguish multiple devices, and *x*, *y*, *z* are the event's coordinates. (*z* is currently unused.) *bstate* is an integer value

whose bits will be set to indicate the type of event, and will be the bitwise OR of one or more of the following constants, where *n* is the button number from 1 to 4: `BUTTONn_PRESSED`, `BUTTONn_RELEASED`, `BUTTONn_CLICKED`, `BUTTONn_DOUBLE_CLICKED`, `BUTTONn_TRIPLE_CLICKED`, `BUTTON_SHIFT`, `BUTTON_CTRL`, `BUTTON_ALT`.

`curses.`**`getsyx`**`()`
Return the current coordinates of the virtual screen cursor in y and x. If leaveok is currently true, then -1,-1 is returned.

`curses.`**`getwin`**`(`*file*`)`
Read window related data stored in the file by an earlier `putwin()` call. The routine then creates and initializes a new window using that data, returning the new window object.

`curses.`**`has_colors`**`()`
Return `True` if the terminal can display colors; otherwise, return `False`.

`curses.`**`has_ic`**`()`
Return `True` if the terminal has insert- and delete-character capabilities. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

`curses.`**`has_il`**`()`
Return `True` if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

`curses.`**`has_key`**`(`*ch*`)`
Take a key value *ch*, and return `True` if the current terminal type recognizes a key with that value.

`curses.`**`halfdelay`**`(`*tenths*`)`
Used for half-delay mode, which is similar to cbreak mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, an exception is raised if nothing has been typed. The value of *tenths* must be a number between `1` and `255`. Use *`nocbreak()`* to leave half-delay mode.

`curses.`**`init_color`**`(`*color_number*, *r*, *g*, *b*`)`
Change the definition of a color, taking the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of *color_number* must be between `0` and `COLORS`. Each of *r*, *g*, *b*, must be a value between `0` and `1000`. When *`init_color()`* is used, all occurrences of that color on the screen immediately change to the new definition. This function is a no-op on most terminals; it is active only if *`can_change_color()`* returns `1`.

`curses.`**`init_pair`**`(`*pair_number*, *fg*, *bg*`)`
Change the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of *pair_number* must be between `1` and `COLOR_PAIRS - 1` (the `0` color pair is wired to white on black and cannot be changed). The value of *fg* and *bg* arguments must be between `0` and `COLORS`. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition.

`curses.`**`initscr`**`()`
Initialize the library. Return a `WindowObject` which represents the whole screen.

注解： If there is an error opening the terminal, the underlying curses library may cause the interpreter to exit.

`curses.`**`is_term_resized`**`(`*nlines*, *ncols*`)`
Return `True` if *`resize_term()`* would modify the window structure, `False` otherwise.

`curses.`**`isendwin`**`()`
Return `True` if *`endwin()`* has been called (that is, the curses library has been deinitialized).

curses.**keyname**(*k*)
Return the name of the key numbered *k*. The name of a key generating printable ASCII character is the key's character. The name of a control-key combination is a two-character string consisting of a caret followed by the corresponding printable ASCII character. The name of an alt-key combination (128–255) is a string consisting of the prefix 'M-' followed by the name of the corresponding ASCII character.

curses.**killchar**()
Return the user's current line kill character. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.**longname**()
Return a string containing the terminfo long name field describing the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to *initscr()*.

curses.**meta**(*yes*)
If *yes* is 1, allow 8-bit characters to be input. If *yes* is 0, allow only 7-bit chars.

curses.**mouseinterval**(*interval*)
Set the maximum time in milliseconds that can elapse between press and release events in order for them to be recognized as a click, and return the previous interval value. The default value is 200 msec, or one fifth of a second.

curses.**mousemask**(*mousemask*)
Set the mouse events to be reported, and return a tuple `(availmask, oldmask)`. *availmask* indicates which of the specified mouse events can be reported; on complete failure it returns 0. *oldmask* is the previous value of the given window's mouse event mask. If this function is never called, no mouse events are ever reported.

curses.**napms**(*ms*)
Sleep for *ms* milliseconds.

curses.**newpad**(*nlines*, *ncols*)
Create and return a pointer to a new pad data structure with the given number of lines and columns. A pad is returned as a window object.

A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (such as from scrolling or echoing of input) do not occur. The `refresh()` and `noutrefresh()` methods of a pad require 6 arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display. The arguments are *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*; the *p* arguments refer to the upper left corner of the pad region to be displayed and the *s* arguments define a clipping box on the screen within which the pad region is to be displayed.

curses.**newwin**(*nlines*, *ncols*)
curses.**newwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)
Return a new window, whose left-upper corner is at `(begin_y, begin_x)`, and whose height/width is *nlines*/*ncols*.

By default, the window will extend from the specified position to the lower right corner of the screen.

curses.**nl**()
Enter newline mode. This mode translates the return key into newline on input, and translates newline into return and line-feed on output. Newline mode is initially on.

curses.**nocbreak**()
Leave cbreak mode. Return to normal "cooked" mode with line buffering.

curses.**noecho**()
Leave echo mode. Echoing of input characters is turned off.

curses.**nonl**()
Leave newline mode. Disable translation of return into newline on input, and disable low-level translation of newline into newline/return on output (but this does not change the behavior of `addch('\n')`, which always does the

equivalent of return and line feed on the virtual screen). With translation off, curses can sometimes speed up vertical motion a little; also, it will be able to detect the return key on input.

curses.**noqiflush**()
When the *noqiflush()* routine is used, normal flush of input and output queues associated with the INTR, QUIT and SUSP characters will not be done. You may want to call *noqiflush()* in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.

curses.**noraw**()
Leave raw mode. Return to normal "cooked" mode with line buffering.

curses.**pair_content**(*pair_number*)
Return a tuple `(fg, bg)` containing the colors for the requested color pair. The value of *pair_number* must be between `1` and `COLOR_PAIRS - 1`.

curses.**pair_number**(*attr*)
Return the number of the color-pair set by the attribute value *attr*. *color_pair()* is the counterpart to this function.

curses.**putp**(*string*)
Equivalent to `tputs(str, 1, putchar)`; emit the value of a specified terminfo capability for the current terminal. Note that the output of *putp()* always goes to standard output.

curses.**qiflush**([*flag*])
If *flag* is `False`, the effect is the same as calling *noqiflush()*. If *flag* is `True`, or no argument is provided, the queues will be flushed when these control characters are read.

curses.**raw**()
Enter raw mode. In raw mode, normal line buffering and processing of interrupt, quit, suspend, and flow control keys are turned off; characters are presented to curses input functions one by one.

curses.**reset_prog_mode**()
Restore the terminal to "program" mode, as previously saved by *def_prog_mode()*.

curses.**reset_shell_mode**()
Restore the terminal to "shell" mode, as previously saved by *def_shell_mode()*.

curses.**resetty**()
Restore the state of the terminal modes to what it was at the last call to *savetty()*.

curses.**resize_term**(*nlines*, *ncols*)
Backend function used by *resizeterm()*, performing most of the work; when resizing the windows, *resize_term()* blank-fills the areas that are extended. The calling application should fill in these areas with appropriate data. The *resize_term()* function attempts to resize all windows. However, due to the calling convention of pads, it is not possible to resize these without additional interaction with the application.

curses.**resizeterm**(*nlines*, *ncols*)
Resize the standard and current windows to the specified dimensions, and adjusts other bookkeeping data used by the curses library that record the window dimensions (in particular the SIGWINCH handler).

curses.**savetty**()
Save the current state of the terminal modes in a buffer, usable by *resetty()*.

curses.**setsyx**(*y*, *x*)
Set the virtual screen cursor to *y*, *x*. If *y* and *x* are both -1, then leaveok is set.

curses.**setupterm**([*termstr*, *fd*])
Initialize the terminal. *termstr* is a string giving the terminal name; if omitted, the value of the `TERM` environment variable will be used. *fd* is the file descriptor to which any initialization sequences will be sent; if not supplied, the file descriptor for `sys.stdout` will be used.

curses.**start_color**()
Must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after *initscr()*.

start_color() initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables in the *curses* module, COLORS and COLOR_PAIRS, containing the maximum number of colors and color-pairs the terminal can support. It also restores the colors on the terminal to the values they had when the terminal was just turned on.

curses.**termattrs**()
Return a logical OR of all video attributes supported by the terminal. This information is useful when a curses program needs complete control over the appearance of the screen.

curses.**termname**()
Return the value of the environment variable TERM, truncated to 14 characters.

curses.**tigetflag**(*capname*)
Return the value of the Boolean capability corresponding to the terminfo capability name *capname*. The value -1 is returned if *capname* is not a Boolean capability, or 0 if it is canceled or absent from the terminal description.

curses.**tigetnum**(*capname*)
Return the value of the numeric capability corresponding to the terminfo capability name *capname*. The value -2 is returned if *capname* is not a numeric capability, or -1 if it is canceled or absent from the terminal description.

curses.**tigetstr**(*capname*)
Return the value of the string capability corresponding to the terminfo capability name *capname*. None is returned if *capname* is not a string capability, or is canceled or absent from the terminal description.

curses.**tparm**(*str*[, ...])
Instantiate the string *str* with the supplied parameters, where *str* should be a parameterized string obtained from the terminfo database. E.g. `tparm(tigetstr("cup"), 5, 3)` could result in `'\033[6;4H'`, the exact result depending on terminal type.

curses.**typeahead**(*fd*)
Specify that the file descriptor *fd* be used for typeahead checking. If *fd* is -1, then no typeahead checking is done.

The curses library does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until refresh or doupdate is called again, allowing faster response to commands typed in advance. This function allows specifying a different file descriptor for typeahead checking.

curses.**unctrl**(*ch*)
Return a string which is a printable representation of the character *ch*. Control characters are displayed as a caret followed by the character, for example as ^C. Printing characters are left as they are.

curses.**ungetch**(*ch*)
Push *ch* so the next getch() will return it.

注解: Only one *ch* can be pushed before getch() is called.

curses.**ungetmouse**(*id*, *x*, *y*, *z*, *bstate*)
Push a KEY_MOUSE event onto the input queue, associating the given state data with it.

curses.**use_env**(*flag*)
If used, this function should be called before *initscr()* or newterm are called. When *flag* is False, the values of lines and columns specified in the terminfo database will be used, even if environment variables LINES and COLUMNS (used by default) are set, or if curses is running in a window (in which case default behavior would be to use the window size if LINES and COLUMNS are not set).

curses.**use_default_colors**()
Allow use of default values for colors on terminals supporting this feature. Use this to support transparency in your application. The default color is assigned to the color number -1. After calling this function, `init_pair(x, curses.COLOR_RED, -1)` initializes, for instance, color pair *x* to a red foreground color on the default background.

curses.**wrapper**(*func*, ...)
Initialize curses and call another callable object, *func*, which should be the rest of your curses-using application. If the application raises an exception, this function will restore the terminal to a sane state before re-raising the exception and generating a traceback. The callable object *func* is then passed the main window 'stdscr' as its first argument, followed by any other arguments passed to *wrapper()*. Before calling *func*, *wrapper()* turns on cbreak mode, turns off echo, enables the terminal keypad, and initializes colors if the terminal has color support. On exit (whether normally or by exception) it restores cooked mode, turns on echo, and disables the terminal keypad.

15.11.2 Window Objects

Window objects, as returned by *initscr()* and *newwin()* above, have the following methods:

window.**addch**(*ch*[, *attr*])
window.**addch**(*y*, *x*, *ch*[, *attr*])

注解: A *character* means a C character (an ASCII code), rather than a Python character (a string of length 1). (This note is true whenever the documentation mentions a character.) The built-in *ord()* is handy for conveying strings to codes.

Paint character *ch* at `(y, x)` with attributes *attr*, overwriting any character previously painter at that location. By default, the character position and attributes are the current settings for the window object.

注解: Writing outside the window, subwindow, or pad raises a *curses.error*. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the character is printed.

---

window.**addnstr**(*str*, *n*[, *attr*])
window.**addnstr**(*y*, *x*, *str*, *n*[, *attr*])
Paint at most *n* characters of the string *str* at `(y, x)` with attributes *attr*, overwriting anything previously on the display.

window.**addstr**(*str*[, *attr*])
window.**addstr**(*y*, *x*, *str*[, *attr*])
Paint the string *str* at `(y, x)` with attributes *attr*, overwriting anything previously on the display.

注解: Writing outside the window, subwindow, or pad raises *curses.error*. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the string is printed.

---

window.**attroff**(*attr*)
Remove attribute *attr* from the "background" set applied to all writes to the current window.

window.**attron**(*attr*)
Add attribute *attr* from the "background" set applied to all writes to the current window.

window.**attrset**(*attr*)
Set the "background" set of attributes to *attr*. This set is initially 0 (no attributes).

window.**bkgd**(*ch*[, *attr*])
Set the background property of the window to the character *ch*, with attributes *attr*. The change is then applied to every character position in that window:

- The attribute of every character in the window is changed to the new background attribute.
- Wherever the former background character appears, it is changed to the new background character.

window.**bkgdset**(*ch*[, *attr*])
Set the window's background. A window's background consists of a character and any combination of attributes. The attribute part of the background is combined (OR'ed) with all non-blank characters that are written into the window. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations.

window.**border**([*ls*[, *rs*[, *ts*[, *bs*[, *tl*[, *tr*[, *bl*[, *br*]]]]]]]])
Draw a border around the edges of the window. Each parameter specifies the character to use for a specific part of the border; see the table below for more details. The characters can be specified as integers or as one-character strings.

注解: A `0` value for any parameter will cause the default character to be used for that parameter. Keyword parameters can *not* be used. The defaults are listed in this table:

参数	描述	默认值
ls	左侧	ACS_VLINE
rs	右侧	ACS_VLINE
ts	顶部	ACS_HLINE
bs	底部	ACS_HLINE
tl	左上角	ACS_ULCORNER
tr	右上角	ACS_URCORNER
bl	左下角	ACS_LLCORNER
br	右下角	ACS_LRCORNER

window.**box**([*vertch*, *horch*])
Similar to *border()*, but both *ls* and *rs* are *vertch* and both *ts* and *bs* are *horch*. The default corner characters are always used by this function.

window.**chgat**(*attr*)
window.**chgat**(*num*, *attr*)
window.**chgat**(*y*, *x*, *attr*)
window.**chgat**(*y*, *x*, *num*, *attr*)
Set the attributes of *num* characters at the current cursor position, or at position `(y, x)` if supplied. If *num* is not given or is `-1`, the attribute will be set on all the characters to the end of the line. This function moves cursor to position `(y, x)` if supplied. The changed line will be touched using the *touchline()* method so that the contents will be redisplayed by the next window refresh.

window.**clear**()
Like *erase()*, but also cause the whole window to be repainted upon next call to *refresh()*.

window.**clearok**(*yes*)
If *yes* is 1, the next call to *refresh()* will clear the window completely.

window.**clrtobot**()
Erase from cursor to the end of the window: all lines below the cursor are deleted, and then the equivalent of *clrtoeol()* is performed.

window.**clrtoeol**()
Erase from cursor to the end of the line.

window.**cursyncup**()
Update the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

window.**delch**([*y*, *x*])
Delete any character at `(y, x)`.

window.**deleteln**()
Delete the line under the cursor. All following lines are moved up by one line.

window.**derwin**(*begin_y*, *begin_x*)
window.**derwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)
An abbreviation for "derive window", *derwin()* is the same as calling *subwin()*, except that *begin_y* and *begin_x* are relative to the origin of the window, rather than relative to the entire screen. Return a window object for the derived window.

window.**echochar**(*ch*[, *attr*])
Add character *ch* with attribute *attr*, and immediately call *refresh()* on the window.

window.**enclose**(*y*, *x*)
Test whether the given pair of screen-relative character-cell coordinates are enclosed by the given window, returning `True` or `False`. It is useful for determining what subset of the screen windows enclose the location of a mouse event.

window.**erase**()
Clear the window.

window.**getbegyx**()
Return a tuple `(y, x)` of co-ordinates of upper-left corner.

window.**getbkgd**()
Return the given window's current background character/attribute pair.

window.**getch**([*y*, *x*])
Get a character. Note that the integer returned does *not* have to be in ASCII range: function keys, keypad keys and so on return numbers higher than 256. In no-delay mode, -1 is returned if there is no input, else *getch()* waits until a key is pressed.

window.**getkey**([*y*, *x*])
Get a character, returning a string instead of an integer, as *getch()* does. Function keys, keypad keys and so on return a multibyte string containing the key name. In no-delay mode, an exception is raised if there is no input.

window.**getmaxyx**()
Return a tuple `(y, x)` of the height and width of the window.

window.**getparyx**()
Return the beginning coordinates of this window relative to its parent window into two integer variables y and x. Return `-1, -1` if this window has no parent.

window.**getstr**([*y*, *x*])
Read a string from the user, with primitive line editing capacity.

window.**getyx**()
Return a tuple `(y, x)` of current cursor position relative to the window's upper-left corner.

window.**hline**(*ch*, *n*)
window.**hline**(*y*, *x*, *ch*, *n*)
Display a horizontal line starting at `(y, x)` with length *n* consisting of the character *ch*.

window.**idcok**(*flag*)
If *flag* is False, curses no longer considers using the hardware insert/delete character feature of the terminal; if *flag* is True, use of character insertion and deletion is enabled. When curses is first initialized, use of character insert/delete is enabled by default.

window.**idlok**(*yes*)
If called with *yes* equal to 1, *curses* will try and use hardware line editing facilities. Otherwise, line insertion/deletion are disabled.

window.**immedok**(*flag*)
If *flag* is True, any change in the window image automatically causes the window to be refreshed; you no longer have to call *refresh()* yourself. However, it may degrade performance considerably, due to repeated calls to wrefresh. This option is disabled by default.

window.**inch**([*y*, *x*])
Return the character at the given position in the window. The bottom 8 bits are the character proper, and upper bits are the attributes.

window.**insch**(*ch*[, *attr*])
window.**insch**(*y*, *x*, *ch*[, *attr*])
Paint character *ch* at (y, x) with attributes *attr*, moving the line from position *x* right by one character.

window.**insdelln**(*nlines*)
Insert *nlines* lines into the specified window above the current line. The *nlines* bottom lines are lost. For negative *nlines*, delete *nlines* lines starting with the one under the cursor, and move the remaining lines up. The bottom *nlines* lines are cleared. The current cursor position remains the same.

window.**insertln**()
Insert a blank line under the cursor. All following lines are moved down by one line.

window.**insnstr**(*str*, *n*[, *attr*])
window.**insnstr**(*y*, *x*, *str*, *n*[, *attr*])
Insert a character string (as many characters as will fit on the line) before the character under the cursor, up to *n* characters. If *n* is zero or negative, the entire string is inserted. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

window.**insstr**(*str*[, *attr*])
window.**insstr**(*y*, *x*, *str*[, *attr*])
Insert a character string (as many characters as will fit on the line) before the character under the cursor. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

window.**instr**([*n*])
window.**instr**(*y*, *x*[, *n*])
Return a string of characters, extracted from the window starting at the current cursor position, or at *y*, *x* if specified. Attributes are stripped from the characters. If *n* is specified, *instr()* returns a string at most *n* characters long (exclusive of the trailing NUL).

window.**is_linetouched**(*line*)
Return True if the specified line was modified since the last call to *refresh()*; otherwise return False. Raise a *curses.error* exception if *line* is not valid for the given window.

window.**is_wintouched**()
Return True if the specified window was modified since the last call to *refresh()*; otherwise return False.

window.**keypad**(*yes*)
If *yes* is 1, escape sequences generated by some keys (keypad, function keys) will be interpreted by *curses*. If *yes* is 0, escape sequences will be left as is in the input stream.

window.**leaveok**(*yes*)
If *yes* is 1, cursor is left where it is on update, instead of being at "cursor position." This reduces cursor movement where possible. If possible the cursor will be made invisible.

If *yes* is 0, cursor will always be at "cursor position" after an update.

window.**move**(*new_y*, *new_x*)
Move cursor to `(new_y, new_x)`.

window.**mvderwin**(*y*, *x*)
Move the window inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

window.**mvwin**(*new_y*, *new_x*)
Move the window so its upper-left corner is at `(new_y, new_x)`.

window.**nodelay**(*yes*)
If *yes* is `1`, *getch()* will be non-blocking.

window.**notimeout**(*yes*)
If *yes* is `1`, escape sequences will not be timed out.

If *yes* is `0`, after a few milliseconds, an escape sequence will not be interpreted, and will be left in the input stream as is.

window.**noutrefresh**()
Mark for refresh but wait. This function updates the data structure representing the desired state of the window, but does not force an update of the physical screen. To accomplish that, call *doupdate()*.

window.**overlay**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])
Overlay the window on top of *destwin*. The windows need not be the same size, only the overlapping region is copied. This copy is non-destructive, which means that the current background character does not overwrite the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of *overlay()* can be used. *sminrow* and *smincol* are the upper-left coordinates of the source window, and the other variables mark a rectangle in the destination window.

window.**overwrite**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])
Overwrite the window on top of *destwin*. The windows need not be the same size, in which case only the overlapping region is copied. This copy is destructive, which means that the current background character overwrites the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of *overwrite()* can be used. *sminrow* and *smincol* are the upper-left coordinates of the source window, the other variables mark a rectangle in the destination window.

window.**putwin**(*file*)
Write all data associated with the window into the provided file object. This information can be later retrieved using the *getwin()* function.

window.**redrawln**(*beg*, *num*)
Indicate that the *num* screen lines, starting at line *beg*, are corrupted and should be completely redrawn on the next *refresh()* call.

window.**redrawwin**()
Touch the entire window, causing it to be completely redrawn on the next *refresh()* call.

window.**refresh**([*pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*])
Update the display immediately (sync actual screen with previous drawing/deleting methods).

The 6 optional arguments can only be specified when the window is a pad created with `newpad()`. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left-hand corner of the rectangle to be displayed in the pad. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

`window.`**`resize`**(*nlines*, *ncols*)
Reallocate storage for a curses window to adjust its dimensions to the specified values. If either dimension is larger than the current values, the window's data is filled with blanks that have the current background rendition (as set by `bkgdset()`) merged into them.

`window.`**`scroll`**([*lines=1*])
Scroll the screen or scrolling region upward by *lines* lines.

`window.`**`scrollok`**(*flag*)
Control what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If *flag* is false, the cursor is left on the bottom line. If *flag* is true, the window is scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call `idlok()`.

`window.`**`setscrreg`**(*top*, *bottom*)
Set the scrolling region from line *top* to line *bottom*. All scrolling actions will take place in this region.

`window.`**`standend`**()
Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

`window.`**`standout`**()
Turn on attribute *A_STANDOUT*.

`window.`**`subpad`**(*begin_y*, *begin_x*)
`window.`**`subpad`**(*nlines*, *ncols*, *begin_y*, *begin_x*)
Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is *ncols*/*nlines*.

`window.`**`subwin`**(*begin_y*, *begin_x*)
`window.`**`subwin`**(*nlines*, *ncols*, *begin_y*, *begin_x*)
Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is *ncols*/*nlines*.

By default, the sub-window will extend from the specified position to the lower right corner of the window.

`window.`**`syncdown`**()
Touch each location in the window that has been touched in any of its ancestor windows. This routine is called by `refresh()`, so it should almost never be necessary to call it manually.

`window.`**`syncok`**(*flag*)
If called with *flag* set to `True`, then `syncup()` is called automatically whenever there is a change in the window.

`window.`**`syncup`**()
Touch all locations in ancestors of the window that have been changed in the window.

`window.`**`timeout`**(*delay*)
Set blocking or non-blocking read behavior for the window. If *delay* is negative, blocking read is used (which will wait indefinitely for input). If *delay* is zero, then non-blocking read is used, and -1 will be returned by `getch()` if no input is waiting. If *delay* is positive, then `getch()` will block for *delay* milliseconds, and return -1 if there is still no input at the end of that time.

window.**touchline**(*start*, *count*[, *changed*])
Pretend *count* lines have been changed, starting with line *start*. If *changed* is supplied, it specifies whether the affected lines are marked as having been changed (*changed*=1) or unchanged (*changed*=0).

window.**touchwin**()
Pretend the whole window has been changed, for purposes of drawing optimizations.

window.**untouchwin**()
Mark all lines in the window as unchanged since the last call to *refresh()*.

window.**vline**(*ch*, *n*)
window.**vline**(*y*, *x*, *ch*, *n*)
Display a vertical line starting at (y, x) with length *n* consisting of the character *ch*.

15.11.3 常量

The *curses* module defines the following data members:

curses.**ERR**
Some curses routines that return an integer, such as getch(), return *ERR* upon failure.

curses.**OK**
Some curses routines that return an integer, such as *napms()*, return *OK* upon success.

curses.**version**
A string representing the current version of the module. Also available as __version__.

Some constants are available to specify character cell attributes. The exact constants available are system dependent.

属性	含义
A_ALTCHARSET	备用字符集模式
A_BLINK	闪烁模式
A_BOLD	粗体模式
A_DIM	暗淡模式
A_INVIS	不可见或空白模式
A_NORMAL	正常属性
A_PROTECT	保护模式
A_REVERSE	反转背景色和前景色
A_STANDOUT	突出模式
A_UNDERLINE	下划线模式
A_HORIZONTAL	水平突出显示
A_LEFT	左高亮
A_LOW	底部高亮
A_RIGHT	右高亮
A_TOP	顶部高亮
A_VERTICAL	垂直突出显示
A_CHARTEXT	用于提取字符的位掩码

有几个常量可用于提取某些方法返回的相应属性。

位掩码	含义
A_ATTRIBUTES	用于提取属性的位掩码
A_CHARTEXT	用于提取字符的位掩码
A_COLOR	用于提取颜色对字段信息的位掩码

键由名称以 KEY_ 开头的整数常量引用。确切的可用键取决于系统。

关键常数	（Windows 注册表的）键
KEY_MIN	最小键值
KEY_BREAK	中断键（不可靠）
KEY_DOWN	向下箭头
KEY_UP	向上箭头
KEY_LEFT	向左箭头
KEY_RIGHT	向右箭头
KEY_HOME	Home key (upward+left arrow)
KEY_BACKSPACE	退格（不可靠）
KEY_F0	Function keys. Up to 64 function keys are supported.
KEY_Fn	Value of function key *n*
KEY_DL	删除行
KEY_IL	插入行
KEY_DC	Delete character
KEY_IC	Insert char or enter insert mode
KEY_EIC	Exit insert char mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll 1 line forward
KEY_SR	Scroll 1 line backward (reverse)
KEY_NPAGE	下一页
KEY_PPAGE	上一页
KEY_STAB	Set tab
KEY_CTAB	Clear tab
KEY_CATAB	Clear all tabs
KEY_ENTER	Enter or send (unreliable)
KEY_SRESET	Soft (partial) reset (unreliable)
KEY_RESET	Reset or hard reset (unreliable)
KEY_PRINT	打印
KEY_LL	Home down or bottom (lower left)
KEY_A1	键盘的左上角
KEY_A3	键盘的右上角
KEY_B2	键盘的中心
KEY_C1	键盘左下方
KEY_C3	键盘右下方
KEY_BTAB	Back tab
KEY_BEG	Beg (beginning)
KEY_CANCEL	取消
KEY_CLOSE	关闭
KEY_COMMAND	Cmd (命令行)
KEY_COPY	复制
KEY_CREATE	创建
KEY_END	End
KEY_EXIT	退出
KEY_FIND	查找
KEY_HELP	帮助
KEY_MARK	标记
KEY_MESSAGE	消息

下页继续

表 1 – 续上页

关键常数	（Windows 注册表的）键
KEY_MOVE	移动
KEY_NEXT	下一个
KEY_OPEN	打开
KEY_OPTIONS	选项
KEY_PREVIOUS	Prev (previous)
KEY_REDO	重做
KEY_REFERENCE	Ref (reference)
KEY_REFRESH	刷新
KEY_REPLACE	替换
KEY_RESTART	重启
KEY_RESUME	恢复
KEY_SAVE	保存
KEY_SBEG	Shifted Beg (beginning)
KEY_SCANCEL	Shifted Cancel
KEY_SCOMMAND	Shifted Command
KEY_SCOPY	Shifted Copy
KEY_SCREATE	Shifted Create
KEY_SDC	Shifted Delete char
KEY_SDL	Shifted Delete line
KEY_SELECT	Select
KEY_SEND	Shifted End
KEY_SEOL	Shifted Clear line
KEY_SEXIT	Shifted Exit
KEY_SFIND	Shifted Find
KEY_SHELP	Shifted Help
KEY_SHOME	Shifted Home
KEY_SIC	Shifted Input
KEY_SLEFT	Shifted Left arrow
KEY_SMESSAGE	Shifted Message
KEY_SMOVE	Shifted Move
KEY_SNEXT	Shifted Next
KEY_SOPTIONS	Shifted Options
KEY_SPREVIOUS	Shifted Prev
KEY_SPRINT	Shifted Print
KEY_SREDO	Shifted Redo
KEY_SREPLACE	Shifted Replace
KEY_SRIGHT	Shifted Right arrow
KEY_SRSUME	Shifted Resume
KEY_SSAVE	Shifted Save
KEY_SSUSPEND	Shifted Suspend
KEY_SUNDO	Shifted Undo
KEY_SUSPEND	Suspend
KEY_UNDO	撤销操作
KEY_MOUSE	Mouse event has occurred
KEY_RESIZE	Terminal resize event
KEY_MAX	Maximum key value

在 VT100 及其软件仿真（例如 X 终端仿真器）上，通常至少有四个功能键（KEY_F1, KEY_F2, KEY_F3, KEY_F4 ）可用，并且箭头键以明显的方式映射到 KEY_UP, KEY_DOWN, KEY_LEFT 和 KEY_RIGHT 。如果您的机器有一个 PC 键盘，可以安全地使用箭头键和十二个功能键（旧的 PC 键盘可能只有十个功能键）；此外，

以下键盘映射是标准的：

键帽	常数
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

The following table lists characters from the alternate character set. These are inherited from the VT100 terminal, and will generally be available on software emulations such as X terminals. When there is no graphic available, curses falls back on a crude printable ASCII approximation.

注解： 只有在调用*initscr()*之后才能使用它们

ACS 代码	含义
ACS_BBSS	右上角的别名
ACS_BLOCK	实心方块
ACS_BOARD	正方形
ACS_BSBS	水平线的别名
ACS_BSSB	左上角的别名
ACS_BSSS	顶部 T 型的别名
ACS_BTEE	底部 T 型
ACS_BULLET	正方形
ACS_CKBOARD	棋盘（点刻）
ACS_DARROW	向下箭头
ACS_DEGREE	等级符
ACS_DIAMOND	菱形
ACS_GEQUAL	大于或等于
ACS_HLINE	水平线
ACS_LANTERN	灯形符号
ACS_LARROW	向左箭头
ACS_LEQUAL	小于或等于
ACS_LLCORNER	左下角
ACS_LRCORNER	右下角
ACS_LTEE	左侧 T 型
ACS_NEQUAL	不等号
ACS_PI	字母 π
ACS_PLMINUS	正负号
ACS_PLUS	加号
ACS_RARROW	向右箭头
ACS_RTEE	右侧 T 型
ACS_S1	扫描线 1
ACS_S3	扫描线 3
ACS_S7	扫描线 7
ACS_S9	扫描线 9
ACS_SBBS	右下角的别名
ACS_SBSB	垂直线的别名

下页继续

表 2 – 续上页

ACS 代码	含义
ACS_SBSS	右侧 T 型的别名
ACS_SSBB	左下角的别名
ACS_SSBS	底部 T 型的别名
ACS_SSSB	左侧 T 型的别名
ACS_SSSS	alternate name for crossover or big plus
ACS_STERLING	英镑
ACS_TTEE	顶部 T 型
ACS_UARROW	向上箭头
ACS_ULCORNER	左上角
ACS_URCORNER	右上角
ACS_VLINE	垂线

下表列出了预定义的颜色：

常数	颜色
COLOR_BLACK	黑色
COLOR_BLUE	蓝色
COLOR_CYAN	青色（浅绿蓝色）
COLOR_GREEN	绿色
COLOR_MAGENTA	洋红色（紫红色）
COLOR_RED	红色
COLOR_WHITE	白色
COLOR_YELLOW	黄色

15.12 curses.textpad —Text input widget for curses programs

1.6 新版功能.

The *curses.textpad* module provides a *Textbox* class that handles elementary text editing in a curses window, supporting a set of keybindings resembling those of Emacs (thus, also of Netscape Navigator, BBedit 6.x, FrameMaker, and many other programs). The module also provides a rectangle-drawing function useful for framing text boxes or for other purposes.

The module *curses.textpad* defines the following function:

curses.textpad.**rectangle**(*win*, *uly*, *ulx*, *lry*, *lrx*)

Draw a rectangle. The first argument must be a window object; the remaining arguments are coordinates relative to that window. The second and third arguments are the y and x coordinates of the upper left hand corner of the rectangle to be drawn; the fourth and fifth arguments are the y and x coordinates of the lower right hand corner. The rectangle will be drawn using VT100/IBM PC forms characters on terminals that make this possible (including xterm and most other software terminal emulators). Otherwise it will be drawn with ASCII dashes, vertical bars, and plus signs.

15.12.1 文本框对象

You can instantiate a *Textbox* object as follows:

class curses.textpad.**Textbox**(*win*)

Return a textbox widget object. The *win* argument should be a curses WindowObject in which the textbox is to be contained. The edit cursor of the textbox is initially located at the upper left hand corner of the containing window, with coordinates (0, 0). The instance's *stripspaces* flag is initially on.

Textbox objects have the following methods:

edit([*validator*])

This is the entry point you will normally use. It accepts editing keystrokes until one of the termination keystrokes is entered. If *validator* is supplied, it must be a function. It will be called for each keystroke entered with the keystroke as a parameter; command dispatch is done on the result. This method returns the window contents as a string; whether blanks in the window are included is affected by the *stripspaces* attribute.

do_command(*ch*)

处理单个按键命令。以下是支持的特殊按键：

按键	动作
Control-A	转到窗口的左边缘。
Control-B	光标向左，如果可能，包含前一行。
Control-D	删除光标下的字符。
Control-E	Go to right edge (stripspaces off) or end of line (stripspaces on).
Control-F	向右移动光标，适当时换行到下一行。
Control-G	终止，返回窗口内容。
Control-H	向后删除字符。
Control-J	如果窗口是 1 行则终止，否则插入换行符。
Control-K	如果行为空，则删除它，否则清除到行尾。
Control-L	刷新屏幕。
Control-N	光标向下; 向下移动一行。
Control-O	在光标位置插入一个空行。
Control-P	光标向上; 向上移动一行。

如果光标位于无法移动的边缘，则移动操作不执行任何操作。在可能的情况下，支持以下同义词：

常数	按键
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

All other keystrokes are treated as a command to insert the given character and move right (with line wrapping).

gather()

Return the window contents as a string; whether blanks in the window are included is affected by the *stripspaces* member.

stripspaces

This attribute is a flag which controls the interpretation of blanks in the window. When it is on, trailing blanks

on each line are ignored; any cursor motion that would land the cursor on a trailing blank goes to the end of that line instead, and trailing blanks are stripped when the window contents are gathered.

15.13 curses.ascii —Utilities for ASCII characters

1.6 新版功能.

The *curses.ascii* module supplies name constants for ASCII characters and functions to test membership in various ASCII character classes. The constants supplied are names for control characters as follows:

名称	含义
NUL	
SOH	标题开始，控制台中断
STX	文本开始
ETX	文本结束
EOT	传输结束
ENQ	查询，附带 ACK 流量控制
ACK	确认
BEL	蜂鸣器
BS	退格
TAB	Tab
HT	TAB 的别名：“水平制表符”
LF	换行
NL	LF 的别名：“新行”
VT	垂直制表符
FF	换页
CR	回车
SO	Shift-out, begin alternate character set
SI	Shift-in, resume default character set
DLE	Data-link escape
DC1	XON, for flow control
DC2	Device control 2, block-mode flow control
DC3	XOFF, for flow control
DC4	设备控制 4
NAK	否定确认
SYN	同步空闲
ETB	末端传输块
CAN	取消
EM	媒体结束
SUB	替换
ESC	退出
FS	文件分隔符
GS	Group separator
RS	记录分隔符，块模式终结器
US	单位分隔符
SP	空格
DEL	删除

Note that many of these have little practical significance in modern usage. The mnemonics derive from teleprinter conventions that predate digital computers.

The module supplies the following functions, patterned on those in the standard C library:

curses.ascii.**isalnum**(*c*)
Checks for an ASCII alphanumeric character; it is equivalent to `isalpha(c) or isdigit(c)`.

curses.ascii.**isalpha**(*c*)
Checks for an ASCII alphabetic character; it is equivalent to `isupper(c) or islower(c)`.

curses.ascii.**isascii**(*c*)
Checks for a character value that fits in the 7-bit ASCII set.

curses.ascii.**isblank**(*c*)
Checks for an ASCII whitespace character; space or horizontal tab.

curses.ascii.**iscntrl**(*c*)
Checks for an ASCII control character (in the range 0x00 to 0x1f or 0x7f).

curses.ascii.**isdigit**(*c*)
Checks for an ASCII decimal digit, `'0'` through `'9'`. This is equivalent to `c in string.digits`.

curses.ascii.**isgraph**(*c*)
Checks for ASCII any printable character except space.

curses.ascii.**islower**(*c*)
Checks for an ASCII lower-case character.

curses.ascii.**isprint**(*c*)
Checks for any ASCII printable character including space.

curses.ascii.**ispunct**(*c*)
Checks for any printable ASCII character which is not a space or an alphanumeric character.

curses.ascii.**isspace**(*c*)
Checks for ASCII white-space characters; space, line feed, carriage return, form feed, horizontal tab, vertical tab.

curses.ascii.**isupper**(*c*)
Checks for an ASCII uppercase letter.

curses.ascii.**isxdigit**(*c*)
Checks for an ASCII hexadecimal digit. This is equivalent to `c in string.hexdigits`.

curses.ascii.**isctrl**(*c*)
Checks for an ASCII control character (ordinal values 0 to 31).

curses.ascii.**ismeta**(*c*)
Checks for a non-ASCII character (ordinal values 0x80 and above).

These functions accept either integers or strings; when the argument is a string, it is first converted using the built-in function *ord()*.

Note that all these functions check ordinal bit values derived from the first character of the string you pass in; they do not actually know anything about the host machine's character encoding. For functions that know about the character encoding (and handle internationalization properly) see the *string* module.

The following two functions take either a single-character string or integer byte value; they return a value of the same type.

curses.ascii.**ascii**(*c*)
Return the ASCII value corresponding to the low 7 bits of *c*.

curses.ascii.**ctrl**(*c*)
Return the control character corresponding to the given character (the character bit value is bitwise-anded with 0x1f).

curses.ascii.**alt**(*c*)
Return the 8-bit character corresponding to the given ASCII character (the character bit value is bitwise-ored with 0x80).

The following function takes either a single-character string or integer value; it returns a string.

curses.ascii.**unctrl**(*c*)
Return a string representation of the ASCII character *c*. If *c* is printable, this string is the character itself. If the character is a control character (0x00–0x1f) the string consists of a caret (`'^'`) followed by the corresponding uppercase letter. If the character is an ASCII delete (0x7f) the string is `'^?'`. If the character has its meta bit (0x80) set, the meta bit is stripped, the preceding rules applied, and `'!'` prepended to the result.

curses.ascii.**controlnames**
A 33-element string array that contains the ASCII mnemonics for the thirty-two ASCII control characters from 0 (NUL) to 0x1f (US), in order, plus the mnemonic `SP` for the space character.

15.14 curses.panel —A panel stack extension for curses

Panels are windows with the added feature of depth, so they can be stacked on top of each other, and only the visible portions of each window will be displayed. Panels can be added, moved up or down in the stack, and removed.

15.14.1 函数

The module *curses.panel* defines the following functions:

curses.panel.**bottom_panel**()
Returns the bottom panel in the panel stack.

curses.panel.**new_panel**(*win*)
Returns a panel object, associating it with the given window *win*. Be aware that you need to keep the returned panel object referenced explicitly. If you don't, the panel object is garbage collected and removed from the panel stack.

curses.panel.**top_panel**()
Returns the top panel in the panel stack.

curses.panel.**update_panels**()
Updates the virtual screen after changes in the panel stack. This does not call *curses.doupdate()*, so you'll have to do this yourself.

15.14.2 Panel Objects

Panel objects, as returned by *new_panel()* above, are windows with a stacking order. There's always a window associated with a panel which determines the content, while the panel methods are responsible for the window's depth in the panel stack.

Panel objects have the following methods:

Panel.**above**()
Returns the panel above the current panel.

Panel.**below**()
Returns the panel below the current panel.

Panel.**bottom**()
Push the panel to the bottom of the stack.

Panel.**hidden**()
Returns true if the panel is hidden (not visible), false otherwise.

Panel.**hide**()
Hide the panel. This does not delete the object, it just makes the window on screen invisible.

Panel.**move**(*y*, *x*)
Move the panel to the screen coordinates `(y, x)`.

Panel.**replace**(*win*)
Change the window associated with the panel to the window *win*.

Panel.**set_userptr**(*obj*)
Set the panel's user pointer to *obj*. This is used to associate an arbitrary piece of data with the panel, and can be any Python object.

Panel.**show**()
Display the panel (which might have been hidden).

Panel.**top**()
Push panel to the top of the stack.

Panel.**userptr**()
Returns the user pointer for the panel. This might be any Python object.

Panel.**window**()
Returns the window object associated with the panel.

15.15 `platform` —获取底层平台的标识数据

2.3 新版功能.

示例代码： Lib/platform.py

注解： Specific platforms listed alphabetically, with Linux included in the Unix section.

15.15.1 跨平台

platform.**architecture**(*executable=sys.executable*, *bits=''*, *linkage=''*)
Queries the given executable (defaults to the Python interpreter binary) for various architecture information.

Returns a tuple `(bits, linkage)` which contain information about the bit architecture and the linkage format used for the executable. Both values are returned as strings.

Values that cannot be determined are returned as given by the parameter presets. If bits is given as `''`, the `sizeof(pointer)` (or `sizeof(long)` on Python version < 1.5.2) is used as indicator for the supported pointer size.

The function relies on the system's `file` command to do the actual work. This is available on most if not all Unix platforms and some non-Unix platforms and then only if the executable points to the Python interpreter. Reasonable defaults are used when the above needs are not met.

注解： On Mac OS X (and perhaps other platforms), executable files may be universal files containing multiple architectures.

To get at the "64-bitness" of the current interpreter, it is more reliable to query the *sys.maxsize* attribute:

```
is_64bits = sys.maxsize > 2**32
```

---

platform.**machine**()
Returns the machine type, e.g. 'i386'. An empty string is returned if the value cannot be determined.

platform.**node**()
Returns the computer's network name (may not be fully qualified!). An empty string is returned if the value cannot be determined.

platform.**platform**(*aliased=0*, *terse=0*)
Returns a single string identifying the underlying platform with as much useful information as possible.

The output is intended to be *human readable* rather than machine parseable. It may look different on different platforms and this is intended.

If *aliased* is true, the function will use aliases for various platforms that report system names which differ from their common names, for example SunOS will be reported as Solaris. The *system_alias()* function is used to implement this.

Setting *terse* to true causes the function to return only the absolute minimum information needed to identify the platform.

platform.**processor**()
Returns the (real) processor name, e.g. 'amdk6'.

An empty string is returned if the value cannot be determined. Note that many platforms do not provide this information or simply return the same value as for *machine()*. NetBSD does this.

platform.**python_build**()
Returns a tuple (buildno, builddate) stating the Python build number and date as strings.

platform.**python_compiler**()
Returns a string identifying the compiler used for compiling Python.

platform.**python_branch**()
Returns a string identifying the Python implementation SCM branch.

2.6 新版功能.

platform.**python_implementation**()
Returns a string identifying the Python implementation. Possible return values are: 'CPython', 'IronPython', 'Jython', 'PyPy'.

2.6 新版功能.

platform.**python_revision**()
Returns a string identifying the Python implementation SCM revision.

2.6 新版功能.

platform.**python_version**()
Returns the Python version as string 'major.minor.patchlevel'.

Note that unlike the Python sys.version, the returned value will always include the patchlevel (it defaults to 0).

platform.**python_version_tuple**()
Returns the Python version as tuple (major, minor, patchlevel) of strings.

Note that unlike the Python sys.version, the returned value will always include the patchlevel (it defaults to '0').

platform.**release**()
Returns the system's release, e.g. '2.2.0' or 'NT' An empty string is returned if the value cannot be determined.

platform.**system**()
Returns the system/OS name, e.g. 'Linux', 'Windows', or 'Java'. An empty string is returned if the value cannot be determined.

platform.**system_alias**(*system*, *release*, *version*)
Returns (system, release, version) aliased to common marketing names used for some systems. It also does some reordering of the information in some cases where it would otherwise cause confusion.

platform.**version**()
Returns the system's release version, e.g. '#3 on degas'. An empty string is returned if the value cannot be determined.

platform.**uname**()
Fairly portable uname interface. Returns a tuple of strings (system, node, release, version, machine, processor) identifying the underlying platform.

Note that unlike the *os.uname()* function this also returns possible processor information as additional tuple entry.

Entries which cannot be determined are set to ''.

15.15.2 Java 平台

platform.**java_ver**(*release=''*, *vendor=''*, *vminfo=('', '', '')*, *osinfo=('', '', '')*)
Jython 的版本接口

Returns a tuple (release, vendor, vminfo, osinfo) with *vminfo* being a tuple (vm_name, vm_release, vm_vendor) and *osinfo* being a tuple (os_name, os_version, os_arch). Values which cannot be determined are set to the defaults given as parameters (which all default to '').

15.15.3 Windows 平台

platform.**win32_ver**(*release=''*, *version=''*, *csd=''*, *ptype=''*)
Get additional version information from the Windows Registry and return a tuple (release, version, csd, ptype) referring to OS release, version number, CSD level (service pack) and OS type (multi/single processor).

As a hint: *ptype* is 'Uniprocessor Free' on single processor NT machines and 'Multiprocessor Free' on multi processor machines. The '*Free*' refers to the OS version being free of debugging code. It could also state '*Checked*' which means the OS version uses debugging code, i.e. code that checks arguments, ranges, etc.

注解: This function works best with Mark Hammond's win32all package installed, but also on Python 2.3 and later (support for this was added in Python 2.6). It obviously only runs on Win32 compatible platforms.

Win95/98 specific

platform.**popen**(*cmd*, *mode='r'*, *bufsize=None*)
Portable *popen()* interface. Find a working popen implementation preferring `win32pipe.popen()`. On Windows NT, `win32pipe.popen()` should work; on Windows 9x it hangs due to bugs in the MS C library.

15.15.4 Mac OS 平台

platform.**mac_ver**(*release=''*, *versioninfo=('', '', '')*, *machine=''*)
Get Mac OS version information and return it as tuple `(release, versioninfo, machine)` with *versioninfo* being a tuple `(version, dev_stage, non_release_version)`.

Entries which cannot be determined are set to `''`. All tuple entries are strings.

15.15.5 Unix Platforms

platform.**dist**(*distname=''*, *version=''*, *id=''*, *supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...)*)
This is an old version of the functionality now provided by *linux_distribution()*. For new code, please use the *linux_distribution()*.

The only difference between the two is that `dist()` always returns the short name of the distribution taken from the `supported_dists` parameter.

2.6 版后已移除.

platform.**linux_distribution**(*distname=''*, *version=''*, *id=''*, *supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...)*, *full_distribution_name=1*)
Tries to determine the name of the Linux OS distribution name.

`supported_dists` may be given to define the set of Linux distributions to look for. It defaults to a list of currently supported Linux distributions identified by their release file name.

If `full_distribution_name` is true (default), the full distribution read from the OS is returned. Otherwise the short name taken from `supported_dists` is used.

Returns a tuple `(distname,version,id)` which defaults to the args given as parameters. `id` is the item in parentheses after the version number. It is usually the version codename.

注解： This function is deprecated since Python 3.5 and removed in Python 3.8. See alternative like the distro package.

2.6 新版功能.

platform.**libc_ver**(*executable=sys.executable*, *lib=''*, *version=''*, *chunksize=2048*)
Tries to determine the libc version against which the file executable (defaults to the Python interpreter) is linked. Returns a tuple of strings `(lib, version)` which default to the given parameters in case the lookup fails.

Note that this function has intimate knowledge of how different libc versions add symbols to the executable is probably only usable for executables compiled using **gcc**.

The file is read and scanned in chunks of *chunksize* bytes.

15.16 errno —Standard errno system symbols

This module makes available standard errno system symbols. The value of each symbol is the corresponding integer value. The names and descriptions are borrowed from linux/include/errno.h, which should be pretty all-inclusive.

errno.**errorcode**
Dictionary providing a mapping from the errno value to the string name in the underlying system. For instance, errno.errorcode[errno.EPERM] maps to 'EPERM'.

To translate a numeric error code to an error message, use *os.strerror()*.

Of the following list, symbols that are not used on the current platform are not defined by the module. The specific list of defined symbols is available as errno.errorcode.keys(). Symbols available can include:

errno.**EPERM**
Operation not permitted

errno.**ENOENT**
No such file or directory

errno.**ESRCH**
No such process

errno.**EINTR**
Interrupted system call

errno.**EIO**
I/O error

errno.**ENXIO**
No such device or address

errno.**E2BIG**
Arg list too long

errno.**ENOEXEC**
Exec format error

errno.**EBADF**
Bad file number

errno.**ECHILD**
No child processes

errno.**EAGAIN**
Try again

errno.**ENOMEM**
Out of memory

errno.**EACCES**
Permission denied

errno.**EFAULT**
Bad address

errno.**ENOTBLK**
Block device required

errno.**EBUSY**
Device or resource busy

errno.**EEXIST**
File exists

errno.**EXDEV**
Cross-device link

errno.**ENODEV**
No such device

errno.**ENOTDIR**
Not a directory

errno.**EISDIR**
Is a directory

errno.**EINVAL**
Invalid argument

errno.**ENFILE**
File table overflow

errno.**EMFILE**
Too many open files

errno.**ENOTTY**
Not a typewriter

errno.**ETXTBSY**
Text file busy

errno.**EFBIG**
File too large

errno.**ENOSPC**
No space left on device

errno.**ESPIPE**
Illegal seek

errno.**EROFS**
Read-only file system

errno.**EMLINK**
Too many links

errno.**EPIPE**
Broken pipe

errno.**EDOM**
Math argument out of domain of func

errno.**ERANGE**
Math result not representable

errno.**EDEADLK**
Resource deadlock would occur

errno.**ENAMETOOLONG**
File name too long

errno.**ENOLCK**
No record locks available

errno.**ENOSYS**
Function not implemented

errno.**ENOTEMPTY**
Directory not empty

errno.**ELOOP**
Too many symbolic links encountered

errno.**EWOULDBLOCK**
Operation would block

errno.**ENOMSG**
No message of desired type

errno.**EIDRM**
Identifier removed

errno.**ECHRNG**
Channel number out of range

errno.**EL2NSYNC**
Level 2 not synchronized

errno.**EL3HLT**
Level 3 halted

errno.**EL3RST**
Level 3 reset

errno.**ELNRNG**
Link number out of range

errno.**EUNATCH**
Protocol driver not attached

errno.**ENOCSI**
No CSI structure available

errno.**EL2HLT**
Level 2 halted

errno.**EBADE**
Invalid exchange

errno.**EBADR**
Invalid request descriptor

errno.**EXFULL**
Exchange full

errno.**ENOANO**
No anode

errno.**EBADRQC**
Invalid request code

errno.**EBADSLT**
Invalid slot

errno.**EDEADLOCK**
File locking deadlock error

errno.**EBFONT**
Bad font file format

errno.**ENOSTR**
Device not a stream

errno.**ENODATA**
No data available

errno.**ETIME**
Timer expired

errno.**ENOSR**
Out of streams resources

errno.**ENONET**
Machine is not on the network

errno.**ENOPKG**
Package not installed

errno.**EREMOTE**
Object is remote

errno.**ENOLINK**
Link has been severed

errno.**EADV**
Advertise error

errno.**ESRMNT**
Srmount error

errno.**ECOMM**
Communication error on send

errno.**EPROTO**
Protocol error

errno.**EMULTIHOP**
Multihop attempted

errno.**EDOTDOT**
RFS specific error

errno.**EBADMSG**
Not a data message

errno.**EOVERFLOW**
Value too large for defined data type

errno.**ENOTUNIQ**
Name not unique on network

errno.**EBADFD**
File descriptor in bad state

errno.**EREMCHG**
Remote address changed

errno.**ELIBACC**
Can not access a needed shared library

errno.**ELIBBAD**
Accessing a corrupted shared library

errno.**ELIBSCN**
.lib section in a.out corrupted

errno.**ELIBMAX**
Attempting to link in too many shared libraries

errno.**ELIBEXEC**
Cannot exec a shared library directly

errno.**EILSEQ**
Illegal byte sequence

errno.**ERESTART**
Interrupted system call should be restarted

errno.**ESTRPIPE**
Streams pipe error

errno.**EUSERS**
Too many users

errno.**ENOTSOCK**
Socket operation on non-socket

errno.**EDESTADDRREQ**
Destination address required

errno.**EMSGSIZE**
Message too long

errno.**EPROTOTYPE**
Protocol wrong type for socket

errno.**ENOPROTOOPT**
Protocol not available

errno.**EPROTONOSUPPORT**
Protocol not supported

errno.**ESOCKTNOSUPPORT**
Socket type not supported

errno.**EOPNOTSUPP**
Operation not supported on transport endpoint

errno.**EPFNOSUPPORT**
Protocol family not supported

errno.**EAFNOSUPPORT**
Address family not supported by protocol

errno.**EADDRINUSE**
Address already in use

errno.**EADDRNOTAVAIL**
Cannot assign requested address

errno.**ENETDOWN**
Network is down

errno.**ENETUNREACH**
Network is unreachable

errno.**ENETRESET**
Network dropped connection because of reset

errno.**ECONNABORTED**
Software caused connection abort

errno.**ECONNRESET**
Connection reset by peer

errno.**ENOBUFS**
No buffer space available

errno.**EISCONN**
Transport endpoint is already connected

errno.**ENOTCONN**
Transport endpoint is not connected

errno.**ESHUTDOWN**
Cannot send after transport endpoint shutdown

errno.**ETOOMANYREFS**
Too many references: cannot splice

errno.**ETIMEDOUT**
Connection timed out

errno.**ECONNREFUSED**
Connection refused

errno.**EHOSTDOWN**
Host is down

errno.**EHOSTUNREACH**
No route to host

errno.**EALREADY**
Operation already in progress

errno.**EINPROGRESS**
Operation now in progress

errno.**ESTALE**
Stale NFS file handle

errno.**EUCLEAN**
Structure needs cleaning

errno.**ENOTNAM**
Not a XENIX named type file

errno.**ENAVAIL**
No XENIX semaphores available

errno.**EISNAM**
Is a named type file

errno.**EREMOTEIO**
Remote I/O error

errno.**EDQUOT**
Quota exceeded

15.17 ctypes —Python 的外部函数库

2.5 新版功能.

ctypes 是 Python 的外部函数库。它提供了与 C 兼容的数据类型，并允许调用 DLL 或共享库中的函数。可使用该模块以纯 Python 形式对这些库进行封装。

15.17.1 ctypes 教程

注意：在本教程中的示例代码使用*doctest* 进行过测试，保证其正确运行。由于有些代码在 Linux，Windows 或 Mac OS X 下的表现不同，这些代码会在 doctest 中包含相关的指令注解。

Note: Some code samples reference the ctypes *c_int* type. This type is an alias for the *c_long* type on 32-bit systems. So, you should not be confused if *c_long* is printed if you would expect *c_int* —they are actually the same type.

载入动态连接库

ctypes 导出了 *cdll* 对象，在 Windows 系统中还导出了 *windll* 和 *oledll* 对象用于载入动态连接库。

You load libraries by accessing them as attributes of these objects. *cdll* loads libraries which export functions using the standard `cdecl` calling convention, while *windll* libraries call functions using the `stdcall` calling convention. *oledll* also uses the `stdcall` calling convention, and assumes the functions return a Windows `HRESULT` error code. The error code is used to automatically raise a `WindowsError` exception when the function call fails.

这是一些 Windows 下的例子。注意：`msvcrt` 是微软 C 标准库，包含了大部分 C 标准函数，这些函数都是以 cdecl 调用协议进行调用的。

```
>>> from ctypes import *
>>> print windll.kernel32
<WinDLL 'kernel32', handle ... at ...>
>>> print cdll.msvcrt
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Windows 会自动添加通常的 `.dll` 文件扩展名。

在 Linux 下，必须使用 包含文件扩展名的文件名来导入共享库。因此不能简单使用对象属性的方式来导入库。因此，你可以使用方法 `LoadLibrary()`，或构造 CDLL 对象来导入库。

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

操作导入的动态链接库中的函数

通过操作 dll 对象的属性来操作这些函数。

```
>>> from ctypes import *
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print windll.kernel32.GetModuleHandleA
<_FuncPtr object at 0x...>
>>> print windll.kernel32.MyOwnFunction
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

注意：Win32 系统的动态库，比如 `kernel32` 和 `user32`，通常会同时导出同一个函数的 ANSI 版本和 UNICODE 版本。UNICODE 版本通常会在名字最后以 `W` 结尾，而 ANSI 版本的则以 `A` 结尾。win32 的 `GetModuleHandle` 函数会根据一个模块名返回一个 *模块句柄*，该函数暨同时包含这样的两个版本的原型函数，并通过宏 UNICODE 是否定义，来决定宏 `GetModuleHandle` 导出的是哪个具体函数。

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

windll does not try to select one of them by magic, you must access the version you need by specifying `GetModuleHandleA` or `GetModuleHandleW` explicitly, and then call it with strings or unicode strings respectively.

有时候，dlls 的导出的函数名不符合 Python 的标识符规范，比如 `"??2@YAPAXI@Z"`。此时，你必须使用 *`getattr()`* 方法来获得该函数。

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

Windows 下，有些 dll 导出的函数没有函数名，而是通过其顺序号调用。对此类函数，你也可以通过 dll 对象的数值索引来操作这些函数。

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

调用函数

你可以貌似是调用其它 Python 函数那样直接调用这些函数。在这个例子中，我们调用了 time() 函数，该函数返回一个系统时间戳（从 Unix 时间起点到现在的秒数），而 “GetModuleHandleA()“ 函数返回一个 win32 模块句柄。

This example calls both functions with a NULL pointer (None should be used as the NULL pointer):

```
>>> print libc.time(None)
1150640792
>>> print hex(windll.kernel32.GetModuleHandleA(None))
0x1d000000
>>>
```

ctypes tries to protect you from calling functions with the wrong number of arguments or the wrong calling convention. Unfortunately this only works on Windows. It does this by examining the stack after the function returns, so although an error is raised the function *has* been called:

```
>>> windll.kernel32.GetModuleHandleA()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>> windll.kernel32.GetModuleHandleA(0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

The same exception is raised when you call an stdcall function with the cdecl calling convention, or vice versa:

```
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf("spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

你必须阅读这些库的头文件或说明文档来确定它们的正确的调用协议。

在 Windows 中，*ctypes* 使用 win32 结构化异常处理来防止由于在调用函数时使用非法参数导致的程序崩溃。

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
WindowsError: exception: access violation reading 0x00000020
>>>
```

There are, however, enough ways to crash Python with *ctypes*, so you should be careful anyway.

None, integers, longs, byte strings and unicode strings are the only native Python objects that can directly be used as parameters in these function calls. None is passed as a C NULL pointer, byte strings and unicode strings are passed as pointer to the memory block that contains their data (char * or wchar_t *). Python integers and Python longs are passed as the platforms default C int type, their value is masked to fit into the C type.

在我们开始调用函数前，我们必须先了解作为函数参数的*ctypes* 数据类型。

基础数据类型

ctypes 定义了一些和 C 兼容的基本数据类型：

ctypes 类型	C 类型	Python 类型
c_bool	_Bool	bool (1)
c_char	char	单字符字符串
c_wchar	wchar_t	1-character unicode string
c_byte	char	int/long
c_ubyte	unsigned char	int/long
c_short	short	int/long
c_ushort	unsigned short	int/long
c_int	int	int/long
c_uint	unsigned int	int/long
c_long	long	int/long
c_ulong	unsigned long	int/long
c_longlong	__int64 或 long long	int/long
c_ulonglong	unsigned __int64 或 unsigned long long	int/long
c_float	float	浮点数
c_double	double	浮点数
c_longdouble	long double	浮点数
c_char_p	char * (NUL terminated)	字符串或 None
c_wchar_p	wchar_t * (NUL terminated)	unicode or None
c_void_p	void *	int/long or None

(1) 构造函数接受任何具有真值的对象。

所有这些类型都可以通过使用正确类型和值的可选初始值调用它们来创建：

```
>>> c_int()
c_long(0)
>>> c_char_p("Hello, World")
c_char_p('Hello, World')
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

由于这些类型是可变的，它们的值也可以在以后更改：

```
>>> i = c_int(42)
>>> print i
c_long(42)
>>> print i.value
42
>>> i.value = -99
>>> print i.value
-99
>>>
```

Assigning a new value to instances of the pointer types *c_char_p*, *c_wchar_p*, and *c_void_p* changes the *memory location* they point to, *not the contents* of the memory block (of course not, because Python strings are immutable):

```
>>> s = "Hello, World"
>>> c_s = c_char_p(s)
>>> print c_s
c_char_p('Hello, World')
>>> c_s.value = "Hi, there"
>>> print c_s
c_char_p('Hi, there')
>>> print s                 # first string is unchanged
Hello, World
>>>
```

但你要注意不能将它们传递给会改变指针所指内存的函数。如果你需要可改变的内存块，ctypes 提供了*create_string_buffer()* 函数，它提供多种方式创建这种内存块。当前的内存块内容可以通过 raw 属性存取，如果你希望将它作为 NUL 结束的字符串，请使用 value 属性:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)      # create a 3 byte buffer, initialized to NUL␣
↪bytes
>>> print sizeof(p), repr(p.raw)
3 '\x00\x00\x00'
>>> p = create_string_buffer("Hello")      # create a buffer containing a NUL␣
↪terminated string
>>> print sizeof(p), repr(p.raw)
6 'Hello\x00'
>>> print repr(p.value)
'Hello'
>>> p = create_string_buffer("Hello", 10)  # create a 10 byte buffer
>>> print sizeof(p), repr(p.raw)
10 'Hello\x00\x00\x00\x00\x00'
>>> p.value = "Hi"
>>> print sizeof(p), repr(p.raw)
10 'Hi\x00lo\x00\x00\x00\x00\x00'
>>>
```

create_string_buffer() 函数替代以前的 ctypes 版本中的 c_buffer() 函数 (仍然可当作别名使用) 和 c_string() 函数。*create_unicode_buffer()* 函数创建包含 unicode 字符的可变内存块，与之对应的 C 语言类型是 wchar_t。

调用函数，继续

注意 printf 将打印到真正标准输出设备，而 * 不是 * *sys.stdout*，因此这些实例只能在控制台提示符下工作，而不能在 *IDLE* 或 *PythonWin* 中运行。

```
>>> printf = libc.printf
>>> printf("Hello, %s\n", "World!")
Hello, World!
14
>>> printf("Hello, %S\n", u"World!")
Hello, World!
14
>>> printf("%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf("%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(下页继续)

(续上页)

```
ArgumentError: argument 2: exceptions.TypeError: Don't know how to convert parameter 2
>>>
```

As has been mentioned before, all Python types except integers, strings, and unicode strings have to be wrapped in their corresponding *ctypes* type, so that they can be converted to the required C data type:

```
>>> printf("An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

使用自定义的数据类型调用函数

You can also customize *ctypes* argument conversion to allow instances of your own classes be used as function arguments. *ctypes* looks for an _as_parameter_ attribute and uses this as the function argument. Of course, it must be one of integer, string, or unicode:

```
>>> class Bottles(object):
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf("%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

If you don't want to store the instance's data in the _as_parameter_ instance variable, you could define a *property()* which makes the data available.

Specifying the required argument types (function prototypes)

It is possible to specify the required argument types of functions exported from DLLs by setting the `argtypes` attribute.

`argtypes` must be a sequence of C data types (the `printf` function is probably not a good example here, because it takes a variable number and different types of parameters depending on the format string, on the other hand this is quite handy to experiment with this feature):

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf("String '%s', Int %d, Double %f\n", "Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

Specifying a format protects against incompatible argument types (just as a prototype for a C function), and tries to convert the arguments to valid types:

```
>>> printf("%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf("%s %d %f\n", "X", 2, 3)
X 2 3.000000
```

(下页继续)

(续上页)

```
13
>>>
```

If you have defined your own classes which you pass to function calls, you have to implement a `from_param()` class method for them to be able to use them in the `argtypes` sequence. The `from_param()` class method receives the Python object passed to the function call, it should do a typecheck or whatever is needed to make sure this object is acceptable, and then return the object itself, its `_as_parameter_` attribute, or whatever you want to pass as the C function argument in this case. Again, the result should be an integer, string, unicode, a *ctypes* instance, or an object with an `_as_parameter_` attribute.

Return types

By default functions are assumed to return the C `int` type. Other return types can be specified by setting the `restype` attribute of the function object.

Here is a more advanced example, it uses the `strchr` function, which expects a string pointer and a char, and returns a pointer to a string:

```
>>> strchr = libc.strchr
>>> strchr("abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p   # c_char_p is a pointer to a string
>>> strchr("abcdef", ord("d"))
'def'
>>> print strchr("abcdef", ord("x"))
None
>>>
```

If you want to avoid the `ord("x")` calls above, you can set the `argtypes` attribute, and the second argument will be converted from a single character Python string into a C char:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr("abcdef", "d")
'def'
>>> strchr("abcdef", "def")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print strchr("abcdef", "x")
None
>>> strchr("abcdef", "d")
'def'
>>>
```

You can also use a callable Python object (a function or a class for example) as the `restype` attribute, if the foreign function returns an integer. The callable will be called with the *integer* the C function returns, and the result of this call will be used as the result of your function call. This is useful to check for error return values and automatically raise an exception:

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
```

(下页继续)

(续上页)

```
...
>>>
>>> GetModuleHandle.restype = ValidHandle
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
WindowsError: [Errno 126] The specified module could not be found.
>>>
```

`WinError` is a function which will call Windows `FormatMessage()` api to get the string representation of an error code, and *returns* an exception. `WinError` takes an optional error code parameter, if no one is used, it calls *`GetLastError()`* to retrieve it.

Please note that a much more powerful error checking mechanism is available through the `errcheck` attribute; see the reference manual for details.

Passing pointers (or: passing parameters by reference)

Sometimes a C api function expects a *pointer* to a data type as parameter, probably to write into the corresponding location, or if the data is too large to be passed by value. This is also known as *passing parameters by reference*.

`ctypes` exports the *`byref()`* function which is used to pass parameters by reference. The same effect can be achieved with the *`pointer()`* function, although *`pointer()`* does a lot more work since it constructs a real pointer object, so it is faster to use *`byref()`* if you don't need the pointer object in Python itself:

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer('\000' * 32)
>>> print i.value, f.value, repr(s.value)
0 0.0 ''
>>> libc.sscanf("1 3.14 Hello", "%d %f %s",
...             byref(i), byref(f), s)
3
>>> print i.value, f.value, repr(s.value)
1 3.1400001049 'Hello'
>>>
```

Structures and unions

Structures and unions must derive from the *`Structure`* and *`Union`* base classes which are defined in the *`ctypes`* module. Each subclass must define a `_fields_` attribute. `_fields_` must be a list of *2-tuples*, containing a *field name* and a *field type*.

The field type must be a *`ctypes`* type like *`c_int`*, or any other derived *`ctypes`* type: structure, union, array, pointer.

Here is a simple example of a POINT structure, which contains two integers named *x* and *y*, and also shows how to initialize a structure in the constructor:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
```

(下页继续)

(续上页)

```
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print point.x, point.y
10 20
>>> point = POINT(y=5)
>>> print point.x, point.y
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: too many initializers
>>>
```

You can, however, build much more complicated structures. A structure can itself contain other structures by using a structure as a field type.

Here is a RECT structure which contains two POINTs named *upperleft* and *lowerright*:

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print rc.upperleft.x, rc.upperleft.y
0 5
>>> print rc.lowerright.x, rc.lowerright.y
0 0
>>>
```

Nested structures can also be initialized in the constructor in several ways:

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

Field *descriptor*s can be retrieved from the *class*, they are useful for debugging because they can provide useful information:

```
>>> print POINT.x
<Field type=c_long, ofs=0, size=4>
>>> print POINT.y
<Field type=c_long, ofs=4, size=4>
>>>
```

警告： *ctypes* does not support passing unions or structures with bit-fields to functions by value. While this may work on 32-bit x86, it's not guaranteed by the library to work in the general case. Unions and structures with bit-fields should always be passed to functions by pointer.

Structure/union alignment and byte order

By default, Structure and Union fields are aligned in the same way the C compiler does it. It is possible to override this behavior be specifying a _pack_ class attribute in the subclass definition. This must be set to a positive integer and specifies the maximum alignment for the fields. This is what #pragma pack(n) also does in MSVC.

ctypes uses the native byte order for Structures and Unions. To build structures with non-native byte order, you can use one of the *BigEndianStructure*, *LittleEndianStructure*, BigEndianUnion, and LittleEndianUnion base classes. These classes cannot contain pointer fields.

Bit fields in structures and unions

It is possible to create structures and unions containing bit fields. Bit fields are only possible for integer fields, the bit width is specified as the third item in the _fields_ tuples:

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print Int.first_16
<Field type=c_long, ofs=0:0, bits=16>
>>> print Int.second_16
<Field type=c_long, ofs=0:16, bits=16>
>>>
```

Arrays

Arrays are sequences, containing a fixed number of instances of the same type.

The recommended way to create array types is by multiplying a data type with a positive integer:

```
TenPointsArrayType = POINT * 10
```

Here is an example of a somewhat artificial data type, a structure containing 4 POINTs among other stuff:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print len(MyStruct().point_array)
4
>>>
```

Instances are created in the usual way, by calling the class:

```
arr = TenPointsArrayType()
for pt in arr:
    print pt.x, pt.y
```

The above code print a series of 0 0 lines, because the array contents is initialized to zeros.

Initializers of the correct type can also be specified:

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print ii
<c_long_Array_10 object at 0x...>
>>> for i in ii: print i,
...
1 2 3 4 5 6 7 8 9 10
>>>
```

Pointers

Pointer instances are created by calling the *pointer()* function on a *ctypes* type:

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

Pointer instances have a *contents* attribute which returns the object to which the pointer points, the `i` object above:

```
>>> pi.contents
c_long(42)
>>>
```

Note that *ctypes* does not have OOR (original object return), it constructs a new, equivalent object each time you retrieve an attribute:

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

Assigning another *c_int* instance to the pointer's contents attribute would cause the pointer to point to the memory location where this is stored:

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

Pointer instances can also be indexed with integers:

```
>>> pi[0]
99
>>>
```

Assigning to an integer index changes the pointed to value:

```
>>> print i
c_long(99)
>>> pi[0] = 22
```

(下页继续)

(续上页)

```
>>> print i
c_long(22)
>>>
```

It is also possible to use indexes different from 0, but you must know what you' re doing, just as in C: You can access or change arbitrary memory locations. Generally you only use this feature if you receive a pointer from a C function, and you *know* that the pointer actually points to an array instead of a single item.

Behind the scenes, the *pointer()* function does more than simply create pointer instances, it has to create pointer *types* first. This is done with the *POINTER()* function, which accepts any *ctypes* type, and returns a new type:

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

Calling the pointer type without an argument creates a `NULL` pointer. `NULL` pointers have a `False` boolean value:

```
>>> null_ptr = POINTER(c_int)()
>>> print bool(null_ptr)
False
>>>
```

ctypes checks for `NULL` when dereferencing pointers (but dereferencing invalid non-`NULL` pointers would crash Python):

```
>>> null_ptr[0]
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>
```

Type conversions

Usually, ctypes does strict type checking. This means, if you have `POINTER(c_int)` in the `argtypes` list of a function or as the type of a member field in a structure definition, only instances of exactly the same type are accepted. There are some exceptions to this rule, where ctypes accepts other objects. For example, you can pass compatible array instances instead of pointer types. So, for `POINTER(c_int)`, ctypes accepts an array of c_int:

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
```

(下页继续)

(续上页)

```
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print bar.values[i]
...
1
2
3
>>>
```

In addition, if a function argument is explicitly declared to be a pointer type (such as `POINTER(c_int)`) in `argtypes`, an object of the pointed type (`c_int` in this case) can be passed to the function. ctypes will apply the required *byref()* conversion in this case automatically.

To set a POINTER type field to `NULL`, you can assign `None`:

```
>>> bar.values = None
>>>
```

Sometimes you have instances of incompatible types. In C, you can cast one type into another type. *ctypes* provides a *cast()* function which can be used in the same way. The `Bar` structure defined above accepts `POINTER(c_int)` pointers or *c_int* arrays for its `values` field, but not instances of other types:

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long instance
>>>
```

For these cases, the *cast()* function is handy.

The *cast()* function can be used to cast a ctypes instance into a pointer to a different ctypes data type. *cast()* takes two parameters, a ctypes object that is or can be converted to a pointer of some kind, and a ctypes pointer type. It returns an instance of the second argument, which references the same memory block as the first argument:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

So, *cast()* can be used to assign to the `values` field of `Bar` the structure:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print bar.values[0]
0
>>>
```

Incomplete Types

Incomplete Types are structures, unions or arrays whose members are not yet specified. In C, they are specified by forward declarations, which are defined later:

```
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

The straightforward translation into ctypes code would be this, but it does not work:

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

because the new `class cell` is not available in the class statement itself. In *ctypes*, we can define the `cell` class and set the `_fields_` attribute later, after the class statement:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

Lets try it. We create two instances of `cell`, and let them point to each other, and finally follow the pointer chain a few times:

```
>>> c1 = cell()
>>> c1.name = "foo"
>>> c2 = cell()
>>> c2.name = "bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print p.name,
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

Callback functions

ctypes allows creating C callable function pointers from Python callables. These are sometimes called *callback functions*.

First, you must create a class for the callback function, the class knows the calling convention, the return type, and the number and types of arguments this function will receive.

The CFUNCTYPE factory function creates types for callback functions using the normal cdecl calling convention, and, on Windows, the WINFUNCTYPE factory function creates types for callback functions using the stdcall calling convention.

Both of these factory functions are called with the result type as first argument, and the callback functions expected argument types as the remaining arguments.

I will present an example here which uses the standard C library' s `qsort()` function, this is used to sort items with the help of a callback function. `qsort()` will be used to sort an array of integers:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

`qsort()` must be called with a pointer to the data to sort, the number of items in the data array, the size of one item, and a pointer to the comparison function, the callback. The callback will then be called with two pointers to items, and it must return a negative integer if the first item is smaller than the second, a zero if they are equal, and a positive integer else.

So our callback function receives pointers to integers, and must return an integer. First we create the `type` for the callback function:

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

For the first implementation of the callback function, we simply print the arguments we get, and return 0 (incremental development ;-):

```
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a, b
...     return 0
...
>>>
```

Create the C callable callback:

```
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

And we' re ready to go:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
```

(下页继续)

(续上页)

```
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
py_cmp_func <ctypes.LP_c_long object at 0x00...> <ctypes.LP_c_long object at 0x00...>
>>>
```

We know how to access the contents of a pointer, so lets redefine our callback:

```
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a[0], b[0]
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

Here is what we get on Windows:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 7 1
py_cmp_func 33 1
py_cmp_func 99 1
py_cmp_func 5 1
py_cmp_func 7 5
py_cmp_func 33 5
py_cmp_func 99 5
py_cmp_func 7 99
py_cmp_func 33 99
py_cmp_func 7 33
>>>
```

It is funny to see that on linux the sort function seems to work much more efficiently, it is doing less comparisons:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

Ah, we’re nearly done! The last step is to actually compare the two items and return a useful result:

```
>>> def py_cmp_func(a, b):
...     print "py_cmp_func", a[0], b[0]
...     return a[0] - b[0]
...
>>>
```

Final run on Windows:

```
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 33 7
py_cmp_func 99 33
py_cmp_func 5 99
py_cmp_func 1 99
py_cmp_func 33 7
py_cmp_func 1 33
```

(下页继续)

(续上页)

```
py_cmp_func 5 33
py_cmp_func 5 7
py_cmp_func 1 7
py_cmp_func 5 1
>>>
```

and on Linux:

```
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

It is quite interesting to see that the Windows `qsort()` function needs more comparisons than the linux version!

As we can easily check, our array is sorted now:

```
>>> for i in ia: print i,
...
1 5 7 33 99
>>>
```

注解： Make sure you keep references to *CFUNCTYPE()* objects as long as they are used from C code. *ctypes* doesn't, and if you don't, they may be garbage collected, crashing your program when a callback is made.

Also, note that if the callback function is called in a thread created outside of Python's control (e.g. by the foreign code that calls the callback), ctypes creates a new dummy Python thread on every invocation. This behavior is correct for most purposes, but it means that values stored with *threading.local* will *not* survive across different callbacks, even when those calls are made from the same C thread.

Accessing values exported from dlls

Some shared libraries not only export functions, they also export variables. An example in the Python library itself is the `Py_OptimizeFlag`, an integer set to 0, 1, or 2, depending on the `-O` or `-OO` flag given on startup.

ctypes can access values like this with the `in_dll()` class methods of the type. *pythonapi* is a predefined symbol giving access to the Python C api:

```
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print opt_flag
c_long(0)
>>>
```

If the interpreter would have been started with `-O`, the sample would have printed `c_long(1)`, or `c_long(2)` if `-OO` would have been specified.

An extended example which also demonstrates the use of pointers accesses the `PyImport_FrozenModules` pointer exported by Python.

Quoting the Python docs: *This pointer is initialized to point to an array of "struct _frozen" records, terminated by one whose members are all NULL or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.*

So manipulating this pointer could even prove useful. To restrict the example size, we show only how this table can be read with *ctypes*:

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int)]
...
>>>
```

We have defined the `struct _frozen` data type, so we can get the pointer to the table:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")
>>>
```

Since `table` is a `pointer` to the array of `struct_frozen` records, we can iterate over it, but we just have to make sure that our loop terminates, because pointers have no size. Sooner or later it would probably crash with an access violation or whatever, so it's better to break out of the loop when we hit the NULL entry:

```
>>> for item in table:
...     print item.name, item.size
...     if item.name is None:
...         break
...
__hello__ 104
__phello__ -104
__phello__.spam 104
None 0
>>>
```

The fact that standard Python has a frozen module and a frozen package (indicated by the negative size member) is not well known, it is only used for testing. Try it out with `import __hello__` for example.

Surprises

There are some edge cases in *ctypes* where you might expect something other than what actually happens.

Consider the following example:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print rc.a.x, rc.a.y, rc.b.x, rc.b.y
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print rc.a.x, rc.a.y, rc.b.x, rc.b.y
```

(下页继续)

(续上页)

```
3 4 3 4
>>>
```

Hm. We certainly expected the last statement to print `3 4 1 2`. What happened? Here are the steps of the `rc.a, rc.b = rc.b, rc.a` line above:

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

Note that `temp0` and `temp1` are objects still using the internal buffer of the `rc` object above. So executing `rc.a = temp0` copies the buffer contents of `temp0` into `rc` 's buffer. This, in turn, changes the contents of `temp1`. So, the last assignment `rc.b = temp1`, doesn't have the expected effect.

Keep in mind that retrieving sub-objects from Structure, Unions, and Arrays doesn't *copy* the sub-object, instead it retrieves a wrapper object accessing the root-object's underlying buffer.

Another example that may behave different from what one would expect is this:

```
>>> s = c_char_p()
>>> s.value = "abc def ghi"
>>> s.value
'abc def ghi'
>>> s.value is s.value
False
>>>
```

Why is it printing `False`? ctypes instances are objects containing a memory block plus some *descriptor*s accessing the contents of the memory. Storing a Python object in the memory block does not store the object itself, instead the `contents` of the object is stored. Accessing the contents again constructs a new Python object each time!

Variable-sized data types

ctypes provides some support for variable-sized arrays and structures.

The *resize()* function can be used to resize the memory buffer of an existing ctypes object. The function takes the object as first argument, and the requested size in bytes as the second argument. The memory block cannot be made smaller than the natural memory block specified by the objects type, a *ValueError* is raised if this is tried:

```
>>> short_array = (c_short * 4)()
>>> print sizeof(short_array)
8
>>> resize(short_array, 4)
Traceback (most recent call last):
    ...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

This is nice and fine, but how would one access the additional elements contained in this array? Since the type still only knows about 4 elements, we get errors accessing other elements:

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
    ...
IndexError: invalid index
>>>
```

Another way to use variable-sized data types with *ctypes* is to use the dynamic nature of Python, and (re-)define the data type after the required size is already known, on a case by case basis.

15.17.2 ctypes reference

Finding shared libraries

When programming in a compiled language, shared libraries are accessed when compiling/linking a program, and when the program is run.

The purpose of the `find_library()` function is to locate a library in a way similar to what the compiler does (on platforms with several versions of a shared library the most recent should be loaded), while the ctypes library loaders act like when a program is run, and call the runtime loader directly.

The `ctypes.util` module provides a function which can help to determine the library to load.

`ctypes.util.`**`find_library`**(*name*)

Try to find a library and return a pathname. *name* is the library name without any prefix like *lib*, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

The exact functionality is system dependent.

On Linux, `find_library()` tries to run external programs (`/sbin/ldconfig`, `gcc`, and `objdump`) to find the library file. It returns the filename of the library file. Here are some examples:

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

On OS X, `find_library()` tries several predefined naming schemes and paths to locate the library, and returns a full pathname if successful:

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

On Windows, `find_library()` searches along the system search path, and returns the full pathname, but since there is no predefined naming scheme a call like `find_library("c")` will fail and return `None`.

If wrapping a shared library with *ctypes*, it *may* be better to determine the shared library name at development time, and hardcode that into the wrapper module instead of using `find_library()` to locate the library at runtime.

Loading shared libraries

There are several ways to load shared libraries into the Python process. One way is to instantiate one of the following classes:

class ctypes.**CDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*)

Instances of this class represent loaded shared libraries. Functions in these libraries use the standard C calling convention, and are assumed to return `int`.

class ctypes.**OleDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*)

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return the windows specific *HRESULT* code. *HRESULT* values contain information specifying whether the function call failed or succeeded, together with additional error code. If the return value signals a failure, an `WindowsError` is automatically raised.

class ctypes.**WinDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*)

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return `int` by default.

On Windows CE only the standard calling convention is used, for convenience the *WinDLL* and *OleDLL* use the standard calling convention on this platform.

The Python *global interpreter lock* is released before calling any function exported by these libraries, and reacquired afterwards.

class ctypes.**PyDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*)

Instances of this class behave like *CDLL* instances, except that the Python GIL is *not* released during the function call, and after the function execution the Python error flag is checked. If the error flag is set, a Python exception is raised.

Thus, this is only useful to call Python C api functions directly.

All these classes can be instantiated by calling them with at least one argument, the pathname of the shared library. If you have an existing handle to an already loaded shared library, it can be passed as the `handle` named parameter, otherwise the underlying platforms `dlopen` or `LoadLibrary` function is used to load the library into the process, and to get a handle to it.

The *mode* parameter can be used to specify how the library is loaded. For details, consult the *dlopen(3)* manpage. On Windows, *mode* is ignored. On posix systems, RTLD_NOW is always added, and is not configurable.

The *use_errno* parameter, when set to true, enables a ctypes mechanism that allows accessing the system *errno* error number in a safe way. *ctypes* maintains a thread-local copy of the systems *errno* variable; if you call foreign functions created with `use_errno=True` then the *errno* value before the function call is swapped with the ctypes private copy, the same happens immediately after the function call.

The function *ctypes.get_errno()* returns the value of the ctypes private copy, and the function *ctypes.set_errno()* changes the ctypes private copy to a new value and returns the former value.

The *use_last_error* parameter, when set to true, enables the same mechanism for the Windows error code which is managed by the *GetLastError()* and `SetLastError()` Windows API functions; *ctypes.get_last_error()* and *ctypes.set_last_error()* are used to request and change the ctypes private copy of the windows error code.

2.6 新版功能: The *use_last_error* and *use_errno* optional parameters were added.

ctypes.**RTLD_GLOBAL**
Flag to use as *mode* parameter. On platforms where this flag is not available, it is defined as the integer zero.

ctypes.**RTLD_LOCAL**
Flag to use as *mode* parameter. On platforms where this is not available, it is the same as *RTLD_GLOBAL*.

ctypes.**DEFAULT_MODE**
The default mode which is used to load shared libraries. On OSX 10.3, this is *RTLD_GLOBAL*, otherwise it is the same as *RTLD_LOCAL*.

Instances of these classes have no public methods. Functions exported by the shared library can be accessed as attributes or by index. Please note that accessing the function through an attribute caches the result and therefore accessing it repeatedly returns the same object each time. On the other hand, accessing it through an index returns a new object each time:

```
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

The following public attributes are available, their name starts with an underscore to not clash with exported function names:

PyDLL.**_handle**
The system handle used to access the library.

PyDLL.**_name**
The name of the library passed in the constructor.

Shared libraries can also be loaded by using one of the prefabricated objects, which are instances of the *LibraryLoader* class, either by calling the `LoadLibrary()` method, or by retrieving the library as attribute of the loader instance.

class ctypes.**LibraryLoader**(*dlltype*)
Class which loads shared libraries. *dlltype* should be one of the *CDLL*, *PyDLL*, *WinDLL*, or *OleDLL* types.

`__getattr__()` has special behavior: It allows loading a shared library by accessing it as attribute of a library loader instance. The result is cached, so repeated attribute accesses return the same library each time.

LoadLibrary(*name*)
Load a shared library into the process and return it. This method always returns a new instance of the library.

These prefabricated library loaders are available:

ctypes.**cdll**
Creates *CDLL* instances.

ctypes.**windll**
仅 Windows 中: 创建*WinDLL* 实例.

ctypes.**oledll**
仅 Windows 中: 创建*OleDLL* 实例。

ctypes.**pydll**
创建*PyDLL* 实例。

For accessing the C Python api directly, a ready-to-use Python shared library object is available:

ctypes.**pythonapi**
An instance of *PyDLL* that exposes Python C API functions as attributes. Note that all these functions are assumed

to return C `int`, which is of course not always the truth, so you have to assign the correct `restype` attribute to use these functions.

Foreign functions

As explained in the previous section, foreign functions can be accessed as attributes of loaded shared libraries. The function objects created in this way by default accept any number of arguments, accept any ctypes data instances as arguments, and return the default result type specified by the library loader. They are instances of a private class:

class ctypes.**_FuncPtr**

Base class for C callable foreign functions.

Instances of foreign functions are also C compatible data types; they represent C function pointers.

This behavior can be customized by assigning to special attributes of the foreign function object.

restype

Assign a ctypes type to specify the result type of the foreign function. Use `None` for `void`, a function not returning anything.

It is possible to assign a callable Python object that is not a ctypes type, in this case the function is assumed to return a C `int`, and the callable will be called with this integer, allowing further processing or error checking. Using this is deprecated, for more flexible post processing or error checking use a ctypes data type as *restype* and assign a callable to the *errcheck* attribute.

argtypes

Assign a tuple of ctypes types to specify the argument types that the function accepts. Functions using the `stdcall` calling convention can only be called with the same number of arguments as the length of this tuple; functions using the C calling convention accept additional, unspecified arguments as well.

When a foreign function is called, each actual argument is passed to the `from_param()` class method of the items in the *argtypes* tuple, this method allows adapting the actual argument to an object that the foreign function accepts. For example, a *c_char_p* item in the *argtypes* tuple will convert a unicode string passed as argument into a byte string using ctypes conversion rules.

New: It is now possible to put items in argtypes which are not ctypes types, but each item must have a `from_param()` method which returns a value usable as argument (integer, string, ctypes instance). This allows defining adapters that can adapt custom objects as function parameters.

errcheck

Assign a Python function or another callable to this attribute. The callable will be called with three or more arguments:

callable(*result*, *func*, *arguments*)

result is what the foreign function returns, as specified by the *restype* attribute.

func is the foreign function object itself, this allows reusing the same callable object to check or post process the results of several functions.

arguments is a tuple containing the parameters originally passed to the function call, this allows specializing the behavior on the arguments used.

The object that this function returns will be returned from the foreign function call, but it can also check the result value and raise an exception if the foreign function call failed.

exception ctypes.**ArgumentError**

This exception is raised when a foreign function call cannot convert one of the passed arguments.

Function prototypes

Foreign functions can also be created by instantiating function prototypes. Function prototypes are similar to function prototypes in C; they describe a function (return type, argument types, calling convention) without defining an implementation. The factory functions must be called with the desired result type and the argument types of the function.

`ctypes.`**`CFUNCTYPE`**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)
The returned function prototype creates functions that use the standard C calling convention. The function will release the GIL during the call. If *use_errno* is set to true, the ctypes private copy of the system *errno* variable is exchanged with the real *errno* value before and after the call; *use_last_error* does the same for the Windows error code.

在 2.6 版更改: The optional *use_errno* and *use_last_error* parameters were added.

`ctypes.`**`WINFUNCTYPE`**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)
Windows only: The returned function prototype creates functions that use the `stdcall` calling convention, except on Windows CE where *WINFUNCTYPE()* is the same as *CFUNCTYPE()*. The function will release the GIL during the call. *use_errno* and *use_last_error* have the same meaning as above.

`ctypes.`**`PYFUNCTYPE`**(*restype*, **argtypes*)
The returned function prototype creates functions that use the Python calling convention. The function will *not* release the GIL during the call.

Function prototypes created by these factory functions can be instantiated in different ways, depending on the type and number of the parameters in the call:

`prototype`(*address*)
Returns a foreign function at the specified address which must be an integer.

`prototype`(*callable*)
Create a C callable function (a callback function) from a Python *callable*.

`prototype`(*func_spec*[, *paramflags*])
Returns a foreign function exported by a shared library. *func_spec* must be a 2-tuple `(name_or_ordinal, library)`. The first item is the name of the exported function as string, or the ordinal of the exported function as small integer. The second item is the shared library instance.

`prototype`(*vtbl_index*, *name*[, *paramflags*[, *iid*]])
Returns a foreign function that will call a COM method. *vtbl_index* is the index into the virtual function table, a small non-negative integer. *name* is name of the COM method. *iid* is an optional pointer to the interface identifier which is used in extended error reporting.

COM methods use a special calling convention: They require a pointer to the COM interface as first argument, in addition to those parameters that are specified in the `argtypes` tuple.

The optional *paramflags* parameter creates foreign function wrappers with much more functionality than the features described above.

paramflags must be a tuple of the same length as `argtypes`.

Each item in this tuple contains further information about a parameter, it must be a tuple containing one, two, or three items.

The first item is an integer containing a combination of direction flags for the parameter:

1 Specifies an input parameter to the function.

2 Output parameter. The foreign function fills in a value.

4 Input parameter which defaults to the integer zero.

The optional second item is the parameter name as string. If this is specified, the foreign function can be called with named parameters.

The optional third item is the default value for this parameter.

This example demonstrates how to wrap the Windows `MessageBoxA` function so that it supports default parameters and named arguments. The C declaration from the windows header file is this:

```
WINUSERAPI int WINAPI
MessageBoxA(
    HWND hWnd,
    LPCSTR lpText,
    LPCSTR lpCaption,
    UINT uType);
```

Here is the wrapping with *ctypes*:

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCSTR, LPCSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", None), (1, "flags",
↪ 0)
>>> MessageBox = prototype(("MessageBoxA", windll.user32), paramflags)
>>>
```

The MessageBox foreign function can now be called in these ways:

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
>>>
```

A second example demonstrates output parameters. The win32 `GetWindowRect` function retrieves the dimensions of a specified window by copying them into `RECT` structure that the caller has to supply. Here is the C declaration:

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
     HWND hWnd,
     LPRECT lpRect);
```

Here is the wrapping with *ctypes*:

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

Functions with output parameters will automatically return the output parameter value if there is a single one, or a tuple containing the output parameter values when there are more than one, so the GetWindowRect function now returns a RECT instance, when called.

Output parameters can be combined with the `errcheck` protocol to do further output processing and error checking. The win32 `GetWindowRect` api function returns a `BOOL` to signal success or failure, so this function could do the error checking, and raises an exception when the api call failed:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

If the `errcheck` function returns the argument tuple it receives unchanged, *ctypes* continues the normal processing it does on the output parameters. If you want to return a tuple of window coordinates instead of a `RECT` instance, you can retrieve the fields in the function and return them instead, the normal processing will no longer take place:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

Utility functions

`ctypes.`**`addressof`**(*obj*)
Returns the address of the memory buffer as integer. *obj* must be an instance of a ctypes type.

`ctypes.`**`alignment`**(*obj_or_type*)
Returns the alignment requirements of a ctypes type. *obj_or_type* must be a ctypes type or instance.

`ctypes.`**`byref`**(*obj*[, *offset*])
Returns a light-weight pointer to *obj*, which must be an instance of a ctypes type. *offset* defaults to zero, and must be an integer that will be added to the internal pointer value.

`byref(obj, offset)` corresponds to this C code:

```
(((char *)&obj) + offset)
```

The returned object can only be used as a foreign function call parameter. It behaves similar to `pointer(obj)`, but the construction is a lot faster.

2.6 新版功能: The *offset* optional argument was added.

`ctypes.`**`cast`**(*obj*, *type*)
This function is similar to the cast operator in C. It returns a new instance of *type* which points to the same memory block as *obj*. *type* must be a pointer type, and *obj* must be an object that can be interpreted as a pointer.

`ctypes.`**`create_string_buffer`**(*init_or_size*[, *size*])
This function creates a mutable character buffer. The returned object is a ctypes array of *c_char*.

init_or_size must be an integer which specifies the size of the array, or a string which will be used to initialize the array items.

If a string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the string should not be used.

If the first parameter is a unicode string, it is converted into an 8-bit string according to ctypes conversion rules.

ctypes.**create_unicode_buffer**(*init_or_size*[, *size*])
This function creates a mutable unicode character buffer. The returned object is a ctypes array of *c_wchar*.

init_or_size must be an integer which specifies the size of the array, or a unicode string which will be used to initialize the array items.

If a unicode string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the string should not be used.

If the first parameter is an 8-bit string, it is converted into a unicode string according to ctypes conversion rules.

ctypes.**DllCanUnloadNow**()
Windows only: This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllCanUnloadNow function that the _ctypes extension dll exports.

ctypes.**DllGetClassObject**()
Windows only: This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllGetClassObject function that the `_ctypes` extension dll exports.

ctypes.util.**find_library**(*name*)
Try to find a library and return a pathname. *name* is the library name without any prefix like `lib`, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

The exact functionality is system dependent.

在 2.6 版更改: Windows only: `find_library("m")` or `find_library("c")` return the result of a call to `find_msvcrt()`.

ctypes.util.**find_msvcrt**()
Windows only: return the filename of the VC runtime library used by Python, and by the extension modules. If the name of the library cannot be determined, `None` is returned.

If you need to free memory, for example, allocated by an extension module with a call to the `free(void *)`, it is important that you use the function in the same library that allocated the memory.

2.6 新版功能.

ctypes.**FormatError**([*code*])
Windows only: Returns a textual description of the error code *code*. If no error code is specified, the last error code is used by calling the Windows api function GetLastError.

ctypes.**GetLastError**()
Windows only: Returns the last error code set by Windows in the calling thread. This function calls the Windows *GetLastError()* function directly, it does not return the ctypes-private copy of the error code.

ctypes.**get_errno**()
Returns the current value of the ctypes-private copy of the system *errno* variable in the calling thread.

2.6 新版功能.

ctypes.**get_last_error**()
Windows only: returns the current value of the ctypes-private copy of the system `LastError` variable in the calling thread.

2.6 新版功能.

ctypes.**memmove**(*dst*, *src*, *count*)
Same as the standard C memmove library function: copies *count* bytes from *src* to *dst*. *dst* and *src* must be integers or ctypes instances that can be converted to pointers.

ctypes.**memset**(*dst*, *c*, *count*)
Same as the standard C memset library function: fills the memory block at address *dst* with *count* bytes of value *c*. *dst* must be an integer specifying an address, or a ctypes instance.

ctypes.**POINTER**(*type*)
This factory function creates and returns a new ctypes pointer type. Pointer types are cached and reused internally, so calling this function repeatedly is cheap. *type* must be a ctypes type.

ctypes.**pointer**(*obj*)
This function creates a new pointer instance, pointing to *obj*. The returned object is of the type `POINTER(type(obj))`.

Note: If you just want to pass a pointer to an object to a foreign function call, you should use `byref(obj)` which is much faster.

ctypes.**resize**(*obj*, *size*)
This function resizes the internal memory buffer of *obj*, which must be an instance of a ctypes type. It is not possible to make the buffer smaller than the native size of the objects type, as given by `sizeof(type(obj))`, but it is possible to enlarge the buffer.

ctypes.**set_conversion_mode**(*encoding*, *errors*)
This function sets the rules that ctypes objects use when converting between 8-bit strings and unicode strings. *encoding* must be a string specifying an encoding, like `'utf-8'` or `'mbcs'`, *errors* must be a string specifying the error handling on encoding/decoding errors. Examples of possible values are `"strict"`, `"replace"`, or `"ignore"`.

set_conversion_mode() returns a 2-tuple containing the previous conversion rules. On windows, the initial conversion rules are `('mbcs', 'ignore')`, on other systems `('ascii', 'strict')`.

ctypes.**set_errno**(*value*)
Set the current value of the ctypes-private copy of the system *errno* variable in the calling thread to *value* and return the previous value.

2.6 新版功能.

ctypes.**set_last_error**(*value*)
Windows only: set the current value of the ctypes-private copy of the system `LastError` variable in the calling thread to *value* and return the previous value.

2.6 新版功能.

ctypes.**sizeof**(*obj_or_type*)
Returns the size in bytes of a ctypes type or instance memory buffer. Does the same as the C `sizeof` operator.

ctypes.**string_at**(*address*[, *size*])
This function returns the string starting at memory address *address*. If size is specified, it is used as size, otherwise the string is assumed to be zero-terminated.

ctypes.**WinError**(*code=None*, *descr=None*)
Windows only: this function is probably the worst-named thing in ctypes. It creates an instance of WindowsError. If *code* is not specified, `GetLastError` is called to determine the error code. If `descr` is not specified, *FormatError()* is called to get a textual description of the error.

ctypes.**wstring_at**(*address*[, *size*])
This function returns the wide character string starting at memory address *address* as unicode string. If *size* is specified, it is used as the number of characters of the string, otherwise the string is assumed to be zero-terminated.

Data types

class ctypes.**_CData**

This non-public class is the common base class of all ctypes data types. Among other things, all ctypes type instances contain a memory block that hold C compatible data; the address of the memory block is returned by the *addressof()* helper function. Another instance variable is exposed as *_objects*; this contains other Python objects that need to be kept alive in case the memory block contains pointers.

Common methods of ctypes data types, these are all class methods (to be exact, they are methods of the *metaclass*):

from_buffer(*source*[, *offset*])

This method returns a ctypes instance that shares the buffer of the *source* object. The *source* object must support the writeable buffer interface. The optional *offset* parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a *ValueError* is raised.

2.6 新版功能.

from_buffer_copy(*source*[, *offset*])

This method creates a ctypes instance, copying the buffer from the *source* object buffer which must be readable. The optional *offset* parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a *ValueError* is raised.

2.6 新版功能.

from_address(*address*)

This method returns a ctypes type instance using the memory specified by *address* which must be an integer.

from_param(*obj*)

This method adapts *obj* to a ctypes type. It is called with the actual object used in a foreign function call when the type is present in the foreign function's `argtypes` tuple; it must return an object that can be used as a function call parameter.

All ctypes data types have a default implementation of this classmethod that normally returns *obj* if that is an instance of the type. Some types accept other objects as well.

in_dll(*library*, *name*)

This method returns a ctypes type instance exported by a shared library. *name* is the name of the symbol that exports the data, *library* is the loaded shared library.

Common instance variables of ctypes data types:

_b_base_

Sometimes ctypes data instances do not own the memory block they contain, instead they share part of the memory block of a base object. The *_b_base_* read-only member is the root ctypes object that owns the memory block.

_b_needsfree_

This read-only variable is true when the ctypes data instance has allocated the memory block itself, false otherwise.

_objects

This member is either `None` or a dictionary containing Python objects that need to be kept alive so that the memory block contents is kept valid. This object is only exposed for debugging; never modify the contents of this dictionary.

基础数据类型

class ctypes.**_SimpleCData**
This non-public class is the base class of all fundamental ctypes data types. It is mentioned here because it contains the common attributes of the fundamental ctypes data types. *_SimpleCData* is a subclass of *_CData*, so it inherits their methods and attributes.

在 2.6 版更改: ctypes data types that are not and do not contain pointers can now be pickled.

Instances have a single attribute:

value
This attribute contains the actual value of the instance. For integer and pointer types, it is an integer, for character types, it is a single character string, for character pointer types it is a Python string or unicode string.

When the `value` attribute is retrieved from a ctypes instance, usually a new object is returned each time. *ctypes* does *not* implement original object return, always a new object is constructed. The same is true for all other ctypes object instances.

Fundamental data types, when returned as foreign function call results, or, for example, by retrieving structure field members or array items, are transparently converted to native Python types. In other words, if a foreign function has a `restype` of *c_char_p*, you will always receive a Python string, *not* a *c_char_p* instance.

Subclasses of fundamental data types do *not* inherit this behavior. So, if a foreign functions `restype` is a subclass of *c_void_p*, you will receive an instance of this subclass from the function call. Of course, you can get the value of the pointer by accessing the `value` attribute.

These are the fundamental ctypes data types:

class ctypes.**c_byte**
Represents the C `signed char` datatype, and interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_char**
Represents the C `char` datatype, and interprets the value as a single character. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.**c_char_p**
Represents the C `char *` datatype when it points to a zero-terminated string. For a general character pointer that may also point to binary data, `POINTER(c_char)` must be used. The constructor accepts an integer address, or a string.

class ctypes.**c_double**
Represents the C `double` datatype. The constructor accepts an optional float initializer.

class ctypes.**c_longdouble**
Represents the C `long double` datatype. The constructor accepts an optional float initializer. On platforms where `sizeof(long double) == sizeof(double)` it is an alias to *c_double*.

2.6 新版功能.

class ctypes.**c_float**
Represents the C `float` datatype. The constructor accepts an optional float initializer.

class ctypes.**c_int**
Represents the C `signed int` datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias to *c_long*.

class ctypes.**c_int8**
Represents the C 8-bit `signed int` datatype. Usually an alias for *c_byte*.

class ctypes.**c_int16**
Represents the C 16-bit `signed int` datatype. Usually an alias for *c_short*.

class ctypes.**c_int32**
Represents the C 32-bit `signed int` datatype. Usually an alias for *c_int*.

class ctypes.**c_int64**
Represents the C 64-bit `signed int` datatype. Usually an alias for *c_longlong*.

class ctypes.**c_long**
Represents the C `signed long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_longlong**
Represents the C `signed long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_short**
Represents the C `signed short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_size_t**
Represents the C `size_t` datatype.

class ctypes.**c_ssize_t**
Represents the C `ssize_t` datatype.

2.7 新版功能.

class ctypes.**c_ubyte**
Represents the C `unsigned char` datatype, it interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_uint**
Represents the C `unsigned int` datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias for *c_ulong*.

class ctypes.**c_uint8**
Represents the C 8-bit `unsigned int` datatype. Usually an alias for *c_ubyte*.

class ctypes.**c_uint16**
Represents the C 16-bit `unsigned int` datatype. Usually an alias for *c_ushort*.

class ctypes.**c_uint32**
Represents the C 32-bit `unsigned int` datatype. Usually an alias for *c_uint*.

class ctypes.**c_uint64**
Represents the C 64-bit `unsigned int` datatype. Usually an alias for *c_ulonglong*.

class ctypes.**c_ulong**
Represents the C `unsigned long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_ulonglong**
Represents the C `unsigned long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_ushort**
Represents the C `unsigned short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_void_p**
Represents the C `void *` type. The value is represented as integer. The constructor accepts an optional integer initializer.

class ctypes.**c_wchar**
Represents the C `wchar_t` datatype, and interprets the value as a single character unicode string. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.**c_wchar_p**
Represents the C `wchar_t *` datatype, which must be a pointer to a zero-terminated wide character string. The constructor accepts an integer address, or a string.

class ctypes.**c_bool**
Represent the C `bool` datatype (more accurately, `_Bool` from C99). Its value can be `True` or `False`, and the constructor accepts any object that has a truth value.

2.6 新版功能.

class ctypes.**HRESULT**
Windows only: Represents a `HRESULT` value, which contains success or error information for a function or method call.

class ctypes.**py_object**
Represents the C `PyObject *` datatype. Calling this without an argument creates a `NULL PyObject *` pointer.

The `ctypes.wintypes` module provides quite some other Windows specific data types, for example `HWND`, `WPARAM`, or `DWORD`. Some useful structures like `MSG` or `RECT` are also defined.

Structured data types

class ctypes.**Union**(**args*, ***kw*)
Abstract base class for unions in native byte order.

class ctypes.**BigEndianStructure**(**args*, ***kw*)
Abstract base class for structures in *big endian* byte order.

class ctypes.**LittleEndianStructure**(**args*, ***kw*)
Abstract base class for structures in *little endian* byte order.

Structures with non-native byte order cannot contain pointer type fields, or any other data types containing pointer type fields.

class ctypes.**Structure**(**args*, ***kw*)
Abstract base class for structures in *native* byte order.

Concrete structure and union types must be created by subclassing one of these types, and at least define a *_fields_* class variable. *ctypes* will create *descriptor*s which allow reading and writing the fields by direct attribute accesses. These are the

fields
A sequence defining the structure fields. The items must be 2-tuples or 3-tuples. The first item is the name of the field, the second item specifies the type of the field; it can be any ctypes data type.

For integer type fields like *c_int*, a third optional item can be given. It must be a small positive integer defining the bit width of the field.

Field names must be unique within one structure or union. This is not checked, only one field can be accessed when names are repeated.

It is possible to define the *_fields_* class variable *after* the class statement that defines the Structure subclass, this allows creating data types that directly or indirectly reference themselves:

```
class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
                 ...
                ]
```

The *_fields_* class variable must, however, be defined before the type is first used (an instance is created, `sizeof()` is called on it, and so on). Later assignments to the *_fields_* class variable will raise an AttributeError.

It is possible to defined sub-subclasses of structure types, they inherit the fields of the base class plus the *_fields_* defined in the sub-subclass, if any.

pack

An optional small integer that allows overriding the alignment of structure fields in the instance. *_pack_* must already be defined when *_fields_* is assigned, otherwise it will have no effect.

anonymous

An optional sequence that lists the names of unnamed (anonymous) fields. *_anonymous_* must be already defined when *_fields_* is assigned, otherwise it will have no effect.

The fields listed in this variable must be structure or union type fields. *ctypes* will create descriptors in the structure type that allow accessing the nested fields directly, without the need to create the structure or union field.

Here is an example type (Windows):

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]
```

The `TYPEDESC` structure describes a COM data type, the `vt` field specifies which one of the union fields is valid. Since the `u` field is defined as anonymous field, it is now possible to access the members directly off the TYPEDESC instance. `td.lptdesc` and `td.u.lptdesc` are equivalent, but the former is faster since it does not need to create a temporary union instance:

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

It is possible to defined sub-subclasses of structures, they inherit the fields of the base class. If the subclass definition has a separate *_fields_* variable, the fields specified in this are appended to the fields of the base class.

Structure and union constructors accept both positional and keyword arguments. Positional arguments are used to initialize member fields in the same order as they are appear in *_fields_*. Keyword arguments in the constructor are interpreted as attribute assignments, so they will initialize *_fields_* with the same name, or create new attributes for names not present in *_fields_*.

Arrays and pointers

class ctypes.**Array**(**args*)
Abstract base class for arrays.

The recommended way to create concrete array types is by multiplying any *ctypes* data type with a positive integer. Alternatively, you can subclass this type and define *_length_* and *_type_* class variables. Array elements can be read and written using standard subscript and slice accesses; for slice reads, the resulting object is *not* itself an *Array*.

length
A positive integer specifying the number of elements in the array. Out-of-range subscripts result in an *IndexError*. Will be returned by *len()*.

type
Specifies the type of each element in the array.

Array subclass constructors accept positional arguments, used to initialize the elements in order.

class ctypes.**_Pointer**
Private, abstract base class for pointers.

Concrete pointer types are created by calling *POINTER()* with the type that will be pointed to; this is done automatically by *pointer()*.

If a pointer points to an array, its elements can be read and written using standard subscript and slice accesses. Pointer objects have no size, so *len()* will raise *TypeError*. Negative subscripts will read from the memory *before* the pointer (as in C), and out-of-range subscripts will probably crash with an access violation (if you're lucky).

type
Specifies the type pointed to.

contents
Returns the object to which to pointer points. Assigning to this attribute changes the pointer to point to the assigned object.

CHAPTER 16

Optional Operating System Services

The modules described in this chapter provide interfaces to operating system features that are available on selected operating systems only. The interfaces are generally modeled after the Unix or C interfaces but they are available on some other systems as well (e.g. Windows or NT). Here's an overview:

16.1 `select` —Waiting for I/O 完成

This module provides access to the `select()` and `poll()` functions available in most operating systems, `epoll()` available on Linux 2.5+ and `kqueue()` available on most BSD. Note that on Windows, it only works for sockets; on other operating systems, it also works for other file types (in particular, on Unix, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

该模块定义以下内容：

exception `select.error`
The exception raised when an error occurs. The accompanying value is a pair containing the numeric error code from `errno` and the corresponding string, as would be printed by the C function `perror()`.

`select.`**`epoll`**`(`[*sizehint=-1*]`)`
(Only supported on Linux 2.5.44 and newer.) Returns an edge polling object, which can be used as Edge or Level Triggered interface for I/O events; see section 边缘触发和水平触发的轮询 *(epoll)* 对象 below for the methods supported by epolling objects.

2.6 新版功能.

`select.`**`poll`**`()`
（部分操作系统不支持）返回一个轮询对象，该对象支持注册和注销文件描述符，支持对描述符进行轮询以获取 I/O 事件。有关轮询对象支持的方法，请参阅下方*Poll 对象* 部分。

`select.`**`kqueue`**`()`
（仅支持 BSD）返回一个内核队列对象，请参阅下方*Kqueue 对象* 获取 kqueue 对象所支持的方法。

2.6 新版功能.

select.**kevent**(*ident*, *filter=KQ_FILTER_READ*, *flags=KQ_EV_ADD*, *fflags=0*, *data=0*, *udata=0*)
（仅支持 BSD）返回一个内核事件对象，请参阅下方*Kevent 对象* 获取 kevent 对象所支持的方法。

2.6 新版功能.

select.**select**(*rlist*, *wlist*, *xlist*[, *timeout*])
这是 Unix `select()` 系统调用的直接接口。前三个参数是由“等待对象”组成的序列，“等待对象”可以是表示文件描述符的整数，也可以是定义了`fileno()` 方法的对象（该方法没有参数，返回一个整数）：

- *rlist*：等待，直到有内容可以读取
- *wlist*：等待，直到可以开始写入
- *xlist*：等待“异常情况”（请参阅当前系统的手册，以获取哪些情况称为异常情况）

允许使用空序列，但是否接受三个空序列取决于平台。（目前已知在 Unix 上可以但 Windows 上不行。）可选的 *timeout* 参数指定超时时长，为浮点数，以秒为单位。如果省略了 *timeout* 参数，则该函数将阻塞，直到至少一个文件描述符准备就绪。超时值为零表示轮询且永不阻塞。

返回值是三个列表，包含已就绪对象，返回的三个列表是前三个参数的子集。当超时时间已到且没有对象就绪时，返回三个空列表。

Among the acceptable object types in the sequences are Python file objects (e.g. `sys.stdin`, or objects returned by `open()` or `os.popen()`), socket objects returned by `socket.socket()`. You may also define a *wrapper* class yourself, as long as it has an appropriate `fileno()` method (that really returns a file descriptor, not just a random integer).

注解： Windows 上不接受文件对象，但接受套接字。在 Windows 上，底层的 `select()` 函数由 WinSock 库提供，且不处理不是源自 WinSock 的文件描述符。

---

select.**PIPE_BUF**
Files reported as ready for writing by `select()`, `poll()` or similar interfaces in this module are guaranteed to not block on a write of up to `PIPE_BUF` bytes. This value is guaranteed by POSIX to be at least 512. Availability: Unix.

2.7 新版功能.

16.1.1 边缘触发和水平触发的轮询 (epoll) 对象

http://linux.die.net/man/4/epoll

屏蔽事件

常数	含义
EPOLLIN	可读
EPOLLOUT	可写
EPOLLPRI	紧急数据读取
EPOLLERR	在关联的文件描述符上有错误情况发生
EPOLLHUP	关联的文件描述符已挂起
EPOLLET	设置触发方式为边缘触发，默认为水平触发
EPOLLONESHOT	设置 one-shot 模式。触发一次事件后，该描述符会在轮询对象内部被禁用。
EPOLLRDNORM	Equivalent to EPOLLIN
EPOLLRDBAND	可以读取优先数据带。
EPOLLWRNORM	Equivalent to EPOLLOUT
EPOLLWRBAND	可以写入优先级数据。
EPOLLMSG	忽略

epoll.**close**()
关闭用于控制 epoll 对象的那个文件描述符。

epoll.**fileno**()
返回用于控制 epoll 对象的文件描述符对应的数字。

epoll.**fromfd**(*fd*)
根据给定的文件描述符创建 epoll 对象。

epoll.**register**(*fd*[, *eventmask*])
在 epoll 对象中注册一个文件描述符。

注解： Registering a file descriptor that's already registered raises an IOError –contrary to *Poll 对象*'s register.

epoll.**modify**(*fd*, *eventmask*)
Modify a register file descriptor.

epoll.**unregister**(*fd*)
从 epoll 对象中删除一个已注册的文件描述符。

epoll.**poll**([*timeout=-1*[, *maxevents=-1*]])
等待事件发生，timeout 是浮点数，单位为秒。

16.1.2 Poll 对象

大多数 Unix 系统支持 poll() 系统调用，为服务器提供了更好的可伸缩性，使服务器可以同时服务于大量客户端。poll() 的伸缩性更好，因为该调用内部仅列出所关注的文件描述符，而 select() 会构造一个 bitmap，在其中将所关注的描述符所对应的 bit 打开，然后重新遍历整个 bitmap。因此 select() 复杂度是 O（最高文件描述符），而 poll() 是 O（文件描述符数量）。

poll.**register**(*fd*[, *eventmask*])
在轮询对象中注册文件描述符。这样，将来调用*poll()* 方法时将检查文件描述符是否有未处理的 I/O 事件。*fd* 可以是整数，也可以是带有*fileno()* 方法的对象（该方法返回一个整数）。文件对象已经实现了 fileno()，因此它们也可以用作参数。

eventmask 是可选的位掩码，用于指定要检查的事件类型，它可以是常量 POLLIN、POLLPRI 和 POLLOUT 的组合，如下表所述。如果未指定本参数，默认将会检查所有 3 种类型的事件。

常数	含义
POLLIN	有要读取的数据
POLLPRI	有紧急数据需要读取
POLLOUT	准备输出：写不会阻塞
POLLERR	某种错误条件
POLLHUP	挂起
POLLNVAL	无效的请求：描述符未打开

注册已注册过的文件描述符不会报错，且等同于只注册一次该描述符。

poll.**modify**(*fd*, *eventmask*)
Modifies an already registered fd. This has the same effect as `register(fd, eventmask)`. Attempting to modify a file descriptor that was never registered causes an *IOError* exception with errno ENOENT to be raised.

2.6 新版功能.

poll.**unregister**(*fd*)
删除轮询对象正在跟踪的某个文件描述符。与*register()* 方法类似，*fd* 可以是整数，也可以是带有*fileno()* 方法的对象（该方法返回一个整数）。

尝试删除从未注册过的文件描述符会抛出*KeyError* 异常。

poll.**poll**([*timeout*])
轮询已注册的文件描述符的集合，并返回一个列表，列表可能为空，也可能有多个 (fd, event) 二元组，其中包含了要报告事件或错误的描述符。*fd* 是文件描述符，*event* 是一个位掩码，表示该描述符所报告的事件—POLLIN 表示可以读取，POLLOUT 表示该描述符可以写入，依此类推。空列表表示调用超时，没有任何文件描述符报告事件。如果指定了 *timeout*，它将指定系统等待事件时，等待多长时间后返回（以毫秒为单位）。如果 *timeout* 为空、负数或*None*，则本调用将阻塞，直到轮询对象发生事件为止。

16.1.3 Kqueue 对象

kqueue.**close**()
关闭用于控制 kqueue 对象的文件描述符。

kqueue.**fileno**()
返回用于控制 epoll 对象的文件描述符对应的数字。

kqueue.**fromfd**(*fd*)
根据给定的文件描述符创建 kqueue 对象。

kqueue.**control**(*changelist*, *max_events*[, *timeout*]) → eventlist
Kevent 的低级接口

- changelist 必须是一个可迭代对象，迭代出 kevent 对象，否则置为 None。
- max_events 必须是 0 或一个正整数。
- timeout 单位为秒（一般为浮点数），默认为 None，即永不超时。

16.1.4 Kevent 对象

https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

kevent.**ident**
Value used to identify the event. The interpretation depends on the filter but it's usually the file descriptor. In the constructor ident can either be an int or an object with a fileno() function. kevent stores the integer internally.

kevent.**filter**
内核过滤器的名称。

常数	含义
KQ_FILTER_READ	获取描述符，并在有数据可读时返回
KQ_FILTER_WRITE	获取描述符，并在有数据可写时返回
KQ_FILTER_AIO	AIO 请求
KQ_FILTER_VNODE	当在 *fflag* 中监视的一个或多个请求事件发生时返回
KQ_FILTER_PROC	监视进程 ID 上的事件
KQ_FILTER_NETDEV	观察网络设备上的事件 [在 Mac OS X 上不可用]
KQ_FILTER_SIGNAL	每当监视的信号传递到进程时返回
KQ_FILTER_TIMER	建立一个任意的计时器

kevent.**flags**
筛选器操作。

常数	含义
KQ_EV_ADD	添加或修改事件
KQ_EV_DELETE	从队列中删除事件
KQ_EV_ENABLE	Permitscontrol() 返回事件
KQ_EV_DISABLE	禁用事件
KQ_EV_ONESHOT	在第一次发生后删除事件
KQ_EV_CLEAR	检索事件后重置状态
KQ_EV_SYSFLAGS	内部事件
KQ_EV_FLAG1	内部事件
KQ_EV_EOF	筛选特定 EOF 条件
KQ_EV_ERROR	请参阅返回值

kevent.**fflags**
筛选特定标志。

KQ_FILTER_READ 和 KQ_FILTER_WRITE 过滤标志：

常数	含义
KQ_NOTE_LOWAT	套接字缓冲区的低水线

KQ_FILTER_VNODE 过滤标志：

常数	含义
KQ_NOTE_DELETE	已调用 *unlink()*
KQ_NOTE_WRITE	发生写入
KQ_NOTE_EXTEND	文件已扩展
KQ_NOTE_ATTRIB	属性已更改
KQ_NOTE_LINK	链接计数已更改
KQ_NOTE_RENAME	文件已重命名
KQ_NOTE_REVOKE	对文件的访问权限已被撤销

KQ_FILTER_PROC filter flags:

常数	含义
KQ_NOTE_EXIT	进程已退出
KQ_NOTE_FORK	该进程调用了 *fork()*
KQ_NOTE_EXEC	进程已执行新进程
KQ_NOTE_PCTRLMASK	内部过滤器标志
KQ_NOTE_PDATAMASK	内部过滤器标志
KQ_NOTE_TRACK	跨 *fork()* 执行进程
KQ_NOTE_CHILD	在 *NOTE_TRACK* 的子进程上返回
KQ_NOTE_TRACKERR	无法附加到子对象

KQ_FILTER_NETDEV 过滤器标志（在 Mac OS X 上不可用）：

常数	含义
KQ_NOTE_LINKUP	链接已建立
KQ_NOTE_LINKDOWN	链接已断开
KQ_NOTE_LINKINV	链接状态无效

kevent.**data**
过滤特定数据。

kevent.**udata**
用户定义的值。

16.2 threading —Higher-level threading interface

源代码: Lib/threading.py

This module constructs higher-level threading interfaces on top of the lower level *thread* module. See also the *mutex* and *Queue* modules.

The *dummy_threading* module is provided for situations where *threading* cannot be used because *thread* is missing.

注解: Starting with Python 2.6, this module provides **PEP 8** compliant aliases and properties to replace the camelCase names that were inspired by Java's threading API. This updated API is compatible with that of the *multiprocessing* module. However, no schedule has been set for the deprecation of the camelCase names and they remain fully supported in both Python 2.x and 3.x.

注解: Starting with Python 2.5, several Thread methods raise *RuntimeError* instead of *AssertionError* if called erroneously.

CPython implementation detail: In CPython, due to the *Global Interpreter Lock*, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use *multiprocessing*. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

This module defines the following functions and objects:

threading.**active_count**()
threading.**activeCount**()
返回当前存活的线程类*Thread*对象。返回的计数等于*enumerate()*返回的列表长度。

在 2.6 版更改: Added `active_count()` spelling.

threading.**Condition**()
A factory function that returns a new condition variable object. A condition variable allows one or more threads to wait until they are notified by another thread.

See 条件对象.

threading.**current_thread**()
threading.**currentThread**()
返回当前对应调用者的控制线程的*Thread*对象。如果调用者的控制线程不是利用*threading*创建，会返回一个功能受限的虚拟线程对象。

在 2.6 版更改: Added `current_thread()` spelling.

threading.**enumerate**()
以列表形式返回当前所有存活的*Thread*对象。该列表包含守护线程，*current_thread()*创建的虚拟线程对象和主线程。它不包含已终结的线程和尚未开始的线程。

threading.**Event**()
A factory function that returns a new event object. An event manages a flag that can be set to true with the *set()* method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true.

See 事件对象.

class threading.**local**
A class that represents thread-local data. Thread-local data are data whose values are thread specific. To manage thread-local data, just create an instance of *local* (or a subclass) and store attributes on it:

```
mydata = threading.local()
mydata.x = 1
```

在不同的线程中，实例的值会不同。

更多相关细节和大量示例，参见 `_threading_local` 模块的文档。

2.4 新版功能.

threading.**Lock**()
A factory function that returns a new primitive lock object. Once a thread has acquired it, subsequent attempts to acquire it block, until it is released; any thread may release it.

See 锁对象.

threading.**RLock**()
A factory function that returns a new reentrant lock object. A reentrant lock must be released by the thread that

acquired it. Once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

See 递归锁对象.

threading.**Semaphore**([*value*])
A factory function that returns a new semaphore object. A semaphore manages a counter representing the number of `release()` calls minus the number of `acquire()` calls, plus an initial value. The `acquire()` method blocks if necessary until it can return without making the counter negative. If not given, *value* defaults to 1.

See 信号量对象.

threading.**BoundedSemaphore**([*value*])
A factory function that returns a new bounded semaphore object. A bounded semaphore checks to make sure its current value doesn't exceed its initial value. If it does, *ValueError* is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it's a sign of a bug. If not given, *value* defaults to 1.

class threading.**Thread**
A class that represents a thread of control. This class can be safely subclassed in a limited fashion.

See 线程对象.

class threading.**Timer**
A thread that executes a function after a specified interval has passed.

See 定时器对象.

threading.**settrace**(*func*)
为所有*threading* 模块开始的线程设置追踪函数。在每个线程的*run()* 方法被调用前，*func* 会被传递给*sys.settrace()* 。

2.3 新版功能.

threading.**setprofile**(*func*)
为所有*threading* 模块开始的线程设置性能测试函数。在每个线程的*run()* 方法被调用前，*func* 会被传递给*sys.setprofile()* 。

2.3 新版功能.

threading.**stack_size**([*size*])
Return the thread stack size used when creating new threads. The optional *size* argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If *size* is not specified, 0 is used. If changing the thread stack size is unsupported, a *ThreadError* is raised. If the specified stack size is invalid, a *ValueError* is raised and the stack size is unmodified. 32kB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32kB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4kB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information). Availability: Windows, systems with POSIX threads.

2.5 新版功能.

exception threading.**ThreadError**
Raised for various threading-related errors as described below. Note that many interfaces use *RuntimeError* instead of *ThreadError*.

Detailed interfaces for the objects are documented below.

该模块的设计基于 Java 的线程模型。但是，在 Java 里面，锁和条件变量是每个对象的基础特性，而在 Python 里面，这些被独立成了单独的对象。Python 的*Thread* 类只是 Java 的 Thread 类的一个子集；目前还没有优先

级，没有线程组，线程还不能被销毁、停止、暂停、恢复或中断。Java 的 Thread 类的静态方法在实现时会映射为模块级函数。

下述方法的执行都是原子性的。

16.2.1 线程对象

This class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the `run()` method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, *only* override the `__init__()` and `run()` methods of this class.

Once a thread object is created, its activity must be started by calling the thread's `start()` method. This invokes the `run()` method in a separate thread of control.

Once the thread's activity is started, the thread is considered 'alive'. It stops being alive when its `run()` method terminates –either normally, or by raising an unhandled exception. The `is_alive()` method tests whether the thread is alive.

Other threads can call a thread's `join()` method. This blocks the calling thread until the thread whose `join()` method is called is terminated.

A thread has a name. The name can be passed to the constructor, and read or changed through the `name` attribute.

A thread can be flagged as a "daemon thread". The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set through the `daemon` property.

注解： 守护线程在程序关闭时会突然关闭。他们的资源（例如已经打开的文档，数据库事务等等）可能没有被正确释放。如果你想你的线程正常停止，设置他们成为非守护模式并且使用合适的信号机制，例如: *Event*。

有个"主线程"对象；这对应 Python 程序里面初始的控制线程。它不是一个守护线程。

There is the possibility that "dummy thread objects" are created. These are thread objects corresponding to "alien threads", which are threads of control started outside the threading module, such as directly from C code. Dummy thread objects have limited functionality; they are always considered alive and daemonic, and cannot be `join()`ed. They are never deleted, since it is impossible to detect the termination of alien threads.

class threading.**Thread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*)

调用这个构造函数时，必需带有关键字参数。参数如下：

group 应该为 `None`；为了日后扩展 `ThreadGroup` 类实现而保留。

target 是用于*run()* 方法调用的可调用对象。默认是 `None`，表示不需要调用任何方法。

name 是线程名称。默认情况下，由"Thread-*N*"格式构成一个唯一的名称，其中 *N* 是小的十进制数。

args 是用于调用目标函数的参数元组。默认是 `()`。

kwargs 是用于调用目标函数的关键字参数字典。默认是 `{}`。

如果子类型重载了构造函数，它一定要确保在做任何事前，先发起调用基类构造器 (`Thread.__init__()`)。

start()

开始线程活动。

It must be called at most once per thread object. It arranges for the object's *run()* method to be invoked in a separate thread of control.

如果同一个线程对象中调用这个方法的次数大于一次，会抛出*RuntimeError*。

run()
代表线程活动的方法。

You may override this method in a subclass. The standard *run()* method invokes the callable object passed to the object's constructor as the *target* argument, if any, with sequential and keyword arguments taken from the *args* and *kwargs* arguments, respectively.

join([*timeout*])
Wait until the thread terminates. This blocks the calling thread until the thread whose *join()* method is called terminates –either normally or through an unhandled exception –or until the optional timeout occurs.

When the *timeout* argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As *join()* always returns None, you must call *isAlive()* after *join()* to decide whether a timeout happened –if the thread is still alive, the *join()* call timed out.

当 *timeout* 参数不存在或者是 None，这个操作会阻塞直到线程终结。

A thread can be *join()*ed many times.

join() raises a *RuntimeError* if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to *join()* a thread before it has been started and attempts to do so raises the same exception.

name
只用于识别的字符串。它没有语义。多个线程可以赋予相同的名称。初始名称由构造函数设置。

2.6 新版功能.

getName()
setName()
Pre-2.6 API for *name*.

ident
The 'thread identifier' of this thread or None if the thread has not been started. This is a nonzero integer. See the *thread.get_ident()* function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

2.6 新版功能.

is_alive()
isAlive()
返回线程是否存活。

This method returns True just before the *run()* method starts until just after the *run()* method terminates. The module function *enumerate()* returns a list of all alive threads.

在 2.6 版更改: Added is_alive() spelling.

daemon
A boolean value indicating whether this thread is a daemon thread (True) or not (False). This must be set before *start()* is called, otherwise *RuntimeError* is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to *daemon* = False.

当没有存活的非守护线程时，整个 Python 程序才会退出。

2.6 新版功能.

isDaemon()

setDaemon()
Pre-2.6 API for *daemon*.

16.2.2 锁对象

A primitive lock is a synchronization primitive that is not owned by a particular thread when locked. In Python, it is currently the lowest level synchronization primitive available, implemented directly by the *thread* extension module.

A primitive lock is in one of two states, "locked" or "unlocked". It is created in the unlocked state. It has two basic methods, `acquire()` and `release()`. When the state is unlocked, `acquire()` changes the state to locked and returns immediately. When the state is locked, `acquire()` blocks until a call to `release()` in another thread changes it to unlocked, then the `acquire()` call resets it to locked and returns. The `release()` method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a *ThreadError* will be raised.

When more than one thread is blocked in `acquire()` waiting for the state to turn to unlocked, only one thread proceeds when a `release()` call resets the state to unlocked; which one of the waiting threads proceeds is not defined, and may vary across implementations.

所有方法的执行都是原子性的。

`Lock.`**acquire**([*blocking*])
可以阻塞或非阻塞地获得锁。

当调用时参数 *blocking* 设置为 `True` （缺省值），阻塞直到锁被释放，然后将锁锁定并返回 `True` 。

在参数 *blocking* 被设置为 `False` 的情况下调用，将不会发生阻塞。如果调用时 *blocking* 设为 `True` 会阻塞，并立即返回 `False` ；否则，将锁锁定并返回 `True`。

`Lock.`**release**()
Release a lock.

当锁被锁定，将它重置为未锁定，并返回。如果其他线程正在等待这个锁解锁而被阻塞，只允许其中一个允许。

When invoked on an unlocked lock, a *ThreadError* is raised.

没有返回值。

locked()
Return true if the lock is acquired.

16.2.3 递归锁对象

重入锁是一个可以被同一个线程多次获取的同步基元组件。在内部，它在基元锁的锁定/非锁定状态上附加了"所属线程"和"递归等级"的概念。在锁定状态下，某些线程拥有锁；在非锁定状态下，没有线程拥有它。

To lock the lock, a thread calls its `acquire()` method; this returns once the thread owns the lock. To unlock the lock, a thread calls its `release()` method. `acquire()`/`release()` call pairs may be nested; only the final `release()` (the `release()` of the outermost pair) resets the lock to unlocked and allows another thread blocked in `acquire()` to proceed.

`RLock.`**acquire**([*blocking=1*])
可以阻塞或非阻塞地获得锁。

当无参数调用时：如果这个线程已经拥有锁，递归级别增加一，并立即返回。否则，如果其他线程拥有该锁，则阻塞至该锁解锁。一旦锁被解锁 (不属于任何线程)，则抢夺所有权，设置递归等级为一，并返回。如果多个线程被阻塞，等待锁被解锁，一次只有一个线程能抢到锁的所有权。在这种情况下，没有返回值。

When invoked with the *blocking* argument set to true, do the same thing as when called without arguments, and return true.

When invoked with the *blocking* argument set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

RLock.**release**()
释放锁，自减递归等级。如果减到零，则将锁重置为非锁定状态 (不被任何线程拥有)，并且，如果其他线程正被阻塞着等待锁被解锁，则仅允许其中一个线程继续。如果自减后，递归等级仍然不是零，则锁保持锁定，仍由调用线程拥有。

只有当前线程拥有锁才能调用这个方法。如果锁被释放后调用这个方法，会引起*RuntimeError*异常。

没有返回值。

16.2.4 条件对象

A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. (Passing one in is useful when several condition variables must share the same lock.)

A condition variable has `acquire()` and `release()` methods that call the corresponding methods of the associated lock. It also has a `wait()` method, and `notify()` and `notifyAll()` methods. These three must only be called when the calling thread has acquired the lock, otherwise a *RuntimeError* is raised.

The `wait()` method releases the lock, and then blocks until it is awakened by a `notify()` or `notifyAll()` call for the same condition variable in another thread. Once awakened, it re-acquires the lock and returns. It is also possible to specify a timeout.

The `notify()` method wakes up one of the threads waiting for the condition variable, if any are waiting. The `notifyAll()` method wakes up all threads waiting for the condition variable.

Note: the `notify()` and `notifyAll()` methods don't release the lock; this means that the thread or threads awakened will not return from their `wait()` call immediately, but only when the thread that called `notify()` or `notifyAll()` finally relinquishes ownership of the lock.

Tip: the typical programming style using condition variables uses the lock to synchronize access to some shared state; threads that are interested in a particular change of state call `wait()` repeatedly until they see the desired state, while threads that modify the state call `notify()` or `notifyAll()` when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer situation with unlimited buffer capacity:

```
# Consume one item
cv.acquire()
while not an_item_is_available():
    cv.wait()
get_an_available_item()
cv.release()

# Produce one item
cv.acquire()
make_an_item_available()
cv.notify()
cv.release()
```

To choose between `notify()` and `notifyAll()`, consider whether one state change can be interesting for only one or several waiting threads. E.g. in a typical producer-consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

class threading.**Condition**([*lock*])
如果给出了非 None 的 *lock* 参数，则它必须为*Lock* 或者*RLock* 对象，并且它将被用作底层锁。否则，将会创建新的*RLock* 对象，并将其用作底层锁。

acquire(**args*)
请求底层锁。此方法调用底层锁的相应方法，返回值是底层锁相应方法的返回值。

release()
释放底层锁。此方法调用底层锁的相应方法。没有返回值。

wait([*timeout*])
等待直到被通知或发生超时。如果线程在调用此方法时没有获得锁，将会引发*RuntimeError* 异常。

This method releases the underlying lock, and then blocks until it is awakened by a *notify()* or *notifyAll()* call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

当提供了 *timeout* 参数且不是 None 时，它应该是一个浮点数，代表操作的超时时间，以秒为单位（可以为小数）。

当底层锁是个*RLock* ，不会使用它的*release()* 方法释放锁，因为当它被递归多次获取时，实际上可能无法解锁。相反，使用了*RLock* 类的内部接口，即使多次递归获取它也能解锁它。然后，在重新获取锁时，使用另一个内部接口来恢复递归级别。

notify(*n=1*)
默认唤醒一个等待这个条件的线程。如果调用线程在没有获得锁的情况下调用这个方法，会引发*RuntimeError* 异常。

这个方法唤醒最多 *n* 个正在等待这个条件变量的线程；如果没有线程在等待，这是一个空操作。

当前实现中，如果至少有 *n* 个线程正在等待，准确唤醒 *n* 个线程。但是依赖这个行为并不安全。未来，优化的实现有时会唤醒超过 *n* 个线程。

注意：被唤醒的线程实际上不会返回它调用的*wait()* ，直到它可以重新获得锁。因为*notify()* 不会释放锁，只有它的调用者应该这样做。

notify_all()
notifyAll()
唤醒所有正在等待这个条件的线程。这个方法行为与*notify()* 相似，但并不只唤醒单一线程，而是唤醒所有等待线程。如果调用线程在调用这个方法时没有获得锁，会引发*RuntimeError* 异常。

在 2.6 版更改: Added notify_all() spelling.

16.2.5 信号量对象

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used P() and V() instead of acquire() and release()).

A semaphore manages an internal counter which is decremented by each acquire() call and incremented by each release() call. The counter can never go below zero; when acquire() finds that it is zero, it blocks, waiting until some other thread calls release().

class threading.**Semaphore**([*value*])
可选参数 *value* 赋予内部计数器初始值，默认值为 1 。如果 *value* 被赋予小于 0 的值，将会引发*ValueError* 异常。

acquire([*blocking*])
获取一个信号量。

When invoked without arguments: if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other thread has called *release()* to make it larger than zero. This is done with proper interlocking so that if multiple *acquire()* calls are blocked, *release()* will wake exactly one of them up. The implementation may pick one at random, so the order in which blocked threads are awakened should not be relied on. There is no return value in this case.

When invoked with *blocking* set to true, do the same thing as when called without arguments, and return true.

When invoked with *blocking* set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

release()
释放一个信号量，将内部计数器的值增加 1。当计数器原先的值为 0 且有其它线程正在等待它再次大于 0 时，唤醒正在等待的线程。

Semaphore 例子

信号量通常用于保护数量有限的资源，例如数据库服务器。在资源数量固定的任何情况下，都应该使用有界信号量。在生成任何工作线程前，应该在主线程中初始化信号量。

```
maxconnections = 5
...
pool_sema = BoundedSemaphore(value=maxconnections)
```

工作线程生成后，当需要连接服务器时，这些线程将调用信号量的 acquire 和 release 方法：

```
pool_sema.acquire()
conn = connectdb()
... use connection ...
conn.close()
pool_sema.release()
```

使用有界信号量能减少这种编程错误：信号量的释放次数多于其请求次数。

16.2.6 事件对象

这是线程之间通信的最简单机制之一：一个线程发出事件信号，而其他线程等待该信号。

一个事件对象管理一个内部标志，调用*set()* 方法可将其设置为 true，调用*clear()* 方法可将其设置为 false，调用*wait()* 方法将进入阻塞直到标志为 true。

class threading.**Event**
The internal flag is initially false.

is_set()
isSet()
Return true if and only if the internal flag is true.

在 2.6 版更改: Added `is_set()` spelling.

set()
将内部标志设置为 true。所有正在等待这个事件的线程将被唤醒。当标志为 true 时，调用*wait()* 方法的线程不会被被阻塞。

clear()
将内部标志设置为 false。之后调用*wait()* 方法的线程将会被阻塞，直到调用*set()* 方法将内部标志再次设置为 true。

wait([*timeout*])
阻塞线程直到内部变量为 true。如果调用时内部标志为 true，将立即返回。否则将阻塞线程，直到调用*set()* 方法将标志设置为 true 或者发生可选的超时。

当提供了 timeout 参数且不是 None 时，它应该是一个浮点数，代表操作的超时时间，以秒为单位（可以为小数）。

This method returns the internal flag on exit, so it will always return `True` except if a timeout is given and the operation times out.

在 2.7 版更改: 很明显，方法总是返回 None。

16.2.7 定时器对象

此类表示一个操作应该在等待一定的时间之后运行—相当于一个定时器。*Timer* 类是*Thread* 类的子类，因此可以像一个自定义线程一样工作。

与线程一样，通过调用 start() 方法启动定时器。而*cancel()* 方法可以停止计时器（在计时结束前），定时器在执行其操作之前等待的时间间隔可能与用户指定的时间间隔不完全相同。

例如

```
def hello():
    print "hello, world"

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

class threading.**Timer**(*interval*, *function*, *args=[]*, *kwargs={}*)
Create a timer that will run *function* with arguments *args* and keyword arguments *kwargs*, after *interval* seconds have passed.

cancel()
停止定时器并取消执行计时器将要执行的操作。仅当计时器仍处于等待状态时有效。

16.2.8 Using locks, conditions, and semaphores in the `with` statement

All of the objects provided by this module that have `acquire()` and `release()` methods can be used as context managers for a `with` statement. The `acquire()` method will be called when the block is entered, and `release()` will be called when the block is exited.

Currently, *Lock*, *RLock*, *Condition*, *Semaphore*, and *BoundedSemaphore* objects may be used as `with` statement context managers. For example:

```
import threading

some_rlock = threading.RLock()

with some_rlock:
    print "some_rlock is locked while this executes"
```

16.2.9 Importing in threaded code

While the import machinery is thread-safe, there are two key restrictions on threaded imports due to inherent limitations in the way that thread-safety is provided:

- Firstly, other than in the main module, an import should not have the side effect of spawning a new thread and then waiting for that thread in any way. Failing to abide by this restriction can lead to a deadlock if the spawned thread directly or indirectly attempts to import a module.
- Secondly, all import attempts must be completed before the interpreter starts shutting itself down. This can be most easily achieved by only performing imports from non-daemon threads created through the threading module. Daemon threads and threads created directly with the thread module will require some other form of synchronization to ensure they do not attempt imports after system shutdown has commenced. Failure to abide by this restriction will lead to intermittent exceptions and crashes during interpreter shutdown (as the late imports attempt to access machinery which is no longer in a valid state).

16.3 `thread` —Multiple threads of control

注解: The *thread* module has been renamed to `_thread` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3; however, you should consider using the high-level *threading* module instead.

This module provides low-level primitives for working with multiple threads (also called *light-weight processes* or *tasks*) —multiple threads of control sharing their global data space. For synchronization, simple locks (also called *mutexes* or *binary semaphores*) are provided. The *threading* module provides an easier to use and higher-level threading API built on top of this module.

The module is optional. It is supported on Windows, Linux, SGI IRIX, Solaris 2.x, as well as on systems that have a POSIX thread (a.k.a. "pthread") implementation. For systems lacking the *thread* module, the *dummy_thread* module is available. It duplicates this module's interface and can be used as a drop-in replacement.

It defines the following constant and functions:

exception `thread.`**error**
Raised on thread-specific errors.

`thread.`**LockType**
This is the type of lock objects.

`thread.`**start_new_thread**(*function*, *args*[, *kwargs*])
Start a new thread and return its identifier. The thread executes the function *function* with the argument list *args* (which must be a tuple). The optional *kwargs* argument specifies a dictionary of keyword arguments. When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

`thread.`**interrupt_main**()
Raise a *KeyboardInterrupt* exception in the main thread. A subthread can use this function to interrupt the main thread.

2.3 新版功能.

`thread.`**exit**()
Raise the *SystemExit* exception. When not caught, this will cause the thread to exit silently.

`thread.`**allocate_lock**()
Return a new lock object. Methods of locks are described below. The lock is initially unlocked.

thread.**get_ident**()
Return the ‘thread identifier’ of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

thread.**stack_size**([*size*])
Return the thread stack size used when creating new threads. The optional *size* argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32kB). If *size* is not specified, 0 is used. If changing the thread stack size is unsupported, the *error* exception is raised. If the specified stack size is invalid, a *ValueError* is raised and the stack size is unmodified. 32kB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32kB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4kB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information). Availability: Windows, systems with POSIX threads.

2.5 新版功能.

Lock objects have the following methods:

lock.**acquire**([*waitflag*])
Without the optional argument, this method acquires the lock unconditionally, if necessary waiting until it is released by another thread (only one thread at a time can acquire a lock —that’s their reason for existence). If the integer *waitflag* argument is present, the action depends on its value: if it is zero, the lock is only acquired if it can be acquired immediately without waiting, while if it is nonzero, the lock is acquired unconditionally as before. The return value is `True` if the lock is acquired successfully, `False` if not.

lock.**release**()
Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

lock.**locked**()
Return the status of the lock: `True` if it has been acquired by some thread, `False` if not.

In addition to these methods, lock objects can also be used via the `with` statement, e.g.:

```
import thread

a_lock = thread.allocate_lock()

with a_lock:
    print "a_lock is locked while this executes"
```

Caveats:

- Threads interact strangely with interrupts: the *KeyboardInterrupt* exception will be received by an arbitrary thread. (When the *signal* module is available, interrupts always go to the main thread.)
- Calling *sys.exit()* or raising the *SystemExit* exception is equivalent to calling *thread.exit()*.
- It is not possible to interrupt the `acquire()` method on a lock —the *KeyboardInterrupt* exception will happen after the lock has been acquired.
- When the main thread exits, it is system defined whether the other threads survive. On SGI IRIX using the native thread implementation, they survive. On most other systems, they are killed without executing `try` … `finally` clauses or executing object destructors.
- When the main thread exits, it does not do any of its usual cleanup (except that `try` … `finally` clauses are honored), and the standard I/O files are not flushed.

16.4 dummy_threading —可直接替代 threading 模块。

源代码： Lib/dummy_threading.py

This module provides a duplicate interface to the *threading* module. It is meant to be imported when the *thread* module is not provided on a platform.

Suggested usage is:

```
try:
    import threading as _threading
except ImportError:
    import dummy_threading as _threading
```

Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.

16.5 dummy_thread —Drop-in replacement for the thread module

注解： The *dummy_thread* module has been renamed to _dummy_thread in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3; however, you should consider using the high-lever *dummy_threading* module instead.

Source code: Lib/dummy_thread.py

This module provides a duplicate interface to the *thread* module. It is meant to be imported when the *thread* module is not provided on a platform.

Suggested usage is:

```
try:
    import thread as _thread
except ImportError:
    import dummy_thread as _thread
```

Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.

16.6 multiprocessing —Process-based “threading” interface

2.6 新版功能.

16.6.1 概述

multiprocessing 是一个用与*threading* 模块相似 API 的支持产生进程的包。*multiprocessing* 包同时提供本地和远程并发，使用子进程代替线程，有效避免*Global Interpreter Lock* 带来的影响。因此，*multiprocessing* 模块允许程序员充分利用机器上的多个核心。Unix 和 Windows 上都可以运行。

The *multiprocessing* module also introduces APIs which do not have analogs in the *threading* module. A prime example of this is the `Pool` object which offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism). The following example demonstrates the common practice of defining such functions in a module so that child processes can successfully import that module. This basic example of data parallelism using `Pool`,

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    p = Pool(5)
    print(p.map(f, [1, 2, 3]))
```

将打印到标准输出

```
[1, 4, 9]
```

Process 类

在*multiprocessing* 中，通过创建一个*Process* 对象然后调用它的 `start()` 方法来生成进程。*Process* 和*threading.Thread* API 相同。一个简单的多进程程序示例是:

```
from multiprocessing import Process

def f(name):
    print 'hello', name

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

要显示所涉及的各个进程 ID，这是一个扩展示例:

```
from multiprocessing import Process
import os

def info(title):
    print title
    print 'module name:', __name__
    if hasattr(os, 'getppid'):  # only available on Unix
        print 'parent process:', os.getppid()
    print 'process id:', os.getpid()

def f(name):
    info('function f')
    print 'hello', name
```

(下页继续)

(续上页)

```
if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

For an explanation of why (on Windows) the `if __name__ == '__main__'` part is necessary, see 编程指导.

在进程之间交换对象

multiprocessing 支持进程之间的两种通信通道：

队列

The *Queue* class is a near clone of *Queue.Queue*. For example:

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print q.get()    # prints "[42, None, 'hello']"
    p.join()
```

队列是线程和进程安全的。

管道

Pipe() 函数返回一个由管道连接的连接对象，默认情况下是双工（双向）。例如：

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print parent_conn.recv()   # prints "[42, None, 'hello']"
    p.join()
```

返回的两个连接对象*Pipe()* 表示管道的两端。每个连接对象都有*send()* 和*recv()* 方法（相互之间的）。请注意，如果两个进程（或线程）同时尝试读取或写入管道的 同一端，则管道中的数据可能会损坏。当然，同时使用管道的不同端的进程不存在损坏的风险。

进程之间的同步

multiprocessing 包含来自*threading* 的所有同步原语的等价物。例如，可以使用锁来确保一次只有一个进程打印到标准输出:

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    print 'hello world', i
    l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

不使用来自不同进程的锁输出容易产生混淆。

在进程之间共享状态

如上所述，在进行并发编程时，通常最好尽量避免使用共享状态。使用多个进程时尤其如此。

但是，如果你真的需要使用一些共享数据，那么*multiprocessing* 提供了两种方法。

共享内存

可以使用*Value* 或*Array* 将数据存储在共享内存映射中。例如，以下代码:

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print num.value
    print arr[:]
```

将打印

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

创建 num 和 arr 时使用的 'd' 和 'i' 参数是*array* 模块使用的类型的 typecode： 'd' 表示双精度浮点数，'i' 表示有符号整数。这些共享对象将是进程和线程安全的。

为了更灵活地使用共享内存，可以使用*multiprocessing.sharedctypes* 模块，该模块支持创建从共享内存分配的任意 ctypes 对象。

服务器进程

由 Manager() 返回的管理器对象控制一个服务器进程，该进程保存 Python 对象并允许其他进程使用代理操作它们。

A manager returned by Manager() will support types list, *dict*, *Namespace*, *Lock*, *RLock*, *Semaphore*, *BoundedSemaphore*, *Condition*, *Event*, *Queue*, *Value* and *Array*. For example,

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    manager = Manager()

    d = manager.dict()
    l = manager.list(range(10))

    p = Process(target=f, args=(d, l))
    p.start()
    p.join()

    print d
    print l
```

将打印

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

服务器进程管理器比使用共享内存对象更灵活，因为它们可以支持任意对象类型。此外，单个管理器可以通过网络由不同计算机上的进程共享。但是，它们比使用共享内存慢。

使用工作进程

Pool 类表示一个工作进程池。它具有允许以几种不同方式将任务分配到工作进程的方法。

例如

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    pool = Pool(processes=4)              # start 4 worker processes

    # print "[0, 1, 4,..., 81]"
    print pool.map(f, range(10))
```

(下页继续)

(续上页)

```
    # print same numbers in arbitrary order
    for i in pool.imap_unordered(f, range(10)):
        print i

    # evaluate "f(20)" asynchronously
    res = pool.apply_async(f, (20,))      # runs in *only* one process
    print res.get(timeout=1)              # prints "400"

    # evaluate "os.getpid()" asynchronously
    res = pool.apply_async(os.getpid, ()) # runs in *only* one process
    print res.get(timeout=1)              # prints the PID of that process

    # launching multiple evaluations asynchronously *may* use more processes
    multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
    print [res.get(timeout=1) for res in multiple_results]

    # make a single worker sleep for 10 secs
    res = pool.apply_async(time.sleep, (10,))
    try:
        print res.get(timeout=1)
    except TimeoutError:
        print "We lacked patience and got a multiprocessing.TimeoutError"
```

请注意，池的方法只能由创建它的进程使用。

注解： Functionality within this package requires that the `__main__` module be importable by the children. This is covered in 编程指导 however it is worth pointing out here. This means that some examples, such as the `Pool` examples will not work in the interactive interpreter. For example:

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
```

(如果你尝试这个，它实际上会以半随机的方式输出三个完整的回溯，然后你可能不得不以某种方式停止主进程。)

16.6.2 参考

multiprocessing 包大部分复制了*threading* 模块的 API。

Process 和异常

class multiprocessing.**Process**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*)
进程对象表示在单独进程中运行的活动。*Process* 类等价于*threading.Thread* 。

The constructor should always be called with keyword arguments. *group* should always be None; it exists solely for compatibility with *threading.Thread*. *target* is the callable object to be invoked by the *run()* method. It defaults to None, meaning nothing is called. *name* is the process name. By default, a unique name is constructed of the form 'Process-N_1:N_2:…:N_k' where N_1,N_2,…,N_k is a sequence of integers whose length is determined by the *generation* of the process. *args* is the argument tuple for the target invocation. *kwargs* is a dictionary of keyword arguments for the target invocation. By default, no arguments are passed to *target*.

如果子类重写构造函数，它必须确保它在对进程执行任何其他操作之前调用基类构造函数（Process.__init__() ）。

run()
表示进程活动的方法。

你可以在子类中重载此方法。标准*run()* 方法调用传递给对象构造函数的可调用对象作为目标参数（如果有），分别从 *args* 和 *kwargs* 参数中获取顺序和关键字参数。

start()
启动进程活动。

每个进程对象最多只能调用一次。它安排对象的*run()* 方法在一个单独的进程中调用。

join([*timeout*])
Block the calling thread until the process whose *join()* method is called terminates or until the optional timeout occurs.

If *timeout* is None then there is no timeout.

一个进程可以合并多次。

进程无法并入自身，因为这会导致死锁。尝试在启动进程之前合并进程是错误的。

name
The process' s name.

The name is a string used for identification purposes only. It has no semantics. Multiple processes may be given the same name. The initial name is set by the constructor.

is_alive()
返回进程是否还活着。

粗略地说，从*start()* 方法返回到子进程终止之前，进程对象仍处于活动状态。

daemon
进程的守护标志，一个布尔值。这必须在*start()* 被调用之前设置。

初始值继承自创建进程。

当进程退出时，它会尝试终止其所有守护进程子进程。

请注意，不允许守护进程创建子进程。否则，守护进程会在子进程退出时终止其子进程。另外，这些 **不是** Unix 守护进程或服务，它们是正常进程，如果非守护进程已经退出，它们将被终止（并且不被合并）。

除了*threading.Thread* API，*Process* 对象还支持以下属性和方法：

pid
返回进程 ID。在生成该进程之前，这将是 None 。

exitcode
的退子进程出代码。如果进程尚未终止，这将是 None 。负值 *-N* 表示孩子被信号 *N* 终止。

authkey
进程的身份验证密钥（字节字符串）。

当*multiprocessing* 初始化时，主进程使用*os.urandom()* 分配一个随机字符串。

当创建*Process* 对象时，它将继承其父进程的身份验证密钥，尽管可以通过将*authkey* 设置为另一个字节字符串来更改。

参见*认证密码* 。

terminate()
终止进程。在 Unix 上，这是使用 SIGTERM 信号完成的；在 Windows 上使用 TerminateProcess() 。请注意，不会执行退出处理程序和 finally 子句等。

请注意，进程的后代进程将不会被终止——它们将简单地变成孤立的。

> **警告：** 如果在关联进程使用管道或队列时使用此方法，则管道或队列可能会损坏，并可能无法被其他进程使用。类似地，如果进程已获得锁或信号量等，则终止它可能导致其他进程死锁。

注意*start()* 、*join()* 、*is_alive()* 、*terminate()* 和*exitcode* 方法只能由创建进程对象的进程调用。

Process 一些方法的示例用法：

```
>>> import multiprocessing, time, signal
>>> p = multiprocessing.Process(target=time.sleep, args=(1000,))
>>> print p, p.is_alive()
<Process(Process-1, initial)> False
>>> p.start()
>>> print p, p.is_alive()
<Process(Process-1, started)> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print p, p.is_alive()
<Process(Process-1, stopped[SIGTERM])> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.**BufferTooShort**
当提供的缓冲区对象太小而无法读取消息时，*Connection.recv_bytes_into()* 引发的异常。

如果 e 是一个*BufferTooShort* 实例，那么 e.args[0] 将把消息作为字节字符串给出。

管道和队列

使用多进程时，一般使用消息机制实现进程间通信，尽可能避免使用同步原语，例如锁。

消息机制包含：*Pipe()* (可以用于在两个进程间传递消息)，以及队列 (能够在多个生产者和消费者之间通信)。

The *Queue*, *multiprocessing.queues.SimpleQueue* and *JoinableQueue* types are multi-producer, multi-consumer FIFO queues modelled on the *Queue.Queue* class in the standard library. They differ in that *Queue* lacks the *task_done()* and *join()* methods introduced into Python 2.5's *Queue.Queue* class.

如果你使用了*JoinableQueue*，那么你 ** 必须 ** 对每个已经移出队列的任务调用*JoinableQueue.task_done()*。不然的话用于统计未完成任务的信号量最终会溢出并抛出异常。

另外还可以通过使用一个管理器对象创建一个共享队列，详见数据管理器。

注解： *multiprocessing* uses the usual *Queue.Empty* and *Queue.Full* exceptions to signal a timeout. They are not available in the *multiprocessing* namespace so you need to import them from *Queue*.

注解： 当一个对象被放入一个队列中时，这个对象首先会被一个后台线程用 pickle 序列化，并将序列化后的数据通过一个底层管道的管道传递到队列中。这种做法会有点让人惊讶，但一般不会出现什么问题。如果它们确实妨碍了你，你可以使用一个由管理器*manager* 创建的队列替换它。

(1) After putting an object on an empty queue there may be an infinitesimal delay before the queue's *empty()* method returns *False* and *get_nowait()* can return without raising *Queue.Empty*.

(2) 如果有多个进程同时将对象放入队列，那么在队列的另一端接受到的对象可能是无序的。但是由同一个进程放入的多个对象的顺序在另一端输出时总是一样的。

警告： If a process is killed using *Process.terminate()* or *os.kill()* while it is trying to use a *Queue*, then the data in the queue is likely to become corrupted. This may cause any other process to get an exception when it tries to use the queue later on.

警告： 正如刚才提到的，如果一个子进程将一些对象放进队列中 (并且它没有用*JoinableQueue.cancel_join_thread* 方法)，那么这个进程在所有缓冲区的对象被刷新进管道之前，是不会终止的。

这意味着，除非你确定所有放入队列中的对象都已经被消费了，否则如果你试图等待这个进程，你可能会陷入死锁中。相似地，如果该子进程不是后台进程，那么父进程可能在试图等待所有非后台进程退出时挂起。

注意用管理器创建的队列不存在这个问题，详见编程指导。

该例子 展示了如何使用队列实现进程间通信。

`multiprocessing.`**`Pipe`**`([`*duplex*`])`

Returns a pair `(conn1, conn2)` of *Connection* objects representing the ends of a pipe.

如果 *duplex* 被置为 `True` (默认值)，那么该管道是双向的。如果 *duplex* 被置为 `False`，那么该管道是单向的，即 `conn1` 只能用于接收消息，而 `conn2` 仅能用于发送消息。

class multiprocessing.**Queue**([*maxsize*])
返回一个使用一个管道和少量锁和信号量实现的共享队列实例。当一个进程将一个对象放进队列中时，一个写入线程会启动并将对象从缓冲区写入管道中。

The usual *Queue.Empty* and *Queue.Full* exceptions from the standard library's *Queue* module are raised to signal timeouts.

Queue implements all the methods of *Queue.Queue* except for *task_done()* and *join()*.

qsize()
返回队列的大致长度。由于多线程或者多进程的上下文，这个数字是不可靠的。

注意，在 Unix 平台上，例如 Mac OS X，这个方法可能会抛出*NotImplementedError* 异常，因为该平台没有实现 sem_getvalue() 。

empty()
如果队列是空的，返回 True ，反之返回 False 。由于多线程或多进程的环境，该状态是不可靠的。

full()
如果队列是满的，返回 True ，反之返回 False 。由于多线程或多进程的环境，该状态是不可靠的。

put(*obj*[, *block*[, *timeout*]])
Put obj into the queue. If the optional argument *block* is True (the default) and *timeout* is None (the default), block if necessary until a free slot is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *Queue.Full* exception if no free slot was available within that time. Otherwise (*block* is False), put an item on the queue if a free slot is immediately available, else raise the *Queue.Full* exception (*timeout* is ignored in that case).

put_nowait(*obj*)
相当于 put(obj, False)。

get([*block*[, *timeout*]])
Remove and return an item from the queue. If optional args *block* is True (the default) and *timeout* is None (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *Queue.Empty* exception if no item was available within that time. Otherwise (block is False), return an item if one is immediately available, else raise the *Queue.Empty* exception (*timeout* is ignored in that case).

get_nowait()
相当于 get(False) 。

Queue has a few additional methods not found in *Queue.Queue*. These methods are usually unnecessary for most code:

close()
指示当前进程将不会再往队列中放入对象。一旦所有缓冲区中的数据被写入管道之后，后台的线程会退出。这个方法在队列被 gc 回收时会自动调用。

join_thread()
等待后台线程。这个方法仅在调用了*close()* 方法之后可用。这会阻塞当前进程，直到后台线程退出，确保所有缓冲区中的数据都被写入管道中。

默认情况下，如果一个不是队列创建者的进程试图退出，它会尝试等待这个队列的后台线程。这个进程可以使用*cancel_join_thread()* 让*join_thread()* 方法什么都不做直接跳过。

cancel_join_thread()
防止*join_thread()* 方法阻塞当前进程。具体而言，这防止进程退出时自动等待后台线程退出。详见*join_thread()*。

可能这个方法称为”allow_exit_without_flush()“会更好。这有可能会导致正在排队进入队列的数据丢失，大多数情况下你不需要用到这个方法，仅当你不关心底层管道中可能丢失的数据，只是希望进程能够马上退出时使用。

注解: 该类的功能依赖于宿主操作系统具有可用的共享信号量实现。否则该类将被禁用，任何试图实例化一个*Queue* 对象的操作都会抛出*ImportError* 异常，更多信息详见 bpo-3770 。后续说明的任何专用队列对象亦如此。

---

class multiprocessing.queues.**SimpleQueue**
It is a simplified *Queue* type, very close to a locked *Pipe*.

empty()
如果队列为空返回 True ， 否则返回 False 。

get()
从队列中移出并返回一个对象。

put(*item*)
将 *item* 放入队列。

class multiprocessing.**JoinableQueue**([*maxsize*])
JoinableQueue, a *Queue* subclass, is a queue which additionally has *task_done()* and *join()* methods.

task_done()
Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each *get()* used to fetch a task, a subsequent call to *task_done()* tells the queue that the processing on the task is complete.

If a *join()* is currently blocking, it will resume when all items have been processed (meaning that a *task_done()* call was received for every item that had been *put()* into the queue).

如果被调用的次数多于放入队列中的项目数量，将引发*ValueError* 异常。

join()
阻塞至队列中所有的元素都被接收和处理完毕。

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls *task_done()* to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, *join()* unblocks.

杂项

multiprocessing.**active_children**()
返回当前进程存活的子进程的列表。

调用该方法有“等待”已经结束的进程的副作用。

multiprocessing.**cpu_count**()
Return the number of CPUs in the system. May raise *NotImplementedError*.

multiprocessing.**current_process**()
返回与当前进程相对应的*Process* 对象。

和*threading.current_thread()* 相同。

multiprocessing.**freeze_support**()
为使用了*multiprocessing* 的程序，提供冻结以产生 Windows 可执行文件的支持。(在 **py2exe**, **PyInstaller** 和 **cx_Freeze** 上测试通过)

需要在 main 模块的 if __name__ == '__main__' 该行之后马上调用该函数。例如:

```
from multiprocessing import Process, freeze_support

def f():
    print 'hello world!'

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

如果没有调用 `freeze_support()` 在尝试运行被冻结的可执行文件时会抛出*RuntimeError* 异常。

对 `freeze_support()` 的调用在非 Windows 平台上是无效的。如果该模块在 Windows 平台的 Python 解释器中正常运行 (该程序没有被冻结)，调用 “freeze_support()“ 也是无效的。

`multiprocessing.`**`set_executable`**`()`
设置在启动子进程时使用的 Python 解释器路径。(默认使用*sys.executable*) 嵌入式编程人员可能需要这样做：

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

before they can create child processes. (Windows only)

注解： *multiprocessing* 并没有包含类似*threading.active_count()* , *threading.enumerate()* , *threading.settrace()* ， *threading.setprofile()*, *threading.Timer* ，或者*threading.local* 的方法和类。

连接对象（Connection）

Connection 对象允许收发可以序列化的对象或字符串。它们可以看作面向消息的连接套接字。

通常使用*Pipe* 创建 Connection 对象。详见：*监听者及客户端*.

`class Connection`

`send`(*obj*)
将一个对象发送到连接的另一端，可以用*recv()* 读取。

The object must be picklable. Very large pickles (approximately 32 MB+, though it depends on the OS) may raise a *ValueError* exception.

`recv`()
返回一个由另一端使用*send()* 发送的对象。该方法会一直阻塞直到接收到对象。如果对端关闭了连接或者没有东西可接收，将抛出*EOFError* 异常。

`fileno`()
返回由连接对象使用的描述符或者句柄。

`close`()
关闭连接对象。

当连接对象被垃圾回收时会自动调用。

`poll`([*timeout*])
返回连接对象中是否有可以读取的数据。

如果未指定 *timeout* ，此方法会马上返回。如果 *timeout* 是一个数字，则指定了最大阻塞的秒数。如果 *timeout* 是 `None` ，那么将一直等待，不会超时。

send_bytes(*buffer*[, *offset*[, *size*]])
Send byte data from an object supporting the buffer interface as a complete message.

If *offset* is given then data is read from that position in *buffer*. If *size* is given then that many bytes will be read from buffer. Very large buffers (approximately 32 MB+, though it depends on the OS) may raise a *ValueError* exception

recv_bytes([*maxlength*])
以字符串形式返回一条从连接对象另一端发送过来的字节数据。此方法在接收到数据前将一直阻塞。如果连接对象被对端关闭或者没有数据可读取，将抛出*EOFError* 异常。

If *maxlength* is specified and the message is longer than *maxlength* then *IOError* is raised and the connection will no longer be readable.

recv_bytes_into(*buffer*[, *offset*])
将一条完整的字节数据消息读入 *buffer* 中并返回消息的字节数。此方法在接收到数据前将一直阻塞。如果连接对象被对端关闭或者没有数据可读取，将抛出*EOFError* 异常。

buffer must be an object satisfying the writable buffer interface. If *offset* is given then the message will be written into the buffer from that position. Offset must be a non-negative integer less than the length of *buffer* (in bytes).

如果缓冲区太小，则将引发 `BufferTooShort` 异常，并且完整的消息将会存放在异常实例 `e` 的 `e.args[0]` 中。

例如:

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes('thank you')
>>> a.recv_bytes()
'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

警告： *Connection.recv()* 方法会自动解封它收到的数据，除非你能够信任发送消息的进程，否则此处可能有安全风险。

因此，除非连接对象是由 `Pipe()` 产生的，否则你应该仅在使用了某种认证手段之后才使用*recv()* 和*send()* 方法。参考认证密码。

警告： 如果一个进程在试图读写管道时被终止了，那么管道中的数据很可能是不完整的，因为此时可能无法确定消息的边界。

同步原语

通常来说同步原语在多进程环境中并不像它们在多线程环境中那么必要。参考*threading* 模块的文档。

注意可以使用管理器对象创建同步原语，参考数据管理器 。

class multiprocessing.**BoundedSemaphore**([*value*])
非常类似*threading.BoundedSemaphore* 的有界信号量对象。

A solitary difference from its close analog exists: its `acquire` method's first argument is named *block* and it supports an optional second argument *timeout*, as is consistent with *Lock.acquire()*.

注解: 在 Mac OS X 平台上，该对象于*Semaphore* 不同在于 `sem_getvalue()` 方法并没有在该平台上实现。

---

class multiprocessing.**Condition**([*lock*])
A condition variable: a clone of *threading.Condition*.

指定的 *lock* 参数应该是*multiprocessing* 模块中的*Lock* 或者*RLock* 对象。

class multiprocessing.**Event**
A clone of *threading.Event*. This method returns the state of the internal semaphore on exit, so it will always return `True` except if a timeout is given and the operation times out.

在 2.7 版更改: Previously, the method always returned `None`.

class multiprocessing.**Lock**
原始锁（非递归锁）对象，类似于*threading.Lock* 。一旦一个进程或者线程拿到了锁，后续的任何其他进程或线程的其他请求都会被阻塞直到锁被释放。任何进程或线程都可以释放锁。除非另有说明，否则*multiprocessing.Lock* 用于进程或者线程的概念和行为都和*threading.Lock* 一致。

注意*Lock* 实际上是一个工厂函数。它返回由默认上下文初始化的 `multiprocessing.synchronize.Lock` 对象。

Lock supports the *context manager* protocol and thus may be used in `with` statements.

acquire(*block=True*, *timeout=None*)
可以阻塞或非阻塞地获得锁。

如果 *block* 参数被设为 `True`（默认值），对该方法的调用在锁处于释放状态之前都会阻塞，然后将锁设置为锁住状态并返回 `True` 。需要注意的是第一个参数名与*threading.Lock.acquire()* 的不同。

如果 *block* 参数被设置成 `False` ，方法的调用将不会阻塞。如果锁当前处于锁住状态，将返回 `False` ；否则将锁设置成锁住状态，并返回 `True` 。

When invoked with a positive, floating-point value for *timeout*, block for at most the number of seconds specified by *timeout* as long as the lock can not be acquired. Invocations with a negative value for *timeout* are equivalent to a *timeout* of zero. Invocations with a *timeout* value of `None` (the default) set the timeout period to infinite. The *timeout* argument has no practical implications if the *block* argument is set to `False` and is thus ignored. Returns `True` if the lock has been acquired or `False` if the timeout period has elapsed. Note that the *timeout* argument does not exist in this method's analog, *threading.Lock.acquire()*.

release()
释放锁，可以在任何进程、线程使用，并不限于锁的拥有者。

其行为与*threading.Lock.release()* 一样，只不过当尝试释放一个没有被持有的锁时，会抛出*ValueError* 异常。

class multiprocessing.**RLock**
递归锁对象: 类似于*threading.RLock* 。递归锁必须由持有线程、进程亲自释放。如果某个进程或者

线程拿到了递归锁，这个进程或者线程可以再次拿到这个锁而不需要等待。但是这个进程或者线程的拿锁操作和释放锁操作的次数必须相同。

注意*RLock* 是一个工厂函数，调用后返回一个使用默认 context 初始化的 `multiprocessing.synchronize.RLock` 对象。

RLock 支持*context manager* 协议，因此可在 `with` 语句内使用。

acquire(*block=True*, *timeout=None*)
可以阻塞或非阻塞地获得锁。

当 *block* 设置为 `True` 时，会一直阻塞直到锁处于空闲状态（没有被任何进程、线程拥有），除非当前进程/线程已经拥有了这把锁。然后当前进程会持有这把锁（在锁没有被持有者的情况下），锁内的递归等级加一，并返回 `True`. 注意，这个函数第一个参数和*threading.RLock.acquire()* 有几个不同点，包括参数名本身。

当 *block* 参数是 `False`, 将不会阻塞，如果此时锁被其他进程或者线程持有，当前进程、线程获取锁操作失败，锁的递归等级也不会改变，函数返回 `False`, 如果当前锁已经处于释放状态，则当前进程、线程则会拿到锁，并且锁内的递归等级加一，函数返回 `True` 。

Use and behaviors of the *timeout* argument are the same as in *Lock.acquire()*. Note that the *timeout* argument does not exist in this method's analog, *threading.RLock.acquire()*.

release()
释放锁，使锁内的递归等级减一。如果释放后锁内的递归等级降低为 0，则会重置锁的状态为释放状态（即没有被任何进程、线程持有），重置后如果有有其他进程和线程在等待这把锁，他们中的一个会获得这个锁而继续运行。如果释放后锁内的递归等级还没到达 0，则这个锁仍将保持未释放状态且当前进程和线程仍然是持有者。

只有当前进程或线程是锁的持有者时, 才允许调用这个方法。如果当前进程或线程不是这个锁的拥有者, 或者这个锁处于已释放的状态 (即没有任何拥有者), 调用这个方法会抛出*AssertionError* 异常。注意这里抛出的异常类型和*threading.RLock.release()* 中实现的行为不一样。

class multiprocessing.**Semaphore**([*value*])
一种信号量对象: 类似于*threading.Semaphore*.

A solitary difference from its close analog exists: its `acquire` method's first argument is named *block* and it supports an optional second argument *timeout*, as is consistent with *Lock.acquire()*.

注解: The `acquire()` method of *BoundedSemaphore*, *Lock*, *RLock* and *Semaphore* has a timeout parameter not supported by the equivalents in *threading*. The signature is `acquire(block=True, timeout=None)` with keyword parameters being acceptable. If *block* is `True` and *timeout* is not `None` then it specifies a timeout in seconds. If *block* is `False` then *timeout* is ignored.

在 Mac OS X 上，不支持 `sem_timedwait` ，所以，使用调用 `acquire()` 时如果使用 timeout 参数，会通过循环 sleep 来模拟 timeout 的行为。

注解: 假如信号 SIGINT 是来自于 `Ctrl-C` ，并且主线程被 `BoundedSemaphore.acquire()`, *Lock.acquire()*, *RLock.acquire()*, `Semaphore.acquire()`, `Condition.acquire()` 或 `Condition.wait()` 阻塞，则调用会立即中断同时抛出*KeyboardInterrupt* 异常。

这和*threading* 的行为不同，此模块中当执行对应的阻塞式调用时，SIGINT 会被忽略。

注解: 这个库的某些功能依赖于宿主机系统的共享信号量，如果系统没有这个特性，`multiprocessing.synchronize` 会被禁用，尝试导入这个包会引发*ImportError* 异常，详细信息请查看 bpo-3770 。

共享 ctypes 对象

在共享内存上创建可被子进程继承的共享对象时是可行的。

multiprocessing.**Value**(*typecode_or_type*, **args*[, *lock*])
Return a *ctypes* object allocated from shared memory. By default the return value is actually a synchronized wrapper for the object.

typecode_or_type 指明了返回的对象类型: 它可能是一个 ctypes 类型或者*array* 模块中每个类型对应的单字符长度的字符串。**args* 会透传给这个类的构造函数。

如果 *lock* 参数是 `True` （默认值）, 将会新建一个递归锁用于同步对于此值的访问操作。如果 *lock* 是*Lock* 或者*RLock* 对象，那么这个传入的锁将会用于同步对这个值的访问操作，如果 *lock* 是 `False` , 那么对这个对象的访问将没有锁保护，也就是说这个变量不是进程安全的。

诸如 += 这类的操作会引发独立的读操作和写操作，也就是说这类操作符并不具有原子性。所以，如果你想让递增操作具有原子性，这样的方式并不能达到要求:

```
counter.value += 1
```

假设共享对象内部关联的锁时递归锁 (默认情况下就是)，那么你可以采用这种方式

```
with counter.get_lock():
    counter.value += 1
```

注意 *locl* 只能是命名参数。

multiprocessing.**Array**(*typecode_or_type*, *size_or_initializer*, *, *lock=True*)
从共享内存中申请并返回一个具有 ctypes 类型的数组对象。默认情况下返回值实际上是被同步器包装过的数组对象。

typecode_or_type 指明了返回的数组中的元素类型: 它可能是一个 ctypes 类型或者*array* 模块中每个类型对应的单字符长度的字符串。如果 *size_or_initializer* 是一个整数，那就会当做数组的长度，并且整个数组的内存会初始化为 0。否则，如果 *size_or_initializer* 会被当成一个序列用于初始化数组中的内一个元素，并且会根据元素个数自动判断数组的长度。

如果 *lock* 为 `True` (默认值) 则将创建一个新的锁对象用于同步对值的访问。如果 *lock* 为一个*Lock* 或*RLock* 对象则该对象将被用于同步对值的访问。如果 *lock* 为 `False` 则对返回对象的访问将不会自动得到锁的保护，也就是说它不是“进程安全的”。

请注意 *lock* 是一个仅限关键字参数。

请注意*ctypes.c_char* 的数组具有 *value* 和 *raw* 属性，允许被用来保存和提取字符串。

multiprocessing.sharedctypes 模块

multiprocessing.sharedctypes 模块提供了一些函数，用于分配来自共享内存的、可被子进程继承的*ctypes* 对象。

注解: 虽然可以将指针存储在共享内存中，但请记住它所引用的是特定进程地址空间中的位置。而且，指针很可能在第二个进程的上下文中无效，尝试从第二个进程对指针进行解引用可能会导致崩溃。

---

multiprocessing.sharedctypes.**RawArray**(*typecode_or_type*, *size_or_initializer*)
从共享内存中申请并返回一个 ctypes 数组。

typecode_or_type 指明了返回的数组中的元素类型: 它可能是一个 ctypes 类型或者*array* 模块中使用的类型字符。如果 *size_or_initializer* 是一个整数，那就会当做数组的长度，并且整个数组的内存会初始化

为 0。否则，如果 *size_or_initializer* 会被当成一个序列用于初始化数组中的每一个元素，并且会根据元素个数自动判断数组的长度。

注意对元素的访问、赋值操作可能是非原子操作 - 使用*Array()* 来借助其中的锁保证操作的原子性。

multiprocessing.sharedctypes.**RawValue**(*typecode_or_type*, **args*)
从共享内存中申请并返回一个 ctypes 对象。

typecode_or_type 指明了返回的对象类型: 它可能是一个 ctypes 类型或者*array* 模块中每个类型对应的单字符长度的字符串。**args* 会透传给这个类的构造函数。

注意对 value 的访问、赋值操作可能是非原子操作 - 使用*Value()* 来借助其中的锁保证操作的原子性。

请注意*ctypes.c_char* 的数组具有 *value* 和 *raw* 属性，允许被用来保存和提取字符串 - 请查看*ctypes* 文档。

multiprocessing.sharedctypes.**Array**(*typecode_or_type*, *size_or_initializer*, **args*[, *lock*])
返回一个纯 ctypes 数组, 或者在此之上经过同步器包装过的对象，这取决于 *lock* 参数的值，除此之外，和*RawArray()* 一样。

如果 *lock* 为 `True` (默认值) 则将创建一个新的锁对象用于同步对值的访问。如果 *lock* 为一个*Lock* 或*RLock* 对象则该对象将被用于同步对值的访问。如果 *lock* 为 `False` 则对所返回对象的访问将不会自动得到锁的保护，也就是说它将不是“进程安全的”。

注意 *locl* 只能是命名参数。

multiprocessing.sharedctypes.**Value**(*typecode_or_type*, **args*[, *lock*])
返回一个纯 ctypes 数组, 或者在此之上经过同步器包装过的进程安全的对象，这取决于 *lock* 参数的值，除此之外，和 `RawArray()` 一样。

如果 *lock* 为 `True` (默认值) 则将创建一个新的锁对象用于同步对值的访问。如果 *lock* 为一个*Lock* 或*RLock* 对象则该对象将被用于同步对值的访问。如果 *lock* 为 `False` 则对所返回对象的访问将不会自动得到锁的保护，也就是说它将不是“进程安全的”。

注意 *locl* 只能是命名参数。

multiprocessing.sharedctypes.**copy**(*obj*)
从共享内存中申请一片空间将 ctypes 对象 *obj* 过来，然后返回一个新的 ctypes 对象。

multiprocessing.sharedctypes.**synchronized**(*obj*[, *lock*])
将一个 ctypes 对象包装为进程安全的对象并返回，使用 *lock* 同步对于它的操作。如果 *lock* 是 `None` (默认值)，则会自动创建一个*multiprocessing.RLock* 对象。

同步器包装后的对象会在原有对象基础上额外增加两个方法: `get_obj()` 返回被包装的对象，`get_lock()` 返回内部用于同步的锁。

需要注意的是，访问包装后的 ctypes 对象会比直接访问原来的纯 ctypes 对象慢得多。

下面的表格对比了创建普通 ctypes 对象和基于共享内存上创建共享 ctypes 对象的语法。(表格中的 `MyStruct` 是*ctypes.Structure* 的子类)

ctypes	使用类型的共享 ctypes	使用 typecode 的共享 ctypes
c_double(2.4)	RawValue(c_double, 2.4)	RawValue(‘d’, 2.4)
MyStruct(4, 6)	RawValue(MyStruct, 4, 6)	
(c_short * 7)()	RawArray(c_short, 7)	RawArray(‘h’, 7)
(c_int * 3)(9, 2, 8)	RawArray(c_int, (9, 2, 8))	RawArray(‘i’, (9, 2, 8))

下面是一个在子进程中修改多个 ctypes 对象的例子。

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', 'hello world', lock=lock)
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()

    print n.value
    print x.value
    print s.value
    print [(a.x, a.y) for a in A]
```

输出如下

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

数据管理器

Managers provide a way to create data which can be shared between different processes. A manager object controls a server process which manages *shared objects*. Other processes can access the shared objects by using proxies.

multiprocessing.**Manager**()
返回一个已启动的*SyncManager* 管理器对象，这个对象可以用于在不同进程中共享数据。返回的管理器对象对应了一个 spawned 方式启动的子进程，并且拥有一系列方法可以用于创建共享对象、返回对应的代理。

当管理器被垃圾回收或者父进程退出时，管理器进程会立即退出。管理器类定义在*multiprocessing.managers* 模块:

class multiprocessing.managers.**BaseManager**([*address*[, *authkey*]])
创建一个 BaseManager 对象。

一旦创建，应该及时调用*start()* 或者 get_server().serve_forever() 以确保管理器对象对应的管理进程已经启动。

address 是管理器对象监听的地址。如果 *address* 是 None，则允许和任意主机的请求建立连接。

authkey is the authentication key which will be used to check the validity of incoming connections to the server process. If *authkey* is None then `current_process().authkey`. Otherwise *authkey* is used and it must be a string.

start([*initializer*[, *initargs*]])
为管理器开启一个子进程，如果 *initializer* 不是 None，子进程在启动时将会调用 `initializer(*initargs)`。

get_server()
返回一个 Server 对象，它是管理器在后台控制的真实的服务。Server 对象拥有 `serve_forever()` 方法。

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000), authkey='abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

Server 额外拥有一个*address* 属性。

connect()
将本地管理器对象连接到一个远程管理器进程:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 5000), authkey='abc')
>>> m.connect()
```

shutdown()
停止管理器的进程。这个方法只能用于已经使用*start()* 启动的服务进程。

它可以被多次调用。

register(*typeid*[, *callable*[, *proxytype*[, *exposed*[, *method_to_typeid*[, *create_method*]]]]])
一个 classmethod，可以将一个类型或者可调用对象注册到管理器类。

typeid 是一种“类型标识”，用于唯一表示某种共享类型，必须是一个字符串。

callable is a callable used for creating objects for this type identifier. If a manager instance will be created using the `from_address()` classmethod or if the *create_method* argument is `False` then this can be left as None.

proxytype 是*BaseProxy* 的子类，可以根据 *typeid* 为共享对象创建一个代理，如果是 None，则会自动创建一个代理类。

exposed 是一个函数名组成的序列，用来指明只有这些方法可以使用*BaseProxy._callmethod()* 代理。(如果 *exposed* 是 None，则会在 `proxytype._exposed_` 存在的情况下转而使用它) 当暴露的方法列表没有指定的时候，共享对象的所有“公共方法”都会被代理。(这里的“公共方法”是指所有拥有 `__call__()` 方法并且不是以 `'_'` 开头的属性)

method_to_typeid 是一个映射，用来指定那些应该返回代理对象的暴露方法所返回的类型。(如果 *method_to_typeid* 是 None，则 `proxytype._method_to_typeid_` 会在存在的情况下被使用) 如果方法名称不在这个映射中或者映射是 None，则方法返回的对象会是一个值拷贝。

create_method 指明，是否要创建一个以 *typeid* 命名并返回一个代理对象的函数，这个函数会被服务进程用于创建共享对象，默认为 True。

BaseManager 实例也有一个只读属性。

address
管理器所用的地址。

class multiprocessing.managers.**SyncManager**
BaseManager 的子类，可用于进程的同步。这个类型的对象使用 multiprocessing.Manager() 创建。

It also supports creation of shared lists and dictionaries.

BoundedSemaphore([*value*])
创建一个共享的*threading.BoundedSemaphore* 对象并返回它的代理。

Condition([*lock*])
创建一个共享的*threading.Condition* 对象并返回它的代理。

如果提供了 *lock* 参数，那它必须是*threading.Lock* 或*threading.RLock* 的代理对象。

Event()
创建一个共享的*threading.Event* 对象并返回它的代理。

Lock()
创建一个共享的*threading.Lock* 对象并返回它的代理。

Namespace()
创建一个共享的 Namespace` 对象并返回它的代理。

Queue([*maxsize*])
Create a shared *Queue.Queue* object and return a proxy for it.

RLock()
创建一个共享的*threading.RLock* 对象并返回它的代理。

Semaphore([*value*])
创建一个共享的*threading.Semaphore* 对象并返回它的代理。

Array(*typecode*, *sequence*)
创建一个数组并返回它的代理。

Value(*typecode*, *value*)
创建一个具有可写 value 属性的对象并返回它的代理。

dict()
dict(*mapping*)
dict(*sequence*)
Create a shared dict object and return a proxy for it.

list()
list(*sequence*)
Create a shared list object and return a proxy for it.

注解： Modifications to mutable values or items in dict and list proxies will not be propagated through the manager, because the proxy has no way of knowing when its values or items are modified. To modify such an item, you can re-assign the modified object to the container proxy:

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# reassigning the dictionary, the proxy is notified of the change
lproxy[0] = d
```

class multiprocessing.managers.**Namespace**
一个可以注册到*SyncManager*的类型。

命名空间对象没有公共方法，但是拥有可写的属性。它的表示 (repr) 会显示所有属性的值。

值得一提的是，当对命名空间对象使用代理的时候，访问所有名称以 '_' 开头的属性都只是代理器上的属性，而不是命名空间对象的属性。

```
>>> manager = multiprocessing.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3    # this is an attribute of the proxy
>>> print Global
Namespace(x=10, y='hello')
```

自定义管理器

要创建一个自定义的管理器，需要新建一个*BaseManager*的子类，然后使用这个管理器类上的*register()*类方法将新类型或者可调用方法注册上去。例如:

```
from multiprocessing.managers import BaseManager

class MathsClass(object):
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    manager = MyManager()
    manager.start()
    maths = manager.Maths()
    print maths.add(4, 3)         # prints 7
    print maths.mul(7, 8)         # prints 56
```

使用远程管理器

可以将管理器服务运行在一台机器上，然后使用客户端从其他机器上访问。(假设它们的防火墙允许这样的网络通信)

运行下面的代码可以启动一个服务，此付包含了一个共享队列，允许远程客户端访问:

```
>>> from multiprocessing.managers import BaseManager
>>> import Queue
>>> queue = Queue.Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
```

(下页继续)

(续上页)

```
>>> m = QueueManager(address=('', 50000), authkey='abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

远程客户端可以通过下面的方式访问服务:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey='abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

也可以通过下面的方式:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey='abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

本地进程也可以访问这个队列，利用上面的客户端代码通过远程方式访问:

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super(Worker, self).__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey='abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

代理对象

代理是一个 指向其他共享对象的对象，这个对象 (很可能) 在另外一个进程中。共享对象也可以说是代理 指涉的对象。多个代理对象可能指向同一个指涉对象。

A proxy object has methods which invoke corresponding methods of its referent (although not every method of the referent will necessarily be available through the proxy). A proxy can usually be used in most of the same ways that its referent can:

```
>>> from multiprocessing import Manager
>>> manager = Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print l
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print repr(l)
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

注意，对代理使用*str()* 函数会返回指涉对象的字符串表示，但是*repr()* 却会返回代理本身的内部字符串表示。

An important feature of proxy objects is that they are picklable so they can be passed between processes. Note, however, that if a proxy is sent to the corresponding manager’s process then unpickling it will produce the referent itself. This means, for example, that one shared object can contain a second:

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)         # referent of a now contains referent of b
>>> print a, b
[[]] []
>>> b.append('hello')
>>> print a, b
[['hello']] ['hello']
```

注解: *multiprocessing* 中的代理类并没有提供任何对于代理值比较的支持。所以，我们会得到如下结果:

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

当需要比较值的时候，应该替换为使用指涉对象的拷贝。

---

class multiprocessing.managers.**BaseProxy**

代理对象是*BaseProxy* 派生类的实例。

_callmethod(*methodname*[, *args*[, *kwds*]])

调用指涉对象的方法并返回结果。

如果 proxy 是一个代理且其指涉的是 obj，那么下面的表达式:

```
proxy._callmethod(methodname, args, kwds)
```

相当于求取以下表达式的值:

```
getattr(obj, methodname)(*args, **kwds)
```

于管理器进程。

返回结果会是一个值拷贝或者一个新的共享对象的代理 - 见函数*BaseManager.register()* 中关于参数 *method_to_typeid* 的文档。

如果这个调用熬出了异常，则这个异常会被*_callmethod()* 透传出来。如果是管理器进程本身抛出的一些其他异常，则会被*_callmethod()* 转换为 RemoteError 异常重新抛出。

特别注意，如果 *methodname* 没有 暴露出来，将会引发一个异常。

_callmethod() 的一个使用示例:

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getslice__', (2, 7))   # equiv to `l[2:7]`
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))     # equiv to `l[20]`
Traceback (most recent call last):
...
IndexError: list index out of range
```

_getvalue()
返回指涉对象的一份拷贝。

如果指涉对象无法序列化，则会抛出一个异常。

__repr__()
返回代理对象的字符串表示。

__str__()
返回指涉对象的字符串表示。

清理

代理对象使用了一个弱引用回调，当它被垃圾回收时，会将自己从拥有此指涉对象的管理器上反注册，

当共享对象没有被任何代理器引用时，会被管理器进程删除。

进程池

可以创建一个进程池，它将使用 Pool 类执行提交给它的任务。

class multiprocessing.**Pool**([*processes*[, *initializer*[, *initargs*[, *maxtasksperchild*]]]])
一个进程池对象，它控制可以提交作业的工作进程池。它支持带有超时和回调的异步结果，以及一个并行的 map 实现。

processes is the number of worker processes to use. If *processes* is None then the number returned by cpu_count() is used. If *initializer* is not None then each worker process will call initializer(*initargs) when it starts.

注意，进程池对象的方法只有创建它的进程能够调用。

2.7 新版功能: *maxtasksperchild* 是一个工作进程在它退出或被一个新的工作进程代替之前能完成的任务数量，为了释放未使用的资源。默认的 *maxtasksperchild* 是 None，意味着工作进程寿与池齐。

注解: 通常来说，Pool 中的 Worker 进程的生命周期和进程池的工作队列一样长。一些其他系统中（如 Apache, mod_wsgi 等）也可以发现另一种模式，他们会让工作进程在完成一些任务后退出，清理、释放资源，然后启动一个新的进程代替旧的工作进程。Pool 的 *maxtasksperchild* 参数给用户提供了这种能力。

apply(*func*[, *args*[, *kwds*]])
Equivalent of the *apply()* built-in function. It blocks until the result is ready, so *apply_async()* is better suited for performing work in parallel. Additionally, *func* is only executed in one of the workers of the pool.

apply_async(*func*[, *args*[, *kwds*[, *callback*]]])
apply() 方法的一个变种，返回一个结果对象。

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it (unless the call failed). *callback* should complete immediately since otherwise the thread which handles the results will get blocked.

map(*func*, *iterable*[, *chunksize*])
A parallel equivalent of the *map()* built-in function (it supports only one *iterable* argument though). It blocks until the result is ready.

这个方法会将可迭代对象分割为许多块，然后提交给进程池。可以将 *chunksize* 设置为一个正整数从而（近似）指定每个块的大小可以。

map_async(*func*, *iterable*[, *chunksize*[, *callback*]])
和*map()* 方法类似，但是返回一个结果对象。

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it (unless the call failed). *callback* should complete immediately since otherwise the thread which handles the results will get blocked.

imap(*func*, *iterable*[, *chunksize*])
An equivalent of *itertools.imap()*.

chunksize 参数的作用和*map()* 方法的一样。对于很长的迭代器，给 *chunksize* 设置一个很大的值会比默认值 1 **极大**地加快执行速度。

同样，如果 *chunksize* 是 1，那么*imap()* 方法所返回的迭代器的 next() 方法拥有一个可选的 *timeout* 参数：如果无法在 *timeout* 秒内执行得到结果，则 “next(timeout)“ 会抛出 multiprocessing.TimeoutError 异常。

imap_unordered(*func*, *iterable*[, *chunksize*])
和*imap()* 相同，只不过通过迭代器返回的结果是任意的。（当进程池中只有一个工作进程的时候，返回结果的顺序才能认为是”有序”的）

close()
阻止后续任务提交到进程池，当所有任务执行完成后，工作进程会退出。

terminate()
不必等待未完成的任务，立即停止工作进程。当进程池对象呗垃圾回收时，*terminate()* 会立即调用。

join()
等待工作进程结束。调用*join()* 前必须先调用*close()* 或者*terminate()* 。

class multiprocessing.pool.**AsyncResult**
Pool.apply_async() 和 Pool.map_async() 返回对象所属的类。

get([*timeout*])
用于获取执行结果。如果 *timeout* 不是 None 并且在 *timeout* 秒内仍然没有执行完得到结果，则抛出

multiprocessing.TimeoutError 异常。如果远程调用发生异常，这个异常会通过*get()* 重新抛出。

wait([*timeout*])
阻塞，直到返回结果，或者 *timeout* 秒后超时。

ready()
用于判断执行状态，是否已经完成。

successful()
Return whether the call completed without raising an exception. Will raise *AssertionError* if the result is not ready.

下面的例子演示了进程池的用法:

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    pool = Pool(processes=4)              # start 4 worker processes

    result = pool.apply_async(f, (10,))   # evaluate "f(10)" asynchronously in a␣
↪single process
    print result.get(timeout=1)           # prints "100" unless your computer is␣
↪*very* slow

    print pool.map(f, range(10))          # prints "[0, 1, 4,..., 81]"

    it = pool.imap(f, range(10))
    print it.next()                       # prints "0"
    print it.next()                       # prints "1"
    print it.next(timeout=1)              # prints "4" unless your computer is *very*␣
↪slow

    result = pool.apply_async(time.sleep, (10,))
    print result.get(timeout=1)           # raises multiprocessing.TimeoutError
```

监听者及客户端

Usually message passing between processes is done using queues or by using *Connection* objects returned by *Pipe()*.

However, the *multiprocessing.connection* module allows some extra flexibility. It basically gives a high level message oriented API for dealing with sockets or Windows named pipes, and also has support for *digest authentication* using the *hmac* module.

multiprocessing.connection.**deliver_challenge**(*connection*, *authkey*)
发送一个随机生成的消息到另一端，并等待回复。

If the reply matches the digest of the message using *authkey* as the key then a welcome message is sent to the other end of the connection. Otherwise *AuthenticationError* is raised.

multiprocessing.connection.**answer_challenge**(*connection*, *authkey*)
接收一条信息，使用 *authkey* 作为键计算信息摘要，然后将摘要发送回去。

If a welcome message is not received, then *AuthenticationError* is raised.

multiprocessing.connection.**Client**(*address*[, *family*[, *authenticate*[, *authkey*]]])
Attempt to set up a connection to the listener which is using address *address*, returning a *Connection*.

连接的类型取决于 *family* 参数，但是通常可以省略，因为可以通过 *address* 的格式推导出来。(查看地址格式)

If *authenticate* is `True` or *authkey* is a string then digest authentication is used. The key used for authentication will be either *authkey* or `current_process().authkey)` if *authkey* is `None`. If authentication fails then *AuthenticationError* is raised. See 认证密码.

class multiprocessing.connection.**Listener**([*address*[, *family*[, *backlog*[, *authenticate*[, *authkey*]]]]])
可以监听连接请求，是对于绑定套接字或者 Windows 命名管道的封装。

address 是监听器对象中的绑定套接字或命名管道使用的地址。

注解: 如果使用‘0.0.0.0’作为监听地址，那么在 Windows 上这个地址无法建立连接。想要建立一个可连接的端点，应该使用‘127.0.0.1’。

---

family 是套接字 (或者命名管道) 使用的类型。它可以是以下一种: `'AF_INET'` (TCP 套接字类型), `'AF_UNIX'` (Unix 域套接字) 或者 `'AF_PIPE'` (Windows 命名管道)。其中只有第一个保证各平台可用。如果 *family* 是 `None` , 那么 family 会根据 *address* 的格式自动推导出来。如果 *address* 也是 `None` , 则取默认值。默认值为可用类型中速度最快的。见地址格式 。注意，如果 *family* 是 `'AF_UNIX'` 而 address 是 “None“ , 套接字会在一个*tempfile.mkstemp()* 创建的私有临时目录中创建。

如果监听器对象使用了套接字，*backlog* (默认值为 1) 会在套接字绑定后传递给它的*listen()* 方法。

If *authenticate* is `True` (`False` by default) or *authkey* is not `None` then digest authentication is used.

If *authkey* is a string then it will be used as the authentication key; otherwise it must be `None`.

If *authkey* is `None` and *authenticate* is `True` then `current_process().authkey` is used as the authentication key. If *authkey* is `None` and *authenticate* is `False` then no authentication is done. If authentication fails then *AuthenticationError* is raised. See 认证密码.

accept()
Accept a connection on the bound socket or named pipe of the listener object and return a *Connection* object. If authentication is attempted and fails, then `AuthenticationError` is raised.

close()
关闭监听器上的绑定套接字或者命名管道。此函数会在监听器被垃圾回收后自动调用。不过仍然建议显式调用函数关闭。

监听器对象拥有下列只读属性:

address
被监听器对象使用的地址。

last_accepted
最后一个连接所使用的地址。如果没有的话就是 `None` 。

The module defines the following exceptions:

exception multiprocessing.connection.**ProcessError**
所有*multiprocessing* 异常的基类。

exception multiprocessing.connection.**BufferTooShort**
当提供的缓冲区对象太小而无法读取消息时，*Connection.recv_bytes_into()* 引发的异常。

exception multiprocessing.connection.**AuthenticationError**
出现身份验证错误时引发。

exception multiprocessing.connection.**TimeoutError**
有超时的方法超时时引发。

示例

下面的服务代码创建了一个使用 'secret password' 作为认证密码的监听器。它会等待连接然后发送一些数据给客户端:

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)      # family is deduced to be 'AF_INET'
listener = Listener(address, authkey='secret password')

conn = listener.accept()
print 'connection accepted from', listener.last_accepted

conn.send([2.25, None, 'junk', float])

conn.send_bytes('hello')

conn.send_bytes(array('i', [42, 1729]))

conn.close()
listener.close()
```

下面的代码连接到服务然后从服务器上 j 接收一些数据:

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)
conn = Client(address, authkey='secret password')

print conn.recv()                 # => [2.25, None, 'junk', float]

print conn.recv_bytes()            # => 'hello'

arr = array('i', [0, 0, 0, 0, 0])
print conn.recv_bytes_into(arr)     # => 8
print arr                         # => array('i', [42, 1729, 0, 0, 0])

conn.close()
```

地址格式

- 'AF_INET' 地址是（主机, 端口）形式的元组类型，其中 主机是一个字符串，端口是整数。
- 'AF_UNIX' 地址是文件系统上文件名的字符串。
- **'AF_PIPE' 是这种格式的字符串** r'\.\pipe{PipeName}' 。如果要用*Client()* 连接到一个名为 *ServerName* 的远程命名管道，应该替换为使用 r'*ServerName*\pipe{PipeName}' 这种格式。

注意，使用两个反斜线开头的字符串默认被当做 'AF_PIPE' 地址而不是 'AF_UNIX' 。

认证密码

When one uses *Connection.recv()*, the data received is automatically unpickled. Unfortunately unpickling data from an untrusted source is a security risk. Therefore *Listener* and *Client()* use the *hmac* module to provide digest authentication.

An authentication key is a string which can be thought of as a password: once a connection is established both ends will demand proof that the other knows the authentication key. (Demonstrating that both ends are using the same key does **not** involve sending the key over the connection.)

如果要求认证但是没有指定认证密钥，则会使用 current_process().authkey 的返回值 (参见*Process*)。这个值将被当前进程所创建的任何*Process* 对象自动继承。这意味着 (在默认情况下) 一个包含多进程的程序中的所有进程会在相互间建立连接的时候共享单个认证密钥。

os.urandom() 也可以用来生成合适的认证密钥。

日志记录

当前模块也提供了一些对 logging 的支持。注意，*logging* 模块本身并没有使用进程间共享的锁，所以来自于多个进程的日志可能（具体取决于使用的日志 handler）相互覆盖或者混杂。

multiprocessing.**get_logger**()
返回*multiprocessing* 使用的 logger，必要的话会创建一个新的。

如果创建的首个 logger 日志级别为 logging.NOTSET 并且没有默认 handler。通过这个 logger 打印的消息不会传递到根 logger。

注意在 Windows 上，子进程只会继承父进程 logger 的日志级别 - 对于 logger 的其他自定义项不会继承。

multiprocessing.**log_to_stderr**()
此函数会调用*get_logger()* 但是会在返回的 logger 上增加一个 handler，将所有输出都使用 '[%(levelname)s/%(processName)s] %(message)s' 的格式发送到*sys.stderr* 。

下面是一个在交互式解释器中打开日志功能的例子:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

In addition to having these two logging functions, the multiprocessing also exposes two additional logging level attributes. These are SUBWARNING and SUBDEBUG. The table below illustrates where theses fit in the normal level hierarchy.

Level	Numeric value
SUBWARNING	25
SUBDEBUG	5

要查看日志等级的完整列表，见*logging* 模块。

These additional logging levels are used primarily for certain debug messages within the multiprocessing module. Below is the same example as above, except with SUBDEBUG enabled:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(multiprocessing.SUBDEBUG)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../pymp-djGBXN/listener-...'
>>> del m
[SUBDEBUG/MainProcess] finalizer calling ...
[INFO/MainProcess] sending shutdown message to manager
[DEBUG/SyncManager-...] manager received shutdown message
[SUBDEBUG/SyncManager-...] calling <Finalize object, callback=unlink, ...
[SUBDEBUG/SyncManager-...] finalizer calling <built-in function unlink> ...
[SUBDEBUG/SyncManager-...] calling <Finalize object, dead>
[SUBDEBUG/SyncManager-...] finalizer calling <function rmtree at 0x5aa730> ...
[INFO/SyncManager-...] manager exiting with exitcode 0
```

multiprocessing.dummy 模块

multiprocessing.dummy 复制了*multiprocessing* 的 API，不过是在*threading* 模块之上包装了一层。

16.6.3 编程指导

使用*multiprocessing* 时，应遵循一些指导原则和习惯用法。

All platforms

避免共享状态

应该尽可能避免在进程间传递大量数据，越少越好。

It is probably best to stick to using queues or pipes for communication between processes rather than using the lower level synchronization primitives from the *threading* module.

可序列化

保证所代理的方法的参数是可以序列化的。

代理的线程安全性

不要在多线程中同时使用一个代理对象，除非你用锁保护它。

（而在不同进程中使用 相同的代理对象从不会发生问题。）

使用 Join 避免僵尸进程

在 Unix 上，如果一个进程执行完成但是没有被 join，就会变成僵尸进程。一般来说，僵尸进程不会很多，因为每次新启动进程（或者*active_children()* 被调用）时，所有已执行完成且没有被 join 的进程都会自动被 join，而且对一个执行完的进程调用*Process.is_alive* 也会 join 这个进程。尽管如此，对自己启动的进程显式调用 join 依然是最佳实践。

继承优于序列化、反序列化

On Windows many types from *multiprocessing* need to be picklable so that child processes can use them. However, one should generally avoid sending shared objects to other processes using pipes or queues. Instead you should arrange the program so that a process which needs access to a shared resource created elsewhere can inherit it from an ancestor process.

避免杀死进程

听过*Process.terminate* 停止一个进程很容易导致这个进程正在使用的共享资源（如锁、信号量、管道和队列）损坏或者变得不可用，无法在其他进程中继续使用。

所以，最好只对那些从来不使用共享资源的进程调用*Process.terminate* 。

Join 使用队列的进程

Bear in mind that a process that has put items in a queue will wait before terminating until all the buffered items are fed by the "feeder" thread to the underlying pipe. (The child process can call the *cancel_join_thread()* method of the queue to avoid this behaviour.)

这意味着，任何使用队列的时候，你都要确保在进程 join 之前，所有存放到队列中的项将会被其他进程、线程完全消费。否则不能保证这个写过队列的进程可以正常终止。记住非精灵进程会自动 join 。

下面是一个会导致死锁的例子:

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                    # this deadlocks
    obj = queue.get()
```

交换最后两行可以修复这个问题（或者直接删掉 `p.join()`）。

显示传递资源给子进程

On Unix a child process can make use of a shared resource created in a parent process using a global resource. However, it is better to pass the object as an argument to the constructor for the child process.

Apart from making the code (potentially) compatible with Windows this also ensures that as long as the child process is still alive the object will not be garbage collected in the parent process. This might be important if some resource is freed when the object is garbage collected in the parent process.

所以对于实例:

```
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

应当重写成这样:

```
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

谨防将*sys.stdin*数据替换为“类似文件的对象”

*multiprocessing*内部会无条件地这样调用:

```
os.close(sys.stdin.fileno())
```

在multiprocessing.Process._bootstrap()方法中——这会导致与”进程中的进程”相关的一些问题。这已经被修改成了:

```
sys.stdin.close()
sys.stdin = open(os.devnull)
```

它解决了进程相互冲突导致文件描述符错误的根本问题，但是对使用带缓冲的“文件类对象”替换*sys.stdin()*作为输出的应用程序造成了潜在的危险。如果多个进程调用了此文件类对象的*close()*方法，会导致相同的数据多次刷写到此对象，损坏数据。

如果你写入文件类对象并实现了自己的缓存，可以在每次追加缓存数据时记录当前进程 id，从而将其变成 fork 安全的，当发现进程 id 变化后舍弃之前的缓存，例如:

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

需要更多信息，请查看 bpo-5155, bpo-5313 以及 bpo-5331

Windows

Since Windows lacks *os.fork()* it has a few extra restrictions:

更依赖序列化

Ensure that all arguments to Process.__init__() are picklable. This means, in particular, that bound or unbound methods cannot be used directly as the target argument on Windows —just define a function and use that instead.

Also, if you subclass *Process* then make sure that instances will be picklable when the *Process.start* method is called.

全局变量

记住，如果子进程中的代码尝试访问一个全局变量，它所看到的值可能和父进程中执行*Process.start*那一刻的值不一样。

当全局变量知识模块级别的常量时，是不会有问题的。

安全导入主模块

确保主模块可以被新启动的 Python 解释器安全导入而不会引发什么副作用（比如又启动了一个子进程）

For example, under Windows running the following module would fail with a *RuntimeError*:

```
from multiprocessing import Process

def foo():
    print 'hello'

p = Process(target=foo)
p.start()
```

应该通过下面的方法使用 `if __name__ == '__main__':`，从而保护程序”入口点”：

```
from multiprocessing import Process, freeze_support

def foo():
    print 'hello'

if __name__ == '__main__':
    freeze_support()
    p = Process(target=foo)
    p.start()
```

（如果程序将正常运行而不是冻结，则可以省略 `freeze_support()` 行）

这允许新启动的 Python 解释器安全导入模块然后运行模块中的 `foo()` 函数。

如果主模块中创建了进程池或者管理器，这个规则也适用。

16.6.4 例子

创建和使用自定义管理器、代理的示例:

```
#
# This module shows how to use arbitrary callables with a subclass of
# `BaseManager`.
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo(object):
    def f(self):
        print 'you called Foo.f()'
    def g(self):
        print 'you called Foo.g()'
    def _h(self):
        print 'you called Foo._h()'
```

(下页继续)

(续上页)

```
# A simple generator function
def baz():
    for i in xrange(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ('next', '__next__')
    def __iter__(self):
        return self
    def next(self):
        return self._callmethod('next')
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)

##

def test():
    manager = MyManager()
    manager.start()

    print '-' * 20

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print '-' * 20

    f2 = manager.Foo2()
    f2.g()
    f2._h()
```

(下页继续)

(续上页)

```
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print '-' * 20

    it = manager.baz()
    for i in it:
        print '<%d>' % i,
    print

    print '-' * 20

    op = manager.operator()
    print 'op.add(23, 45) =', op.add(23, 45)
    print 'op.pow(2, 94) =', op.pow(2, 94)
    print 'op.getslice(range(10), 2, 6) =', op.getslice(range(10), 2, 6)
    print 'op.repeat(range(5), 3) =', op.repeat(range(5), 3)
    print 'op._exposed_ =', op._exposed_

##

if __name__ == '__main__':
    freeze_support()
    test()
```

使用 Pool:

```
#
# A test of `multiprocessing.Pool` class
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import multiprocessing
import time
import random
import sys

#
# Functions used by test code
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
        )

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b
```

(下页继续)

(续上页)

```
def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

def f(x):
    return 1.0 / (x-5.0)

def pow3(x):
    return x**3

def noop(x):
    pass

#
# Test code
#

def test():
    print 'cpu_count() = %d\n' % multiprocessing.cpu_count()

    #
    # Create pool
    #

    PROCESSES = 4
    print 'Creating pool with %d processes\n' % PROCESSES
    pool = multiprocessing.Pool(PROCESSES)
    print 'pool = %s' % pool
    print

    #
    # Tests
    #

    TASKS = [(mul, (i, 7)) for i in range(10)] + \
            [(plus, (i, 8)) for i in range(10)]

    results = [pool.apply_async(calculate, t) for t in TASKS]
    imap_it = pool.imap(calculatestar, TASKS)
    imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

    print 'Ordered results using pool.apply_async():'
    for r in results:
        print '\t', r.get()
    print

    print 'Ordered results using pool.imap():'
    for x in imap_it:
        print '\t', x
    print

    print 'Unordered results using pool.imap_unordered():'
    for x in imap_unordered_it:
        print '\t', x
    print
```

(下页继续)

(续上页)

```
    print 'Ordered results using pool.map() --- will block till complete:'
    for x in pool.map(calculatestar, TASKS):
        print '\t', x
    print

    #
    # Simple benchmarks
    #

    N = 100000
    print 'def pow3(x): return x**3'

    t = time.time()
    A = map(pow3, xrange(N))
    print '\tmap(pow3, xrange(%d)):\n\t\t%s seconds' % \
          (N, time.time() - t)

    t = time.time()
    B = pool.map(pow3, xrange(N))
    print '\tpool.map(pow3, xrange(%d)):\n\t\t%s seconds' % \
          (N, time.time() - t)

    t = time.time()
    C = list(pool.imap(pow3, xrange(N), chunksize=N//8))
    print '\tlist(pool.imap(pow3, xrange(%d), chunksize=%d)):\n\t\t%s' \
          ' seconds' % (N, N//8, time.time() - t)

    assert A == B == C, (len(A), len(B), len(C))
    print

    L = [None] * 1000000
    print 'def noop(x): pass'
    print 'L = [None] * 1000000'

    t = time.time()
    A = map(noop, L)
    print '\tmap(noop, L):\n\t\t%s seconds' % \
          (time.time() - t)

    t = time.time()
    B = pool.map(noop, L)
    print '\tpool.map(noop, L):\n\t\t%s seconds' % \
          (time.time() - t)

    t = time.time()
    C = list(pool.imap(noop, L, chunksize=len(L)//8))
    print '\tlist(pool.imap(noop, L, chunksize=%d)):\n\t\t%s seconds' % \
          (len(L)//8, time.time() - t)

    assert A == B == C, (len(A), len(B), len(C))
    print

    del A, B, C, L

    #
```

(下页继续)

(续上页)

```
    # Test error handling
    #

    print 'Testing error handling:'

    try:
        print pool.apply(f, (5,))
    except ZeroDivisionError:
        print '\tGot ZeroDivisionError as expected from pool.apply()'
    else:
        raise AssertionError('expected ZeroDivisionError')

    try:
        print pool.map(f, range(10))
    except ZeroDivisionError:
        print '\tGot ZeroDivisionError as expected from pool.map()'
    else:
        raise AssertionError('expected ZeroDivisionError')

    try:
        print list(pool.imap(f, range(10)))
    except ZeroDivisionError:
        print '\tGot ZeroDivisionError as expected from list(pool.imap())'
    else:
        raise AssertionError('expected ZeroDivisionError')

    it = pool.imap(f, range(10))
    for i in range(10):
        try:
            x = it.next()
        except ZeroDivisionError:
            if i == 5:
                pass
        except StopIteration:
            break
        else:
            if i == 5:
                raise AssertionError('expected ZeroDivisionError')

    assert i == 9
    print '\tGot ZeroDivisionError as expected from IMapIterator.next()'
    print

    #
    # Testing timeouts
    #

    print 'Testing ApplyResult.get() with timeout:',
    res = pool.apply_async(calculate, TASKS[0])
    while 1:
        sys.stdout.flush()
        try:
            sys.stdout.write('\n\t%s' % res.get(0.02))
            break
        except multiprocessing.TimeoutError:
            sys.stdout.write('.')
```

(下页继续)

(续上页)

```
    print
    print

    print 'Testing IMapIterator.next() with timeout:',
    it = pool.imap(calculatestar, TASKS)
    while 1:
        sys.stdout.flush()
        try:
            sys.stdout.write('\n\t%s' % it.next(0.02))
        except StopIteration:
            break
        except multiprocessing.TimeoutError:
            sys.stdout.write('.')
    print
    print

    #
    # Testing callback
    #

    print 'Testing callback:'

    A = []
    B = [56, 0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

    r = pool.apply_async(mul, (7, 8), callback=A.append)
    r.wait()

    r = pool.map_async(pow3, range(10), callback=A.extend)
    r.wait()

    if A == B:
        print '\tcallbacks succeeded\n'
    else:
        print '\t*** callbacks failed\n\t\t%s != %s\n' % (A, B)

    #
    # Check there are no outstanding tasks
    #

    assert not pool._cache, 'cache = %r' % pool._cache

    #
    # Check close() methods
    #

    print 'Testing close():'

    for worker in pool._pool:
        assert worker.is_alive()

    result = pool.apply_async(time.sleep, [0.5])
    pool.close()
    pool.join()

    assert result.get() is None
```

(下页继续)

(续上页)

```
    for worker in pool._pool:
        assert not worker.is_alive()

    print '\tclose() succeeded\n'

    #
    # Check terminate() method
    #

    print 'Testing terminate():'

    pool = multiprocessing.Pool(2)
    DELTA = 0.1
    ignore = pool.apply(pow3, [2])
    results = [pool.apply_async(time.sleep, [DELTA]) for i in range(100)]
    pool.terminate()
    pool.join()

    for worker in pool._pool:
        assert not worker.is_alive()

    print '\tterminate() succeeded\n'

    #
    # Check garbage collection
    #

    print 'Testing garbage collection:'

    pool = multiprocessing.Pool(2)
    DELTA = 0.1
    processes = pool._pool
    ignore = pool.apply(pow3, [2])
    results = [pool.apply_async(time.sleep, [DELTA]) for i in range(100)]

    results = pool = None

    time.sleep(DELTA * 2)

    for worker in processes:
        assert not worker.is_alive()

    print '\tgarbage collection succeeded\n'


if __name__ == '__main__':
    multiprocessing.freeze_support()

    assert len(sys.argv) in (1, 2)

    if len(sys.argv) == 1 or sys.argv[1] == 'processes':
        print ' Using processes '.center(79, '-')
    elif sys.argv[1] == 'threads':
        print ' Using threads '.center(79, '-')
        import multiprocessing.dummy as multiprocessing
```

(下页继续)

(续上页)

```
    else:
        print 'Usage:\n\t%s [processes | threads]' % sys.argv[0]
        raise SystemExit(2)

    test()
```

Synchronization types like locks, conditions and queues:

```
#
# A test file for the `multiprocessing` package
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import time, sys, random
from Queue import Empty

import multiprocessing                # may get overwritten


#### TEST_VALUE

def value_func(running, mutex):
    random.seed()
    time.sleep(random.random()*4)

    mutex.acquire()
    print '\n\t\t\t' + str(multiprocessing.current_process()) + ' has finished'
    running.value -= 1
    mutex.release()

def test_value():
    TASKS = 10
    running = multiprocessing.Value('i', TASKS)
    mutex = multiprocessing.Lock()

    for i in range(TASKS):
        p = multiprocessing.Process(target=value_func, args=(running, mutex))
        p.start()

    while running.value > 0:
        time.sleep(0.08)
        mutex.acquire()
        print running.value,
        sys.stdout.flush()
        mutex.release()

    print
    print 'No more running processes'


#### TEST_QUEUE

def queue_func(queue):
    for i in range(30):
```

(下页继续)

(续上页)

```
        time.sleep(0.5 * random.random())
        queue.put(i*i)
    queue.put('STOP')

def test_queue():
    q = multiprocessing.Queue()

    p = multiprocessing.Process(target=queue_func, args=(q,))
    p.start()

    o = None
    while o != 'STOP':
        try:
            o = q.get(timeout=0.3)
            print o,
            sys.stdout.flush()
        except Empty:
            print 'TIMEOUT'

    print


#### TEST_CONDITION

def condition_func(cond):
    cond.acquire()
    print '\t' + str(cond)
    time.sleep(2)
    print '\tchild is notifying'
    print '\t' + str(cond)
    cond.notify()
    cond.release()

def test_condition():
    cond = multiprocessing.Condition()

    p = multiprocessing.Process(target=condition_func, args=(cond,))
    print cond

    cond.acquire()
    print cond
    cond.acquire()
    print cond

    p.start()

    print 'main is waiting'
    cond.wait()
    print 'main has woken up'

    print cond
    cond.release()
    print cond
    cond.release()

    p.join()
```

(下页继续)

(续上页)

```
    print cond


#### TEST_SEMAPHORE

def semaphore_func(sema, mutex, running):
    sema.acquire()

    mutex.acquire()
    running.value += 1
    print running.value, 'tasks are running'
    mutex.release()

    random.seed()
    time.sleep(random.random()*2)

    mutex.acquire()
    running.value -= 1
    print '%s has finished' % multiprocessing.current_process()
    mutex.release()

    sema.release()

def test_semaphore():
    sema = multiprocessing.Semaphore(3)
    mutex = multiprocessing.RLock()
    running = multiprocessing.Value('i', 0)

    processes = [
        multiprocessing.Process(target=semaphore_func,
                                args=(sema, mutex, running))
        for i in range(10)
        ]

    for p in processes:
        p.start()

    for p in processes:
        p.join()


#### TEST_JOIN_TIMEOUT

def join_timeout_func():
    print '\tchild sleeping'
    time.sleep(5.5)
    print '\n\tchild terminating'

def test_join_timeout():
    p = multiprocessing.Process(target=join_timeout_func)
    p.start()

    print 'waiting for process to finish'

    while 1:
        p.join(timeout=1)
```

(下页继续)

(续上页)

```
        if not p.is_alive():
            break
        print '.',
        sys.stdout.flush()


#### TEST_EVENT

def event_func(event):
    print '\t%r is waiting' % multiprocessing.current_process()
    event.wait()
    print '\t%r has woken up' % multiprocessing.current_process()

def test_event():
    event = multiprocessing.Event()

    processes = [multiprocessing.Process(target=event_func, args=(event,))
                 for i in range(5)]

    for p in processes:
        p.start()

    print 'main is sleeping'
    time.sleep(2)

    print 'main is setting event'
    event.set()

    for p in processes:
        p.join()


#### TEST_SHAREDVALUES

def sharedvalues_func(values, arrays, shared_values, shared_arrays):
    for i in range(len(values)):
        v = values[i][1]
        sv = shared_values[i].value
        assert v == sv

    for i in range(len(values)):
        a = arrays[i][1]
        sa = list(shared_arrays[i][:])
        assert a == sa

    print 'Tests passed'

def test_sharedvalues():
    values = [
        ('i', 10),
        ('h', -2),
        ('d', 1.25)
        ]
    arrays = [
        ('i', range(100)),
        ('d', [0.25 * i for i in range(100)]),
```

(下页继续)

(续上页)

```
        ('H', range(1000))
        ]

    shared_values = [multiprocessing.Value(id, v) for id, v in values]
    shared_arrays = [multiprocessing.Array(id, a) for id, a in arrays]

    p = multiprocessing.Process(
        target=sharedvalues_func,
        args=(values, arrays, shared_values, shared_arrays)
        )
    p.start()
    p.join()

    assert p.exitcode == 0


####

def test(namespace=multiprocessing):
    global multiprocessing

    multiprocessing = namespace

    for func in [ test_value, test_queue, test_condition,
                  test_semaphore, test_join_timeout, test_event,
                  test_sharedvalues ]:

        print '\n\t######## %s\n' % func.__name__
        func()

    ignore = multiprocessing.active_children()      # cleanup any old processes
    if hasattr(multiprocessing, '_debug_info'):
        info = multiprocessing._debug_info()
        if info:
            print info
            raise ValueError('there should be no positive refcounts left')


if __name__ == '__main__':
    multiprocessing.freeze_support()

    assert len(sys.argv) in (1, 2)

    if len(sys.argv) == 1 or sys.argv[1] == 'processes':
        print ' Using processes '.center(79, '-')
        namespace = multiprocessing
    elif sys.argv[1] == 'manager':
        print ' Using processes and a manager '.center(79, '-')
        namespace = multiprocessing.Manager()
        namespace.Process = multiprocessing.Process
        namespace.current_process = multiprocessing.current_process
        namespace.active_children = multiprocessing.active_children
    elif sys.argv[1] == 'threads':
        print ' Using threads '.center(79, '-')
        import multiprocessing.dummy as namespace
    else:
```

(下页继续)

(续上页)

```
        print 'Usage:\n\t%s [processes | manager | threads]' % sys.argv[0]
        raise SystemExit(2)

    test(namespace)
```

一个演示如何使用队列来向一组工作进程提供任务并收集结果的例子：

```
#
# Simple example which uses a pool of workers to carry out some tasks.
#
# Notice that the results will probably not come out of the output
# queue in the same in the same order as the corresponding tasks were
# put on the input queue.  If it is important to get the results back
# in the original order then consider using `Pool.map()` or
# `Pool.imap()` (which will save on the amount of code needed anyway).
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import time
import random

from multiprocessing import Process, Queue, current_process, freeze_support

#
# Function run by worker processes
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)

#
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
```

(下页继续)

(续上页)

```
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print 'Unordered results:'
    for i in range(len(TASKS1)):
        print '\t', done_queue.get()

    # Add more tasks using `put()`
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
        print '\t', done_queue.get()

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')


if __name__ == '__main__':
    freeze_support()
    test()
```

An example of how a pool of worker processes can each run a `SimpleHTTPServer.HttpServer` instance while sharing a single listening socket.

```
#
# Example where a pool of http servers share a single listening socket
#
# On Windows this module depends on the ability to pickle a socket
# object so that the worker processes can inherit a copy of the server
# object.  (We import `multiprocessing.reduction` to enable this pickling.)
#
# Not sure if we should synchronize access to `socket.accept()` method by
# using a process-shared lock -- does not seem to be necessary.
#
# Copyright (c) 2006-2008, R Oudkerk
```

(下页继续)

(续上页)

```
# All rights reserved.
#

import os
import sys

from multiprocessing import Process, current_process, freeze_support
from BaseHTTPServer import HTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler

if sys.platform == 'win32':
    import multiprocessing.reduction    # make sockets pickable/inheritable


def note(format, *args):
    sys.stderr.write('[%s]\t%s\n' % (current_process().name, format%args))


class RequestHandler(SimpleHTTPRequestHandler):
    # we override log_message() to show which process is handling the request
    def log_message(self, format, *args):
        note(format, *args)

def serve_forever(server):
    note('starting server')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        pass


def runpool(address, number_of_processes):
    # create a single server object -- children will each inherit a copy
    server = HTTPServer(address, RequestHandler)

    # create child processes to act as workers
    for i in range(number_of_processes-1):
        Process(target=serve_forever, args=(server,)).start()

    # main process also acts as a worker
    serve_forever(server)


def test():
    DIR = os.path.join(os.path.dirname(__file__), '..')
    ADDRESS = ('localhost', 8000)
    NUMBER_OF_PROCESSES = 4

    print 'Serving at http://%s:%d using %d worker processes' % \
          (ADDRESS[0], ADDRESS[1], NUMBER_OF_PROCESSES)
    print 'To exit press Ctrl-' + ['C', 'Break'][sys.platform=='win32']

    os.chdir(DIR)
    runpool(ADDRESS, NUMBER_OF_PROCESSES)
```

(下页继续)

(续上页)

```
if __name__ == '__main__':
    freeze_support()
    test()
```

Some simple benchmarks comparing *multiprocessing* with *threading*:

```
#
# Simple benchmarks for the multiprocessing package
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#

import time, sys, multiprocessing, threading, Queue, gc

if sys.platform == 'win32':
    _timer = time.clock
else:
    _timer = time.time

delta = 1


#### TEST_QUEUESPEED

def queuespeed_func(q, c, iterations):
    a = '0' * 256
    c.acquire()
    c.notify()
    c.release()

    for i in xrange(iterations):
        q.put(a)

    q.put('STOP')

def test_queuespeed(Process, q, c):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        p = Process(target=queuespeed_func, args=(q, c, iterations))
        c.acquire()
        p.start()
        c.wait()
        c.release()

        result = None
        t = _timer()

        while result != 'STOP':
            result = q.get()

        elapsed = _timer() - t
```

(下页继续)

(续上页)

```
        p.join()

    print iterations, 'objects passed through the queue in', elapsed, 'seconds'
    print 'average number/sec:', iterations/elapsed


#### TEST_PIPESPEED

def pipe_func(c, cond, iterations):
    a = '0' * 256
    cond.acquire()
    cond.notify()
    cond.release()

    for i in xrange(iterations):
        c.send(a)

    c.send('STOP')

def test_pipespeed():
    c, d = multiprocessing.Pipe()
    cond = multiprocessing.Condition()
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        p = multiprocessing.Process(target=pipe_func,
                                    args=(d, cond, iterations))
        cond.acquire()
        p.start()
        cond.wait()
        cond.release()

        result = None
        t = _timer()

        while result != 'STOP':
            result = c.recv()

        elapsed = _timer() - t
        p.join()

    print iterations, 'objects passed through connection in',elapsed,'seconds'
    print 'average number/sec:', iterations/elapsed


#### TEST_SEQSPEED

def test_seqspeed(seq):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
```

(下页继续)

(续上页)

```
        iterations *= 2

        t = _timer()

        for i in xrange(iterations):
            a = seq[5]

        elapsed = _timer()-t

    print iterations, 'iterations in', elapsed, 'seconds'
    print 'average number/sec:', iterations/elapsed


#### TEST_LOCK

def test_lockspeed(l):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        t = _timer()

        for i in xrange(iterations):
            l.acquire()
            l.release()

        elapsed = _timer()-t

    print iterations, 'iterations in', elapsed, 'seconds'
    print 'average number/sec:', iterations/elapsed


#### TEST_CONDITION

def conditionspeed_func(c, N):
    c.acquire()
    c.notify()

    for i in xrange(N):
        c.wait()
        c.notify()

    c.release()

def test_conditionspeed(Process, c):
    elapsed = 0
    iterations = 1

    while elapsed < delta:
        iterations *= 2

        c.acquire()
        p = Process(target=conditionspeed_func, args=(c, iterations))
        p.start()
```

(下页继续)

(续上页)

```
        c.wait()

        t = _timer()

        for i in xrange(iterations):
            c.notify()
            c.wait()

        elapsed = _timer()-t

        c.release()
        p.join()

    print iterations * 2, 'waits in', elapsed, 'seconds'
    print 'average number/sec:', iterations * 2 / elapsed

####

def test():
    manager = multiprocessing.Manager()

    gc.disable()

    print '\n\t######## testing Queue.Queue\n'
    test_queuespeed(threading.Thread, Queue.Queue(),
                    threading.Condition())
    print '\n\t######## testing multiprocessing.Queue\n'
    test_queuespeed(multiprocessing.Process, multiprocessing.Queue(),
                    multiprocessing.Condition())
    print '\n\t######## testing Queue managed by server process\n'
    test_queuespeed(multiprocessing.Process, manager.Queue(),
                    manager.Condition())
    print '\n\t######## testing multiprocessing.Pipe\n'
    test_pipespeed()

    print

    print '\n\t######## testing list\n'
    test_seqspeed(range(10))
    print '\n\t######## testing list managed by server process\n'
    test_seqspeed(manager.list(range(10)))
    print '\n\t######## testing Array("i", ..., lock=False)\n'
    test_seqspeed(multiprocessing.Array('i', range(10), lock=False))
    print '\n\t######## testing Array("i", ..., lock=True)\n'
    test_seqspeed(multiprocessing.Array('i', range(10), lock=True))

    print

    print '\n\t######## testing threading.Lock\n'
    test_lockspeed(threading.Lock())
    print '\n\t######## testing threading.RLock\n'
    test_lockspeed(threading.RLock())
    print '\n\t######## testing multiprocessing.Lock\n'
    test_lockspeed(multiprocessing.Lock())
    print '\n\t######## testing multiprocessing.RLock\n'
```

(下页继续)

(续上页)

```
    test_lockspeed(multiprocessing.RLock())
    print '\n\t######## testing lock managed by server process\n'
    test_lockspeed(manager.Lock())
    print '\n\t######## testing rlock managed by server process\n'
    test_lockspeed(manager.RLock())

    print

    print '\n\t######## testing threading.Condition\n'
    test_conditionspeed(threading.Thread, threading.Condition())
    print '\n\t######## testing multiprocessing.Condition\n'
    test_conditionspeed(multiprocessing.Process, multiprocessing.Condition())
    print '\n\t######## testing condition managed by a server process\n'
    test_conditionspeed(multiprocessing.Process, manager.Condition())

    gc.enable()

if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()
```

16.7 mmap —内存映射文件支持

Memory-mapped file objects behave like both strings and like file objects. Unlike normal string objects, however, these are mutable. You can use mmap objects in most places where strings are expected; for example, you can use the *re* module to search through a memory-mapped file. Since they' re mutable, you can change a single character by doing `obj[index] = 'a'`, or change a substring by assigning to a slice: `obj[i1:i2] = '...'`. You can also read and write data starting at the current file position, and `seek()` through the file to different positions.

内存映射文件是由*mmap* 构造函数创建的，其在 Unix 和 Windows 上是不同的。无论哪种情况，你都必须为一个打开的文件提供文件描述符以进行更新。如果你希望映射一个已有的 Python 文件对象，请使用该对象的 `fileno()` 方法来获取 *fileno* 参数的正确值。否则，你可以使用*os.open()* 函数来打开这个文件，这会直接返回一个文件描述符（结束时仍然需要关闭该文件）。

注解： 如果要为可写的缓冲文件创建内存映射，则应当首先*flush()* 该文件。这确保了对缓冲区的本地修改在内存映射中可用。

For both the Unix and Windows versions of the constructor, *access* may be specified as an optional keyword parameter. *access* accepts one of three values: `ACCESS_READ`, `ACCESS_WRITE`, or `ACCESS_COPY` to specify read-only, write-through or copy-on-write memory respectively. *access* can be used on both Unix and Windows. If *access* is not specified, Windows mmap returns a write-through mapping. The initial memory values for all three access types are taken from the specified file. Assignment to an `ACCESS_READ` memory map raises a *TypeError* exception. Assignment to an `ACCESS_WRITE` memory map affects both memory and the underlying file. Assignment to an `ACCESS_COPY` memory map affects memory but does not update the underlying file.

在 2.5 版更改: 要映射匿名内存，应将 -1 作为 fileno 和 length 一起传递。

在 2.6 版更改: mmap.mmap has formerly been a factory function creating mmap objects. Now mmap.mmap is the class itself.

class mmap.**mmap**(*fileno*, *length*[, *tagname*[, *access*[, *offset*]]])

（**Windows 版本**）映射被文件句柄 *fileno* 指定的文件的 *length* 个字节，并创建一个 mmap 对象。如果

length 大于当前文件大小，则文件将扩展为包含 *length* 个字节。如果 *length* 为 0，则映射的最大长度为当前文件大小。如果文件为空，Windows 会引发异常（你无法在 Windows 上创建空映射）。

如果 *tagname* 被指定且不是 None ，则是为映射提供标签名称的字符串。Windows 允许你对同一文件拥有许多不同的映射。如果指定现有标签的名称，则会打开该标签，否则将创建该名称的新标签。如果省略此参数或设置为 None ，则创建的映射不带名称。避免使用 tag 参数将有助于使代码在 Unix 和 Windows 之间可移植。

offset 可以被指定为非负整数偏移量。mmap 引用将相对于从文件开头的偏移。*offset* 默认为 0。*offeset* 必须是 ALLOCATIONGRANULARITY 的倍数。

class mmap.**mmap**(*fileno*, *length*[, *flags*[, *prot*[, *access*[, *offset*]]]])

(Unix 版本) 映射文件描述符 *fileno* 指定的文件的 *length* 个字节，并返回一个 mmap 对象。如果 *length* 为 0 ，则当调用*mmap* 时，映射的最大长度将为文件的当前大小。

flags specifies the nature of the mapping. MAP_PRIVATE creates a private copy-on-write mapping, so changes to the contents of the mmap object will be private to this process, and MAP_SHARED creates a mapping that's shared with all other processes mapping the same areas of the file. The default value is MAP_SHARED.

prot, if specified, gives the desired memory protection; the two most useful values are PROT_READ and PROT_WRITE, to specify that the pages may be read or written. *prot* defaults to PROT_READ | PROT_WRITE.

access may be specified in lieu of *flags* and *prot* as an optional keyword parameter. It is an error to specify both *flags*, *prot* and *access*. See the description of *access* above for information on how to use this parameter.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of ALLOCATIONGRANULARITY which is equal to PAGESIZE on Unix systems.

To ensure validity of the created memory mapping the file specified by the descriptor *fileno* is internally automatically synchronized with physical backing store on Mac OS X and OpenVMS.

This example shows a simple way of using *mmap*:

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write("Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print mm.readline()  # prints "Hello Python!"
    # read content via slice notation
    print mm[:5]  # prints "Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = " world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print mm.readline()  # prints "Hello  world!"
    # close the map
    mm.close()
```

The next example demonstrates how to create an anonymous map and exchange data between the parent and child processes:

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write("Hello world!")

pid = os.fork()

if pid == 0:  # In a child process
    mm.seek(0)
    print mm.readline()

    mm.close()
```

Memory-mapped file objects support the following methods:

close()
Closes the mmap. Subsequent calls to other methods of the object will result in a ValueError exception being raised. This will not close the open file.

find(*string*[, *start*[, *end*]])
Returns the lowest index in the object where the substring *string* is found, such that *string* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns `-1` on failure.

flush([*offset*, *size*])
Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If *offset* and *size* are specified, only changes to the given range of bytes will be flushed to disk; otherwise, the whole extent of the mapping is flushed. *offset* must be a multiple of the `PAGESIZE` or `ALLOCATIONGRANULARITY`.

(Windows version) A nonzero value returned indicates success; zero indicates failure.

(Unix 版本) 返回零值以表示成功。当调用失败时将引发异常。

move(*dest*, *src*, *count*)
Copy the *count* bytes starting at offset *src* to the destination index *dest*. If the mmap was created with `ACCESS_READ`, then calls to move will raise a *TypeError* exception.

read(*num*)
Return a string containing up to *num* bytes starting from the current file position; the file position is updated to point after the bytes that were returned.

read_byte()
Returns a string of length 1 containing the character at the current file position, and advances the file position by 1.

readline()
Returns a single line, starting at the current file position and up to the next newline.

resize(*newsize*)
Resizes the map and the underlying file, if any. If the mmap was created with `ACCESS_READ` or `ACCESS_COPY`, resizing the map will raise a *TypeError* exception.

rfind(*string*[, *start*[, *end*]])
Returns the highest index in the object where the substring *string* is found, such that *string* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns `-1` on failure.

seek(*pos*[, *whence*])
Set the file's current position. *whence* argument is optional and defaults to `os.SEEK_SET` or `0` (absolute file

positioning); other values are `os.SEEK_CUR` or `1` (seek relative to the current position) and `os.SEEK_END` or `2` (seek relative to the file's end).

size()
Return the length of the file, which can be larger than the size of the memory-mapped area.

tell()
Returns the current position of the file pointer.

write(*string*)
Write the bytes in *string* into memory at the current position of the file pointer; the file position is updated to point after the bytes that were written. If the mmap was created with `ACCESS_READ`, then writing to it will raise a *TypeError* exception.

write_byte(*byte*)
Write the single-character string *byte* into memory at the current position of the file pointer; the file position is advanced by `1`. If the mmap was created with `ACCESS_READ`, then writing to it will raise a *TypeError* exception.

16.8 readline —GNU readline 接口

The *readline* module defines a number of functions to facilitate completion and reading/writing of history files from the Python interpreter. This module can be used directly, or via the *rlcompleter* module, which supports completion of Python identifiers at the interactive prompt. Settings made using this module affect the behaviour of both the interpreter's interactive prompt and the prompts offered by the *raw_input()* and *input()* built-in functions.

注解： The underlying Readline library API may be implemented by the `libedit` library instead of GNU readline. On MacOS X the *readline* module detects which library is being used at run time.

`libedit` 所用的配置文件与 GNU readline 的不同。如果你要在程序中载入配置字符串你可以在 `readline.__doc__` 中检测文本“libedit”来区分 GNU readline 和 libedit。

Readline keybindings may be configured via an initialization file, typically `.inputrc` in your home directory. See Readline Init File in the GNU Readline manual for information about the format and allowable constructs of that file, and the capabilities of the Readline library in general.

16.8.1 初始化文件

下列函数与初始化文件和用户配置有关：

readline.**parse_and_bind**(*string*)
执行在 *string* 参数中提供的初始化行。此函数会调用底层库中的 `rl_parse_and_bind()`。

readline.**read_init_file**([*filename*])
执行一个 readline 初始化文件。默认文件名为最近所使用的文件名。此函数会调用底层库中的 `rl_read_init_file()`。

16.8.2 行缓冲区

下列函数会在行缓冲区上操作。

readline.**get_line_buffer**()
返回行缓冲区的当前内容 (底层库中的 rl_line_buffer)。

readline.**insert_text**(*string*)
将文本插入行缓冲区的当前游标位置。该函数会调用底层库中的 rl_insert_text()，但会忽略其返回值。

readline.**redisplay**()
改变屏幕的显示以反映行缓冲区的当前内容。该函数会调用底层库中的 rl_redisplay()。

16.8.3 历史文件

下列函数会在历史文件上操作：

readline.**read_history_file**([*filename*])
载入一个 readline 历史文件，并将其添加到历史列表。默认文件名为 ~/.history。此函数会调用底层库中的 read_history()。

readline.**write_history_file**([*filename*])
将历史列表保存为 readline 历史文件，覆盖任何现有文件。默认文件名为 ~/.history。此函数会调用底层库中的 write_history()。

readline.**get_history_length**()
readline.**set_history_length**(*length*)
设置或返回需要保存到历史文件的行数。*write_history_file()* 函数会通过调用底层库中的 history_truncate_file() 以使用该值来截取历史文件。负值意味着不限制历史文件的大小。

16.8.4 历史列表

以下函数会在全局历史列表上操作：

readline.**clear_history**()
清除当前历史。此函数会调用底层库的 clear_history()。此 Python 函数仅当 Python 编译包带有支持此功能的库版本时才会存在。

2.4 新版功能.

readline.**get_current_history_length**()
返回历史列表的当前项数。(此函数不同于*get_history_length()*，后者是返回将被写入历史文件的最大行数。)

2.3 新版功能.

readline.**get_history_item**(*index*)
返回序号为 *index* 的历史条目的当前内容。条目序号从一开始。此函数会调用底层库中的 history_get()。

2.3 新版功能.

readline.**remove_history_item**(*pos*)
从历史列表中移除指定位置上的历史条目。条目位置从零开始。此函数会调用底层库中的 remove_history()。

2.4 新版功能.

readline.**replace_history_item**(*pos*, *line*)
将指定位置上的历史条目替换为 *line*。条目位置从零开始。此函数会调用底层库中的 replace_history_entry()。

2.4 新版功能.

readline.**add_history**(*line*)
将 *line* 添加到历史缓冲区，相当于是最近输入的一行。此函数会调用底层库中的 add_history()。

16.8.5 启动钩子

2.3 新版功能.

readline.**set_startup_hook**([*function*])
设置或移除底层库的 rl_startup_hook 回调所发起调用的函数。如果指定了 *function*，它将被用作新的钩子函数；如果省略或为 None，任何已安装的函数将被移除。钩子函数将在 readline 打印第一个提示信息之前不带参数地被调用。

readline.**set_pre_input_hook**([*function*])
设置或移除底层库的 rl_pre_input_hook 回调所发起调用的函数。如果指定了 *function*，它将被用作新的钩子函数；如果省略或为 None，任何已安装的函数将被移除。钩子函数将在打印第一个提示信息之后、readline 开始读取输入字符之前不带参数地被调用。此函数仅当 Python 编译包带有支持此功能的库版本时才会存在。

16.8.6 Completion

以下函数与自定义单词补全函数的实现有关。这通常使用 Tab 键进行操作，能够提示并自动补全正在输入的单词。默认情况下，Readline 设置为由*rlcompleter* 来补全交互模式解释器的 Python 标识符。如果*readline* 模块要配合自定义的补全函数来使用，则需要设置不同的单词分隔符。

readline.**set_completer**([*function*])
设置或移除补全函数。如果指定了 *function*，它将被用作新的补全函数；如果省略或为 None，任何已安装的补全函数将被移除。补全函数的调用形式为 function(text, state)，其中 *state* 为 0, 1, 2, …, 直至其返回一个非字符串值。它应当返回下一个以 *text* 开头的候选补全内容。

已安装的补全函数将由传递给底层库中 rl_completion_matches() 的 *entry_func* 回调函数来发起调用。*text* 字符串来自于底层库中 rl_attempted_completion_function 回调函数的第一个形参。

readline.**get_completer**()
获取补全函数，如果没有设置补全函数则返回 None。

2.3 新版功能.

readline.**get_completion_type**()
获取正在尝试的补全类型。此函数会将底层库中的 rl_completion_type 变量作为一个整数返回。

2.6 新版功能.

readline.**get_begidx**()
readline.**get_endidx**()
获取补全域的开始和结束序号。这些序号就是传给底层库中 rl_attempted_completion_function 回调函数的 *start* 和 *end* 参数。

readline.**set_completer_delims**(*string*)
readline.**get_completer_delims**()
设置或获取补全的单词分隔符。此分隔符确定了要考虑补全的单词的开始和结束位置（补全域）。这些函数会访问底层库的 rl_completer_word_break_characters 变量。

readline.**set_completion_display_matches_hook**([*function*])
设置或移除补全显示函数。如果指定了 *function*，它将被用作新的补全显示函数；如果省略或为 None，任何已安装的补全显示函数将被移除。此函数会设置或清除底层库的 rl_completion_display_matches_hook 回调函数。补全显示函数会在每次需要显示匹配项时以 function(substitution, [matches], longest_match_length) 的形式被调用。

2.6 新版功能.

16.8.7 示例

The following example demonstrates how to use the *readline* module's history reading and writing functions to automatically load and save a history file named .pyhist from the user's home directory. The code below would normally be executed automatically during interactive sessions from the user's PYTHONSTARTUP file.

```
import os
import readline
histfile = os.path.join(os.path.expanduser("~"), ".pyhist")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    readline.set_history_length(1000)
except IOError:
    pass
import atexit
atexit.register(readline.write_history_file, histfile)
del os, histfile
```

以下示例扩展了*code.InteractiveConsole*类以支持历史保存/恢复。

```
import code
import readline
import atexit
import os

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except IOError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```

16.9 rlcompleter —GNU readline 的补全函数

源代码: Lib/rlcompleter.py

---

rlcompeleter 通过补全有效的 Python 标识符和关键字定义了一个适用于*readline* 模块的补全函数。

当此模块在具有可用的*readline* 模块的 Unix 平台被导入, 一个 Completer 实例将被自动创建并且它的 complete() 方法将设置为*readline* 的补全器.

示例:

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__          readline.get_line_buffer(  readline.read_init_file(
readline.__file__         readline.insert_text(      readline.set_completer(
readline.__name__         readline.parse_and_bind(
>>> readline.
```

The *rlcompleter* module is designed for use with Python's interactive mode. A user can add the following lines to his or her initialization file (identified by the PYTHONSTARTUP environment variable) to get automatic Tab completion:

```
try:
    import readline
except ImportError:
    print "Module readline not available."
else:
    import rlcompleter
    readline.parse_and_bind("tab: complete")
```

在没有*readline* 的平台, 此模块定义的 Completer 类仍然可以用于自定义行为.

16.9.1 Completer 对象

Completer 对象具有以下方法:

Completer.**complete**(*text*, *state*)
为 *text* 返回第 *state* 项补全。

If called for *text* that doesn't include a period character (`'.'`), it will complete from names currently defined in *__main__*, *__builtin__* and keywords (as defined by the *keyword* module).

如果为带有句点的名称执行调用，它将尝试尽量求值直到最后一部分为止而不产生附带影响（函数不会被求值，但它可以生成对 __getattr__() 的调用），并通过*dir()* 函数来匹配剩余部分。在对表达式求值期间引发的任何异常都会被捕获、静默处理并返回*None*。

CHAPTER 17

Interprocess Communication and Networking

The modules described in this chapter provide mechanisms for different processes to communicate.

Some modules only work for two processes that are on the same machine, e.g. *signal* and *subprocess*. Other modules support networking protocols that two or more processes can use to communicate across machines.

本章中描述的模块列表是：

17.1 subprocess —子进程管理

2.4 新版功能.

subprocess 模块允许你生成新的进程，连接它们的输入、输出、错误管道，并且获取它们的返回码。此模块打算代替一些老旧的模块与功能：

```
os.system
os.spawn*
os.popen*
popen2.*
commands.*
```

Information about how this module can be used to replace the older functions can be found in the *subprocess-replacements* section.

参见：

POSIX users (Linux, BSD, etc.) are strongly encouraged to install and use the much more recent subprocess32 module instead of the version included with python 2.7. It is a drop in replacement with better behavior in many situations.

PEP 324 –提出 subprocess 模块的 PEP

17.1.1 使用 subprocess 模块

The recommended way to launch subprocesses is to use the following convenience functions. For more advanced use cases when these do not meet your needs, use the underlying *Popen* interface.

subprocess.**call**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*)
Run the command described by *args*. Wait for command to complete, then return the returncode attribute.

The arguments shown above are merely the most common ones, described below in 常用参数 (hence the slightly odd notation in the abbreviated signature). The full function signature is the same as that of the *Popen* constructor - this functions passes all supplied arguments directly through to that interface.

示例:

```
>>> subprocess.call(["ls", "-l"])
0

>>> subprocess.call("exit 1", shell=True)
1
```

警告: Using shell=True can be a security hazard. See the warning under 常用参数 for details.

注解: Do not use stdout=PIPE or stderr=PIPE with this function as that can deadlock based on the child process output volume. Use *Popen* with the communicate() method when you need pipes.

subprocess.**check_call**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*)
Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise *CalledProcessError*. The *CalledProcessError* object will have the return code in the *returncode* attribute.

The arguments shown above are merely the most common ones, described below in 常用参数 (hence the slightly odd notation in the abbreviated signature). The full function signature is the same as that of the *Popen* constructor - this functions passes all supplied arguments directly through to that interface.

示例:

```
>>> subprocess.check_call(["ls", "-l"])
0

>>> subprocess.check_call("exit 1", shell=True)
Traceback (most recent call last):
   ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1
```

2.5 新版功能.

警告: Using shell=True can be a security hazard. See the warning under 常用参数 for details.

注解: Do not use stdout=PIPE or stderr=PIPE with this function as that can deadlock based on the child process output volume. Use *Popen* with the communicate() method when you need pipes.

subprocess.**check_output**(*args*, *, *stdin=None*, *stderr=None*, *shell=False*, *universal_newlines=False*)

Run command with arguments and return its output as a byte string.

If the return code was non-zero it raises a *CalledProcessError*. The *CalledProcessError* object will have the return code in the *returncode* attribute and any output in the *output* attribute.

The arguments shown above are merely the most common ones, described below in 常用参数 (hence the slightly odd notation in the abbreviated signature). The full function signature is largely the same as that of the *Popen* constructor, except that *stdout* is not permitted as it is used internally. All other supplied arguments are passed directly through to the *Popen* constructor.

示例:

```
>>> subprocess.check_output(["echo", "Hello World!"])
'Hello World!\n'

>>> subprocess.check_output("exit 1", shell=True)
Traceback (most recent call last):
   ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1
```

To also capture standard error in the result, use `stderr=subprocess.STDOUT`:

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

2.7 新版功能.

警告: Using `shell=True` can be a security hazard. See the warning under 常用参数 for details.

注解: Do not use `stderr=PIPE` with this function as that can deadlock based on the child process error volume. Use *Popen* with the `communicate()` method when you need a stderr pipe.

subprocess.**PIPE**

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to *Popen* and indicates that a pipe to the standard stream should be opened.

subprocess.**STDOUT**

可被*Popen* 的 *stdin* ， *stdout* 或者 *stderr* 参数使用的特殊值，表示标准错误与标准输出使用同一句柄。

exception subprocess.**CalledProcessError**

Exception raised when a process run by *check_call()* or *check_output()* returns a non-zero exit status.

returncode

Exit status of the child process.

cmd

用于创建子进程的指令。

output

Output of the child process if this exception is raised by *check_output()*. Otherwise, `None`.

常用参数

为了支持丰富的使用案例，*Popen* 的构造函数（以及方便的函数）接受大量可选的参数。对于大多数典型的用例，许多参数可以被安全地留以它们的默认值。通常需要的参数有：

args 被所有调用需要，应当为一个字符串，或者一个程序参数序列。提供一个参数序列通常更好，它可以更小心地使用参数中的转义字符以及引用（例如允许文件名中的空格）。如果传递一个简单的字符串，则 *shell* 参数必须为*True*（见下文）或者该字符串中将被运行的程序名必须用简单的命名而不指定任何参数。

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are *PIPE*, an existing file descriptor (a positive integer), an existing file object, and None. *PIPE* indicates that a new pipe to the child should be created. With the default settings of None, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, *stderr* can be *STDOUT*, which indicates that the stderr data from the child process should be captured into the same file handle as for stdout.

When *stdout* or *stderr* are pipes and *universal_newlines* is True then all line endings will be converted to '\n' as described for the *universal newlines* 'U' mode argument to *open()*.

如果 *shell* 设为 True,，则使用 shell 执行指定的指令。如果您主要使用 Python 增强的控制流（它比大多数系统 shell 提供的强大），并且仍然希望方便地使用其他 shell 功能，如 shell 管道、文件通配符、环境变量展开以及 ~ 展开到用户家目录，这将非常有用。但是，注意 Python 自己也实现了许多类似 shell 的特性（例如*glob*, *fnmatch*, *os.walk()*, *os.path.expandvars()*, *os.path.expanduser()* 和*shutil*）。

警告： Executing shell commands that incorporate unsanitized input from an untrusted source makes a program vulnerable to shell injection, a serious security flaw which can result in arbitrary command execution. For this reason, the use of shell=True is **strongly discouraged** in cases where the command string is constructed from external input:

```
>>> from subprocess import call
>>> filename = input("What file would you like to display?\n")
What file would you like to display?
non_existent; rm -rf / #
>>> call("cat " + filename, shell=True) # Uh-oh. This will end badly...
```

shell=False disables all shell based features, but does not suffer from this vulnerability; see the Note in the *Popen* constructor documentation for helpful hints in getting shell=False to work.

When using shell=True, *pipes.quote()* can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

这些选项以及所有其他选项在*Popen* 构造函数文档中有更详细的描述。

Popen 构造函数

此模块的底层的进程创建与管理由*Popen* 类处理。它提供了很大的灵活性，因此开发者能够处理未被便利函数覆盖的不常见用例。

class subprocess.**Popen**(*args*, *bufsize=0*, *executable=None*, *stdin=None*, *stdout=None*, *stderr=None*, *preexec_fn=None*, *close_fds=False*, *shell=False*, *cwd=None*, *env=None*, *universal_newlines=False*, *startupinfo=None*, *creationflags=0*)

Execute a child program in a new process. On Unix, the class uses *os.execvp()*-like behavior to execute the child program. On Windows, the class uses the Windows CreateProcess() function. The arguments to *Popen* are as follows.

args 应当是一个程序的参数列表或者一个简单的字符串。默认情况下，如果 *args* 是一个序列，将运行的程序是此序列的第一项。如果 *args* 是一个字符串，解释是平台相关的，如下所述。有关默认行为的其他差异，见 *shell* 和 *executable* 参数。除非另有说明，推荐将 *args* 作为序列传递。

On Unix, if *args* is a string, the string is interpreted as the name or path of the program to execute. However, this can only be done if not passing arguments to the program.

注解： *shlex.split()* can be useful when determining the correct tokenization for *args*, especially in complex cases:

```
>>> import shlex, subprocess
>>> command_line = raw_input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
>>> args = shlex.split(command_line)
>>> print args
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd', "echo '
↪$MONEY'"]
>>> p = subprocess.Popen(args) # Success!
```

特别注意，由 shell 中的空格分隔的选项（例如 *-input*）和参数（例如 *eggs.txt* ）位于分开的列表元素中，而在需要时使用引号或反斜杠转义的参数在 shell （例如包含空格的文件名或上面显示的 *echo* 命令）是单独的列表元素。

在 Windows，如果 *args* 是一个序列，他将通过一个在*Converting an argument sequence to a string on Windows* 描述的方式被转换为一个字符串。这是因为底层的 `CreateProcess()` 只处理字符串。

参数 *shell* （默认为 `False`）指定是否使用 shell 执行程序。如果 *shell* 为 `True`，更推荐将 *args* 作为字符串传递而非序列。

On Unix with `shell=True`, the shell defaults to `/bin/sh`. If *args* is a string, the string specifies the command to execute through the shell. This means that the string must be formatted exactly as it would be when typed at the shell prompt. This includes, for example, quoting or backslash escaping filenames with spaces in them. If *args* is a sequence, the first item specifies the command string, and any additional items will be treated as additional arguments to the shell itself. That is to say, *Popen* does the equivalent of:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

在 Windows，使用 `shell=True`，环境变量 `COMSPEC` 指定了默认 shell。在 Windows 你唯一需要指定 `shell=True` 的情况是你想要执行内置在 shell 中的命令（例如 **`dir`** 或者 **`copy`**）。在运行一个批处理文件或者基于控制台的可执行文件时，不需要 `shell=True`。

> **警告：** Passing `shell=True` can be a security hazard if combined with untrusted input. See the warning under 常用参数 for details.

bufsize, if given, has the same meaning as the corresponding argument to the built-in open() function: `0` means unbuffered, `1` means line buffered, any other positive value means use a buffer of (approximately) that size. A negative *bufsize* means to use the system default, which usually means fully buffered. The default value for *bufsize* is `0` (unbuffered).

注解： If you experience performance issues, it is recommended that you try to enable buffering by setting *bufsize* to either -1 or a large enough positive value (such as 4096).

The *executable* argument specifies a replacement program to execute. It is very seldom needed. When `shell=False`, *executable* replaces the program to execute specified by *args*. However, the original *args* is still passed to the program. Most programs treat the program specified by *args* as the command name, which can then be different from the program actually executed. On Unix, the *args* name becomes the display name for the executable in utilities such as **ps**. If `shell=True`, on Unix the *executable* argument specifies a replacement shell for the default `/bin/sh`.

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are *PIPE*, an existing file descriptor (a positive integer), an existing file object, and `None`. *PIPE* indicates that a new pipe to the child should be created. With the default settings of `None`, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, *stderr* can be *STDOUT*, which indicates that the stderr data from the child process should be captured into the same file handle as for stdout.

If *preexec_fn* is set to a callable object, this object will be called in the child process just before the child is executed. (Unix only)

If *close_fds* is true, all file descriptors except `0`, `1` and `2` will be closed before the child process is executed. (Unix only). Or, on Windows, if *close_fds* is true then no handles will be inherited by the child process. Note that on Windows, you cannot set *close_fds* to true and also redirect the standard handles by setting *stdin*, *stdout* or *stderr*.

If *cwd* is not `None`, the child's current directory will be changed to *cwd* before it is executed. Note that this directory is not considered when searching the executable, so you can't specify the program's path relative to *cwd*.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of inheriting the current process' environment, which is the default behavior.

注解: 如果指定，*env* 必须提供所有被子进程需求的变量。在 Windows，为了运行一个 side-by-side assembly，指定的 *env* **必须**包含一个有效的 `SystemRoot`。

If *universal_newlines* is `True`, the file objects *stdout* and *stderr* are opened as text files in *universal newlines* mode. Lines may be terminated by any of `'\n'`, the Unix end-of-line convention, `'\r'`, the old Macintosh convention or `'\r\n'`, the Windows convention. All of these external representations are seen as `'\n'` by the Python program.

注解: This feature is only available if Python is built with universal newline support (the default). Also, the newlines attribute of the file objects *stdout*, *stdin* and *stderr* are not updated by the communicate() method.

If given, *startupinfo* will be a *STARTUPINFO* object, which is passed to the underlying `CreateProcess` function. *creationflags*, if given, can be *CREATE_NEW_CONSOLE* or *CREATE_NEW_PROCESS_GROUP*. (Windows only)

异常

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent. Additionally, the exception object will have one extra attribute called `child_traceback`, which is a string containing traceback information from the child's point of view.

最常见的被抛出异常是*OSError*。例如，当尝试执行一个不存在的文件时就会发生。应用程序需要为*OSError*异常做好准备。

如果*Popen*调用时有无效的参数，则一个*ValueError*将被抛出。

`check_all()` 与*check_output()*在调用的进程返回非零退出码时将抛出*CalledProcessError*。

Security

Unlike some other popen functions, this implementation will never call a system shell implicitly. This means that all characters, including shell metacharacters, can safely be passed to child processes. Obviously, if the shell is invoked explicitly, then it is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately.

17.1.2 Popen 对象

Popen 类的实例拥有以下方法：

Popen.**poll**()
Check if child process has terminated. Set and return *returncode* attribute.

Popen.**wait**()
等待子进程被终止。设置并返回*returncode* 属性。

> **警告:** This will deadlock when using `stdout=PIPE` and/or `stderr=PIPE` and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use *communicate()* to avoid that.

Popen.**communicate**(*input=None*)
Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate. The optional *input* argument should be a string to be sent to the child process, or `None`, if no data should be sent to the child.

communicate() returns a tuple `(stdoutdata, stderrdata)`.

注意如果你想要向进程的 stdin 传输数据，你需要通过 `stdin=PIPE` 创建此 Popen 对象。类似的，要从结果元组获取任何非 `None` 值，你同样需要设置 `stdout=PIPE` 或者 `stderr=PIPE`。

> **注解:** 内存里数据读取是缓冲的，所以如果数据尺寸过大或无限，不要使用此方法。

Popen.**send_signal**(*signal*)
将信号 *signal* 发送给子进程。

> **注解:** 在 Windows，SIGTERM 是一个*terminate()* 的别名。CTRL_C_EVENT 和 CTRL_BREAK_EVENT 可以被发送给以包含 `CREATE_NEW_PROCESS` 的 *creationflags* 形参启动的进程。

2.6 新版功能.

Popen.**terminate**()
Stop the child. On Posix OSs the method sends SIGTERM to the child. On Windows the Win32 API function `TerminateProcess()` is called to stop the child.

2.6 新版功能.

Popen.**kill**()
Kills the child. On Posix OSs the function sends SIGKILL to the child. On Windows *kill()* is an alias for *terminate()*.

2.6 新版功能.

以下属性也是可用的：

警告： 使用*communicate()* 而非*.stdin.write*， *.stdout.read* 或者*.stderr.read* 来避免由于任意其他 OS 管道缓冲区被子进程填满阻塞而导致的死锁。

Popen.**stdin**
If the *stdin* argument was *PIPE*, this attribute is a file object that provides input to the child process. Otherwise, it is None.

Popen.**stdout**
If the *stdout* argument was *PIPE*, this attribute is a file object that provides output from the child process. Otherwise, it is None.

Popen.**stderr**
If the *stderr* argument was *PIPE*, this attribute is a file object that provides error output from the child process. Otherwise, it is None.

Popen.**pid**
子进程的进程号。

注意如果你设置了 *shell* 参数为 True，则这是生成的子 shell 的进程号。

Popen.**returncode**
此进程的退出码，由*poll()* 和*wait()* 设置（以及直接由*communicate()* 设置）。一个 None 值表示此进程仍未结束。

A negative value −N indicates that the child was terminated by signal N (Unix only).

17.1.3 Windows Popen 助手

STARTUPINFO 类和以下常数仅在 Windows 有效。

class subprocess.**STARTUPINFO**
Partial support of the Windows STARTUPINFO structure is used for *Popen* creation.

dwFlags
一个位字段，用于确定进程在创建窗口时是否使用某些*STARTUPINFO* 属性。

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW
```

hStdInput
如果*dwFlags* 被指定为*STARTF_USESTDHANDLES*，则此属性是进程的标准输入句柄，如果*STARTF_USESTDHANDLES* 未指定，则默认的标准输入是键盘缓冲区。

hStdOutput
如果*dwFlags* 被指定为*STARTF_USESTDHANDLES*，则此属性是进程的标准输出句柄。除此之外，此此属性将被忽略并且默认标准输出是控制台窗口缓冲区。

hStdError
如果*dwFlags* 被指定为*STARTF_USESTDHANDLES*，则此属性是进程的标准错误句柄。除此之外，此属性将被忽略并且默认标准错误为控制台窗口的缓冲区。

wShowWindow
如果*dwFlags* 指定了*STARTF_USESHOWWINDOW*，此属性可为能被指定为函数 ShowWindow 的 nCmdShow 的形参的任意值，除了 SW_SHOWDEFAULT。如此之外，此属性被忽略。

SW_HIDE 被提供给此属性。它在*Popen* 由 shell=True 调用时使用。

Constants

subprocess 模块曝出以下常数。

subprocess.**STD_INPUT_HANDLE**
标准输入设备，这是控制台输入缓冲区 CONIN$。

subprocess.**STD_OUTPUT_HANDLE**
标准输出设备。最初，这是活动控制台屏幕缓冲区 CONOUT$。

subprocess.**STD_ERROR_HANDLE**
标准错误设备。最初，这是活动控制台屏幕缓冲区 CONOUT$。

subprocess.**SW_HIDE**
隐藏窗口。另一个窗口将被激活。

subprocess.**STARTF_USESTDHANDLES**
Specifies that the *STARTUPINFO.hStdInput*, *STARTUPINFO.hStdOutput*, and *STARTUPINFO.hStdError* attributes contain additional information.

subprocess.**STARTF_USESHOWWINDOW**
Specifies that the *STARTUPINFO.wShowWindow* attribute contains additional information.

subprocess.**CREATE_NEW_CONSOLE**
The new process has a new console, instead of inheriting its parent's console (the default).

This flag is always set when *Popen* is created with `shell=True`.

subprocess.**CREATE_NEW_PROCESS_GROUP**
A *Popen* `creationflags` parameter to specify that a new process group will be created. This flag is necessary for using *os.kill()* on the subprocess.

This flag is ignored if *CREATE_NEW_CONSOLE* is specified.

17.1.4 Replacing Older Functions with the `subprocess` Module

In this section, "a becomes b" means that b can be used as a replacement for a.

注解： All "a" functions in this section fail (more or less) silently if the executed program cannot be found; the "b" replacements raise *OSError* instead.

In addition, the replacements using *check_output()* will fail with a *CalledProcessError* if the requested operation produces a non-zero return code. The output is still available as the *output* attribute of the raised exception.

In the following examples, we assume that the relevant functions have already been imported from the *subprocess* module.

Replacing /bin/sh shell backquote

```
output=`mycmd myarg`
```

becomes:

```
output = check_output(["mycmd", "myarg"])
```

Replacing shell pipeline

```
output=`dmesg | grep hda`
```

becomes:

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

The p1.stdout.close() call after starting the p2 is important in order for p1 to receive a SIGPIPE if p2 exits before p1.

Alternatively, for trusted input, the shell's own pipeline support may still be used directly:

```
output=`dmesg | grep hda`
```

becomes:

```
output=check_output("dmesg | grep hda", shell=True)
```

Replacing `os.system()`

```
status = os.system("mycmd" + " myarg")
# becomes
status = subprocess.call("mycmd" + " myarg", shell=True)
```

注释:

- Calling the program through the shell is usually not required.

A more realistic example would look like this:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print >>sys.stderr, "Child was terminated by signal", -retcode
    else:
        print >>sys.stderr, "Child returned", retcode
except OSError as e:
    print >>sys.stderr, "Execution failed:", e
```

Replacing the os.spawn family

P_NOWAIT example:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT example:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

Vector example:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

Environment example:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

Replacing os.popen(), os.popen2(), os.popen3()

```
pipe = os.popen("cmd", 'r', bufsize)
==>
pipe = Popen("cmd", shell=True, bufsize=bufsize, stdout=PIPE).stdout
```

```
pipe = os.popen("cmd", 'w', bufsize)
==>
pipe = Popen("cmd", shell=True, bufsize=bufsize, stdin=PIPE).stdin
```

```
(child_stdin, child_stdout) = os.popen2("cmd", mode, bufsize)
==>
p = Popen("cmd", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin,
 child_stdout,
 child_stderr) = os.popen3("cmd", mode, bufsize)
==>
p = Popen("cmd", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4("cmd", mode,
                                                   bufsize)
```

(下页继续)

(续上页)

```
==>
p = Popen("cmd", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

On Unix, os.popen2, os.popen3 and os.popen4 also accept a sequence as the command to execute, in which case arguments will be passed directly to the program without shell intervention. This usage can be replaced as follows:

```
(child_stdin, child_stdout) = os.popen2(["/bin/ls", "-l"], mode,
                                        bufsize)
==>
p = Popen(["/bin/ls", "-l"], bufsize=bufsize, stdin=PIPE, stdout=PIPE)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

Return code handling translates as follows:

```
pipe = os.popen("cmd", 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print "There were some errors"
==>
process = Popen("cmd", shell=True, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print "There were some errors"
```

Replacing functions from the popen2 module

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)
==>
p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

On Unix, popen2 also accepts a sequence as the command to execute, in which case arguments will be passed directly to the program without shell intervention. This usage can be replaced as follows:

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize,
                                            mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

popen2.Popen3 and *popen2.Popen4* basically work as *subprocess.Popen*, except that:

- *Popen* raises an exception if the execution fails.
- the *capturestderr* argument is replaced with the *stderr* argument.
- `stdin=PIPE` and `stdout=PIPE` must be specified.
- popen2 closes all file descriptors by default, but you have to specify `close_fds=True` with *Popen*.

17.1.5 注释

Converting an argument sequence to a string on Windows

On Windows, an *args* sequence is converted to a string that can be parsed using the following rules (which correspond to the rules used by the MS C runtime):

1. Arguments are delimited by white space, which is either a space or a tab.
2. A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
3. A double quotation mark preceded by a backslash is interpreted as a literal double quotation mark.
4. Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
5. If backslashes immediately precede a double quotation mark, every pair of backslashes is interpreted as a literal backslash. If the number of backslashes is odd, the last backslash escapes the next double quotation mark as described in rule 3.

17.2 socket —底层网络接口

This module provides access to the BSD *socket* interface. It is available on all modern Unix systems, Windows, Mac OS X, BeOS, OS/2, and probably additional platforms.

注解: 一些行为可能因平台不同而异，因为调用的是操作系统的套接字 API。

For an introduction to socket programming (in C), see the following papers: An Introductory 4.3BSD Interprocess Communication Tutorial, by Stuart Sechrest and An Advanced 4.3BSD Interprocess Communication Tutorial, by Samuel J. Leffler et al, both in the UNIX Programmer's Manual, Supplementary Documents 1 (sections PS1:7 and PS1:8). The platform-specific reference material for the various socket-related system calls are also a valuable source of information on the details of socket semantics. For Unix, refer to the manual pages; for Windows, see the WinSock (or Winsock 2) specification. For IPv6-ready APIs, readers may want to refer to **RFC 3493** titled Basic Socket Interface Extensions for IPv6.

这个 Python 接口是用 Python 的面向对象风格对 Unix 系统调用和套接字库接口的直译：函数*socket()* 返回一个 套接字对象，其方法是对各种套接字系统调用的实现。形参类型一般与 C 接口相比更高级：例如在 Python 文件 `read()` 和 `write()` 操作中，接收操作的缓冲区分配是自动的，发送操作的缓冲区长度是隐式的。

Socket addresses are represented as follows: A single string is used for the *AF_UNIX* address family. A pair `(host, port)` is used for the *AF_INET* address family, where *host* is a string representing either a hostname in Internet domain notation like `'daring.cwi.nl'` or an IPv4 address like `'100.50.200.5'`, and *port* is an integer. For *AF_INET6* address family, a four-tuple `(host, port, flowinfo, scopeid)` is used, where *flowinfo* and *scopeid* represents `sin6_flowinfo` and `sin6_scope_id` member in `struct sockaddr_in6` in C. For *socket* module methods, *flowinfo* and *scopeid* can be omitted just for backward compatibility. Note, however, omission of *scopeid* can cause problems in manipulating scoped IPv6 addresses. Other address families are currently not supported. The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created.

For IPv4 addresses, two special forms are accepted instead of a host address: the empty string represents `INADDR_ANY`, and the string `'<broadcast>'` represents `INADDR_BROADCAST`. The behavior is not available for IPv6 for backward compatibility, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.

如果你在 IPv4/v6 套接字地址的 *host* 部分中使用了一个主机名，此程序可能会表现不确定行为，因为 Python 使用 DNS 解析返回的第一个地址。套接字地址在实际的 IPv4/v6 中以不同方式解析，根据 DNS 解析和/或 host 配置。为了确定行为，在 *host* 部分中使用数字地址。

2.5 新版功能: AF_NETLINK sockets are represented as pairs `pid, groups`.

2.6 新版功能: Linux-only support for TIPC is also available using the `AF_TIPC` address family. TIPC is an open, non-IP based networked protocol designed for use in clustered computer environments. Addresses are represented by a tuple, and the fields depend on the address type. The general tuple form is `(addr_type, v1, v2, v3 [, scope])`, where:

- *addr_type* 取 `TIPC_ADDR_NAMESEQ`、`TIPC_ADDR_NAME` 或 `TIPC_ADDR_ID` 中的一个。
- *scope* 取 `TIPC_ZONE_SCOPE`、`TIPC_CLUSTER_SCOPE` 和 `TIPC_NODE_SCOPE` 中的一个。
- 如果 *addr_type* 为 `TIPC_ADDR_NAME`，那么 *v1* 是服务器类型，*v2* 是端口标识符，*v3* 应为 0。

  如果 *addr_type* 为 `TIPC_ADDR_NAMESEQ`，那么 *v1* 是服务器类型，*v2* 是端口号下限，而 *v3* 是端口号上限。

  如果 *addr_type* 为 `TIPC_ADDR_ID`，那么 *v1* 是节点 (node)，*v2* 是 ref，*v3* 应为 0。

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; errors related to socket or address semantics raise the error *socket.error*.

可以用*setblocking()* 设置非阻塞模式。一个基于超时的 generalization 通过*settimeout()* 支持。

The module *socket* exports the following constants and functions:

exception socket.**error**
: This exception is raised for socket-related errors. The accompanying value is either a string telling what went wrong or a pair `(errno, string)` representing an error returned by a system call, similar to the value accompanying *os.error*. See the module *errno*, which contains names for the error codes defined by the underlying operating system.

  在 2.6 版更改: *socket.error* is now a child class of *IOError*.

exception socket.**herror**
: This exception is raised for address-related errors, i.e. for functions that use *h_errno* in the C API, including *gethostbyname_ex()* and *gethostbyaddr()*.

  The accompanying value is a pair `(h_errno, string)` representing an error returned by a library call. *string* represents the description of *h_errno*, as returned by the `hstrerror()` C function.

exception socket.**gaierror**
: This exception is raised for address-related errors, for *getaddrinfo()* and *getnameinfo()*. The accompanying value is a pair `(error, string)` representing an error returned by a library call. *string* represents the description of *error*, as returned by the `gai_strerror()` C function. The *error* value will match one of the `EAI_*` constants defined in this module.

exception socket.**timeout**
: This exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to `settimeout()`. The accompanying value is a string whose value is currently always "timed out".

  2.3 新版功能.

socket.**AF_UNIX**
socket.**AF_INET**
socket.**AF_INET6**
: These constants represent the address (and protocol) families, used for the first argument to *socket()*. If the *AF_UNIX* constant is not defined then this protocol is unsupported.

socket.**SOCK_STREAM**

socket.**SOCK_DGRAM**
socket.**SOCK_RAW**
socket.**SOCK_RDM**
socket.**SOCK_SEQPACKET**
These constants represent the socket types, used for the second argument to *socket()*. (Only *SOCK_STREAM* and *SOCK_DGRAM* appear to be generally useful.)

SO_*
socket.**SOMAXCONN**
MSG_*
SOL_*
IPPROTO_*
IPPORT_*
INADDR_*
IP_*
IPV6_*
EAI_*
AI_*
NI_*
TCP_*
此列表内的许多常量，记载在 Unix 文档中的套接字和/或 IP 协议部分，同时也定义在本 socket 模块中。它们通常用于套接字对象的 `setsockopt()` 和 `getsockopt()` 方法的参数中。在大多数情况下，仅那些在 Unix 头文件中有定义的符号会在本模块中定义，部分符号提供了默认值。

SIO_*
RCVALL_*
Windows 的 WSAIoctl() 的常量。这些常量用于套接字对象的*ioctl()* 方法的参数。

2.6 新版功能.

TIPC_*
TIPC 相关常量，与 C socket API 导出的常量一致。更多信息请参阅 TIPC 文档。

2.6 新版功能.

socket.**has_ipv6**
本常量为一个布尔值，该值指示当前平台是否支持 IPv6。

2.3 新版功能.

socket.**create_connection**(*address*[, *timeout*[, *source_address*]])
连接到一个 TCP 服务，该服务正在侦听 Internet *address*（用二元组 `(host, port)` 表示）。连接后返回套接字对象。这是比*socket.connect()* 更高级的函数：如果 *host* 是非数字主机名，它将尝试从*AF_INET* 和*AF_INET6* 解析它，然后依次尝试连接到所有可能的地址，直到连接成功。这使得编写兼容 IPv4 和 IPv6 的客户端变得容易。

传入可选参数 *timeout* 可以在套接字实例上设置超时（在尝试连接前）。如果未提供 *timeout*，则使用由*getdefaulttimeout()* 返回的全局默认超时设置。

如果提供了 *source_address*，它必须为二元组 `(host, port)`，以便套接字在连接之前绑定为其源地址。如果 host 或 port 分别为 '' 或 0，则使用操作系统默认行为。

2.6 新版功能.

在 2.7 版更改: 添加了 *source_address* 参数

socket.**getaddrinfo**(*host*, *port*[, *family*[, *socktype*[, *proto*[, *flags*]]]])
将 *host*/*port* 参数转换为 5 元组的序列，其中包含创建（连接到某服务的）套接字所需的所有参数。*host* 是域名，是字符串格式的 IPv4/v6 地址或 `None`。*port* 是字符串格式的服务名称，如 `'http'`、端口号（数字）或 `None`。传入 `None` 作为 *host* 和 *port* 的值，相当于将 `NULL` 传递给底层 C API。

The *family*, *socktype* and *proto* arguments can be optionally specified in order to narrow the list of addresses returned. By default, their value is `0`, meaning that the full range of results is selected. The *flags* argument can be one or several of the `AI_*` constants, and will influence how results are computed and returned. Its default value is `0`. For example, `AI_NUMERICHOST` will disable domain name resolution and will raise an error if *host* is a domain name.

本函数返回的 5 元组列表具有以下结构：

```
(family, socktype, proto, canonname, sockaddr)
```

In these tuples, *family*, *socktype*, *proto* are all integers and are meant to be passed to the *socket()* function. *canonname* will be a string representing the canonical name of the *host* if `AI_CANONNAME` is part of the *flags* argument; else *canonname* will be empty. *sockaddr* is a tuple describing a socket address, whose format depends on the returned *family* (a `(address, port)` 2-tuple for *AF_INET*, a `(address, port, flow info, scope id)` 4-tuple for *AF_INET6*), and is meant to be passed to the *socket.connect()* method.

下面的示例获取了 TCP 连接地址信息，假设该连接通过 80 端口连接至 `example.org` （如果系统未启用 IPv6，则结果可能会不同）：

```
>>> socket.getaddrinfo("example.org", 80, 0, 0, socket.IPPROTO_TCP)
[(10, 1, 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (2, 1, 6, '', ('93.184.216.34', 80))]
```

2.2 新版功能.

`socket.`**`getfqdn`**`(`[*name*]`)`

返回 *name* 的全限定域名 (Fully Qualified Domain Name –FQDN)。如果 *name* 省略或为空，则将其解释为本地主机。为了查找全限定名称，首先将检查由*gethostbyaddr()* 返回的主机名，然后是主机的别名（如果存在）。选中第一个包含句点的名字。如果无法获取全限定域名，则返回由*gethostname()* 返回的主机名。

2.0 新版功能.

`socket.`**`gethostbyname`**`(`*hostname*`)`

将主机名转换为 IPv4 地址格式。IPv4 地址以字符串格式返回，如 `'100.50.200.5'`。如果主机名本身是 IPv4 地址，则原样返回。更完整的接口请参考*gethostbyname_ex()*。*gethostbyname()* 不支持 IPv6 域名解析，应使用*getaddrinfo()* 来支持 IPv4/v6 双协议栈。

`socket.`**`gethostbyname_ex`**`(`*hostname*`)`

将主机名转换为 IPv4 地址格式的扩展接口。返回三元组 `(hostname, aliaslist, ipaddrlist)`，其中 *hostname* 是响应给定 *ip_address* 的主要主机名，*aliaslist* 是相同地址的其他可用主机名的列表（可能为空），而 *ipaddrlist* 是 IPv4 地址列表，包含相同主机名、相同接口的不同地址（通常是一个地址，但不总是如此）。*gethostbyname_ex()* 不支持 IPv6 名称解析，应使用*getaddrinfo()* 来支持 IPv4/v6 双协议栈。

`socket.`**`gethostname`**`()`

返回一个字符串，包含当前正在运行 Python 解释器的机器的主机名。

If you want to know the current machine's IP address, you may want to use `gethostbyname(gethostname())`. This operation assumes that there is a valid address-to-host mapping for the host, and the assumption does not always hold.

Note: *gethostname()* doesn't always return the fully qualified domain name; use `getfqdn()` (see above).

`socket.`**`gethostbyaddr`**`(`*ip_address*`)`

返回三元组 `(hostname, aliaslist, ipaddrlist)`，其中 *hostname* 是响应给定 *ip_address* 的主要主机名，*aliaslist* 是相同地址的其他可用主机名的列表（可能为空），而 *ipaddrlist* 是 IPv4/v6 地址列表，包含相同主机名、相同接口的不同地址（很可能仅包含一个地址）。要查询全限定域名，请使用函数*getfqdn()*。*gethostbyaddr()* 支持 IPv4 和 IPv6。

socket.**getnameinfo**(*sockaddr*, *flags*)
将套接字地址 *sockaddr* 转换为二元组 (host, port)。*host* 的形式可能是全限定域名，或是由数字表示的地址，具体取决于 *flags* 的设置。同样，*port* 可以包含字符串格式的端口名称或数字格式的端口号。

2.2 新版功能.

socket.**getprotobyname**(*protocolname*)
将 Internet 协议名称（如 'icmp'）转换为常量，该常量适用于*socket()* 函数的第三个（可选）参数。通常只有在“raw”模式 (*SOCK_RAW*) 中打开的套接字才需要使用该常量。对于正常的套接字模式，协议省略或为零时，会自动选择正确的协议。

socket.**getservbyname**(*servicename*[, *protocolname*])
将 Internet 服务名称和协议名称转换为该服务的端口号。协议名称是可选的，如果提供的话应为 'tcp' 或 'udp'，否则将匹配出所有协议。

socket.**getservbyport**(*port*[, *protocolname*])
将 Internet 端口号和协议名称转换为该服务的服务名称。协议名称是可选的，如果提供的话应为 'tcp' 或 'udp'，否则将匹配出所有协议。

socket.**socket**([*family*[, *type*[, *proto*]]])
Create a new socket using the given address family, socket type and protocol number. The address family should be *AF_INET* (the default), *AF_INET6* or *AF_UNIX*. The socket type should be *SOCK_STREAM* (the default), *SOCK_DGRAM* or perhaps one of the other SOCK_ constants. The protocol number is usually zero and may be omitted in that case.

socket.**socketpair**([*family*[, *type*[, *proto*]]])
Build a pair of connected socket objects using the given address family, socket type, and protocol number. Address family, socket type, and protocol number are as for the *socket()* function above. The default family is *AF_UNIX* if defined on the platform; otherwise, the default is *AF_INET*. Availability: Unix.

2.4 新版功能.

socket.**fromfd**(*fd*, *family*, *type*[, *proto*])
Duplicate the file descriptor *fd* (an integer as returned by a file object’s fileno() method) and build a socket object from the result. Address family, socket type and protocol number are as for the *socket()* function above. The file descriptor should refer to a socket, but this is not checked —subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the Unix inet daemon). The socket is assumed to be in blocking mode. Availability: Unix.

socket.**ntohl**(*x*)
将 32 位正整数从网络字节序转换为主机字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 4 字节交换操作。

socket.**ntohs**(*x*)
将 16 位正整数从网络字节序转换为主机字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 2 字节交换操作。

socket.**htonl**(*x*)
将 32 位正整数从主机字节序转换为网络字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 4 字节交换操作。

socket.**htons**(*x*)
将 16 位正整数从主机字节序转换为网络字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 2 字节交换操作。

socket.**inet_aton**(*ip_string*)
Convert an IPv4 address from dotted-quad string format (for example, ‘123.45.67.89’) to 32-bit packed binary format, as a string four characters in length. This is useful when conversing with a program that uses the standard

C library and needs objects of type `struct in_addr`, which is the C type for the 32-bit packed binary this function returns.

inet_aton() 也接受句点数少于三的字符串，详情请参阅 Unix 手册 *inet(3)*。

If the IPv4 address string passed to this function is invalid, *socket.error* will be raised. Note that exactly what is valid depends on the underlying C implementation of `inet_aton()`.

inet_aton() 不支持 IPv6，在 IPv4/v6 双协议栈下应使用*inet_pton()* 来代替。

`socket.`**`inet_ntoa`**(*packed_ip*)
Convert a 32-bit packed IPv4 address (a string four characters in length) to its standard dotted-quad string representation (for example, '123.45.67.89'). This is useful when conversing with a program that uses the standard C library and needs objects of type `struct in_addr`, which is the C type for the 32-bit packed binary data this function takes as an argument.

If the string passed to this function is not exactly 4 bytes in length, *socket.error* will be raised. *inet_ntoa()* does not support IPv6, and *inet_ntop()* should be used instead for IPv4/v6 dual stack support.

`socket.`**`inet_pton`**(*address_family*, *ip_string*)
将特定地址簇的 IP 地址（字符串）转换为压缩二进制格式。当库或网络协议需要接受 `struct in_addr` 类型的对象（类似*inet_aton()*）或 `struct in6_addr` 类型的对象时，*inet_pton()* 很有用。

Supported values for *address_family* are currently *AF_INET* and *AF_INET6*. If the IP address string *ip_string* is invalid, *socket.error* will be raised. Note that exactly what is valid depends on both the value of *address_family* and the underlying implementation of `inet_pton()`.

Availability: Unix (maybe not all platforms).

2.3 新版功能.

`socket.`**`inet_ntop`**(*address_family*, *packed_ip*)
Convert a packed IP address (a string of some number of characters) to its standard, family-specific string representation (for example, `'7.10.0.5'` or `'5aef:2b::8'`) *inet_ntop()* is useful when a library or network protocol returns an object of type `struct in_addr` (similar to *inet_ntoa()*) or `struct in6_addr`.

Supported values for *address_family* are currently *AF_INET* and *AF_INET6*. If the string *packed_ip* is not the correct length for the specified address family, *ValueError* will be raised. A *socket.error* is raised for errors from the call to *inet_ntop()*.

Availability: Unix (maybe not all platforms).

2.3 新版功能.

`socket.`**`getdefaulttimeout`**()
返回用于新套接字对象的默认超时（以秒为单位的浮点数）。值 `None` 表示新套接字对象没有超时。首次导入 socket 模块时，默认值为 `None`。

2.3 新版功能.

`socket.`**`setdefaulttimeout`**(*timeout*)
Set the default timeout in seconds (float) for new socket objects. A value of `None` indicates that new socket objects have no timeout. When the socket module is first imported, the default is `None`.

2.3 新版功能.

`socket.`**`SocketType`**
这是一个 Python 类型对象，表示套接字对象的类型。它等同于 `type(socket(...))`。

参见:

Module ***SocketServer*** 用于简化网络服务端编写的类。

模块*ssl* 套接字对象的 TLS/SSL 封装。

17.2.1 套接字对象

Socket objects have the following methods. Except for `makefile()` these correspond to Unix system calls applicable to sockets.

`socket.`**`accept`**`()`
接受一个连接。此 scoket 必须绑定到一个地址上并且监听连接。返回值是一个 `(conn, address)` 对，其中 *conn* 是一个 新的套接字对象，用于在此连接上收发数据，*address* 是连接另一端的套接字所绑定的地址。

`socket.`**`bind`**`(`*address*`)`
将套接字绑定到 *address*。套接字必须尚未绑定。(*address* 的格式取决于地址簇——参见上文)

注解: This method has historically accepted a pair of parameters for *AF_INET* addresses instead of only a tuple. This was never intentional and is no longer available in Python 2.0 and later.

`socket.`**`close`**`()`
Close the socket. All future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

注解: *close()* 释放与连接相关联的资源，但不一定立即关闭连接。如果需要及时关闭连接，请在调用*close()* 之前调用*shutdown()*。

`socket.`**`connect`**`(`*address*`)`
连接到 *address* 处的远程套接字。(*address* 的格式取决于地址簇——参见上文)

注解: This method has historically accepted a pair of parameters for *AF_INET* addresses instead of only a tuple. This was never intentional and is no longer available in Python 2.0 and later.

`socket.`**`connect_ex`**`(`*address*`)`
类似于 `connect(address)`，但是对于 C 级别的 `connect()` 调用返回的错误，本函数将返回错误指示器，而不是抛出异常 (对于其他问题，如 “找不到主机”，仍然可以抛出异常)。如果操作成功，则错误指示器为 `0`，否则为 `errno` 变量的值。这对支持如异步连接很有用。

注解: This method has historically accepted a pair of parameters for *AF_INET* addresses instead of only a tuple. This was never intentional and is no longer available in Python 2.0 and later.

`socket.`**`fileno`**`()`
Return the socket’s file descriptor (a small integer). This is useful with *select.select()*.

在 Windows 下，此方法返回的小整数在允许使用文件描述符的地方无法使用 (如*os.fdopen()*)。Unix 无此限制。

`socket.`**`getpeername`**`()`
返回套接字连接到的远程地址。举例而言，这可以用于查找远程 IPv4/v6 套接字的端口号。(返回的地址格式取决于地址簇——参见上文。) 部分系统不支持此函数。

socket.**getsockname**()
返回套接字本身的地址。举例而言，这可以用于查找 IPv4/v6 套接字的端口号。(返回的地址格式取决于地址簇——参见上文。)

socket.**getsockopt**(*level*, *optname*[, *buflen*])
Return the value of the given socket option (see the Unix man page *getsockopt(2)*). The needed symbolic constants (SO_* etc.) are defined in this module. If *buflen* is absent, an integer option is assumed and its integer value is returned by the function. If *buflen* is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a string. It is up to the caller to decode the contents of the buffer (see the optional built-in module *struct* for a way to decode C structures encoded as strings).

socket.**ioctl**(*control*, *option*)

Platform Windows

The *ioctl()* method is a limited interface to the WSAIoctl system interface. Please refer to the Win32 documentation for more information.

On other platforms, the generic *fcntl.fcntl()* and *fcntl.ioctl()* functions may be used; they accept a socket object as their first argument.

2.6 新版功能.

socket.**listen**(*backlog*)
Listen for connections made to the socket. The *backlog* argument specifies the maximum number of queued connections and should be at least 0; the maximum value is system-dependent (usually 5), the minimum value is forced to 0.

socket.**makefile**([*mode*[, *bufsize*]])
Return a *file object* associated with the socket. (File objects are described in *File Objects*.) The file object does not close the socket explicitly when its *close()* method is called, but only removes its reference to the socket object, so that the socket will be closed if it is not referenced from anywhere else.

The socket must be in blocking mode (it can not have a timeout). The optional *mode* and *bufsize* arguments are interpreted the same way as by the built-in *file()* function.

注解: On Windows, the file-like object created by *makefile()* cannot be used where a file object with a file descriptor is expected, such as the stream arguments of *subprocess.Popen()*.

---

socket.**recv**(*bufsize*[, *flags*])
Receive data from the socket. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by *bufsize*. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero.

注解: For best match with hardware and network realities, the value of *bufsize* should be a relatively small power of 2, for example, 4096.

---

socket.**recvfrom**(*bufsize*[, *flags*])
Receive data from the socket. The return value is a pair (string, address) where *string* is a string representing the data received and *address* is the address of the socket sending the data. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family —see above.)

socket.**recvfrom_into**(*buffer*[, *nbytes*[, *flags*]])
Receive data from the socket, writing it into *buffer* instead of creating a new string. The return value is a pair (nbytes, address) where *nbytes* is the number of bytes received and *address* is the address of the socket

sending the data. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family —see above.)

2.5 新版功能.

socket.**recv_into**(*buffer*[, *nbytes*[, *flags*]])
Receive up to *nbytes* bytes from the socket, storing the data into a buffer rather than creating a new string. If *nbytes* is not specified (or 0), receive up to the size available in the given buffer. Returns the number of bytes received. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero.

2.5 新版功能.

socket.**send**(*string*[, *flags*])
Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for *recv()* above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. For further information on this concept, consult the socket-howto.

socket.**sendall**(*string*[, *flags*])
Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for *recv()* above. Unlike *send()*, this method continues to send data from *string* until either all data has been sent or an error occurs. `None` is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

socket.**sendto**(*string*, *address*)
socket.**sendto**(*string*, *flags*, *address*)
Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*. The optional *flags* argument has the same meaning as for *recv()* above. Return the number of bytes sent. (The format of *address* depends on the address family —see above.)

socket.**setblocking**(*flag*)
Set blocking or non-blocking mode of the socket: if *flag* is 0, the socket is set to non-blocking, else to blocking mode. Initially all sockets are in blocking mode. In non-blocking mode, if a *recv()* call doesn't find any data, or if a *send()* call can't immediately dispose of the data, an *error* exception is raised; in blocking mode, the calls block until they can proceed. `s.setblocking(0)` is equivalent to `s.settimeout(0.0)`; `s.setblocking(1)` is equivalent to `s.settimeout(None)`.

socket.**settimeout**(*value*)
Set a timeout on blocking socket operations. The *value* argument can be a nonnegative float expressing seconds, or `None`. If a float is given, subsequent socket operations will raise a *timeout* exception if the timeout period *value* has elapsed before the operation has completed. Setting a timeout of `None` disables timeouts on socket operations. `s.settimeout(0.0)` is equivalent to `s.setblocking(0)`; `s.settimeout(None)` is equivalent to `s.setblocking(1)`.

2.3 新版功能.

socket.**gettimeout**()
Return the timeout in seconds (float) associated with socket operations, or `None` if no timeout is set. This reflects the last call to *setblocking()* or *settimeout()*.

2.3 新版功能.

Some notes on socket blocking and timeouts: A socket object can be in one of three modes: blocking, non-blocking, or timeout. Sockets are always created in blocking mode. In blocking mode, operations block until complete or the system returns an error (such as connection timed out). In non-blocking mode, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately. In timeout mode, operations fail if they cannot be completed within the timeout specified for the socket or if the system returns an error. The *setblocking()* method is simply a shorthand for certain *settimeout()* calls.

Timeout mode internally sets the socket in non-blocking mode. The blocking and timeout modes are shared between file

descriptors and socket objects that refer to the same network endpoint. A consequence of this is that file objects returned by the *makefile()* method must only be used when the socket is in blocking mode; in timeout or non-blocking mode file operations that cannot be completed immediately will fail.

Note that the *connect()* operation is subject to the timeout setting, and in general it is recommended to call *settimeout()* before calling *connect()* or pass a timeout parameter to *create_connection()*. The system network stack may return a connection timeout error of its own regardless of any Python socket timeout setting.

socket.**setsockopt**(*level*, *optname*, *value*)
Set the value of the given socket option (see the Unix manual page *setsockopt(2)*). The needed symbolic constants are defined in the *socket* module (SO_* etc.). The value can be an integer or a string representing a buffer. In the latter case it is up to the caller to ensure that the string contains the proper bits (see the optional built-in module *struct* for a way to encode C structures as strings).

socket.**shutdown**(*how*)
Shut down one or both halves of the connection. If *how* is SHUT_RD, further receives are disallowed. If *how* is SHUT_WR, further sends are disallowed. If *how* is SHUT_RDWR, further sends and receives are disallowed. Depending on the platform, shutting down one half of the connection can also close the opposite half (e.g. on Mac OS X, shutdown(SHUT_WR) does not allow further reads on the other end of the connection).

Note that there are no methods read() or write(); use *recv()* and *send()* without *flags* argument instead.

Socket objects also have these (read-only) attributes that correspond to the values given to the *socket* constructor.

socket.**family**
The socket family.

2.5 新版功能.

socket.**type**
The socket type.

2.5 新版功能.

socket.**proto**
The socket protocol.

2.5 新版功能.

17.2.2 示例

Here are four minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence *socket()*, *bind()*, *listen()*, *accept()* (possibly repeating the *accept()* to service more than one client), while a client only needs the sequence *socket()*, *connect()*. Also note that the server does not *sendall()*/*recv()* on the socket it is listening on but on the new socket returned by *accept()*.

The first two examples support IPv4 only.

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
```

(下页继续)

(续上页)

```
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.sendall(data)
conn.close()
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.sendall('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)
```

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to the all addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

```
# Echo server program
import socket
import sys

HOST = None               # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except socket.error as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print 'could not open socket'
    sys.exit(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()
```

```
# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except socket.error as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print 'could not open socket'
    sys.exit(1)
s.sendall('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)
```

The last example shows how to write a very simple network sniffer with raw sockets on Windows. The example requires administrator privileges to modify the interface:

```
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print s.recvfrom(65565)

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

Running an example several times with too small delay between executions, could lead to this error:

```
socket.error: [Errno 98] Address already in use
```

This is because the previous execution has left the socket in a `TIME_WAIT` state, and can't be immediately reused.

There is a *socket* flag to set, in order to prevent this, socket.SO_REUSEADDR:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

the SO_REUSEADDR flag tells the kernel to reuse a local socket in TIME_WAIT state, without waiting for its natural timeout to expire.

17.3 ssl —套接字对象的 TLS/SSL 封装

2.6 新版功能.

源代码: Lib/ssl.py

This module provides access to Transport Layer Security (often known as "Secure Sockets Layer") encryption and peer authentication facilities for network sockets, both client-side and server-side. This module uses the OpenSSL library. It is available on all modern Unix systems, Windows, Mac OS X, and probably additional platforms, as long as OpenSSL is installed on that platform.

在 2.7.13 版更改: Updated to support linking with OpenSSL 1.1.0

注解: Some behavior may be platform dependent, since calls are made to the operating system socket APIs. The installed version of OpenSSL may also cause variations in behavior. For example, TLSv1.1 and TLSv1.2 come with openssl version 1.0.1.

警告: Don't use this module without reading the *Security considerations*. Doing so may lead to a false sense of security, as the default settings of the ssl module are not necessarily appropriate for your application.

This section documents the objects and functions in the ssl module; for more general information about TLS, SSL, and certificates, the reader is referred to the documents in the "See Also" section at the bottom.

This module provides a class, ssl.SSLSocket, which is derived from the *socket.socket* type, and provides a socket-like wrapper that also encrypts and decrypts the data going over the socket with SSL. It supports additional methods such as getpeercert(), which retrieves the certificate of the other side of the connection, and cipher(),which retrieves the cipher being used for the secure connection.

For more sophisticated applications, the *ssl.SSLContext* class helps manage settings and certificates, which can then be inherited by SSL sockets created through the *SSLContext.wrap_socket()* method.

17.3.1 Functions, Constants, and Exceptions

exception ssl.**SSLError**

Raised to signal an error from the underlying SSL implementation (currently provided by the OpenSSL library). This signifies some problem in the higher-level encryption and authentication layer that's superimposed on the underlying network connection. This error is a subtype of *socket.error*, which in turn is a subtype of *IOError*. The error code and message of *SSLError* instances are provided by the OpenSSL library.

library
A string mnemonic designating the OpenSSL submodule in which the error occurred, such as `SSL`, `PEM` or `X509`. The range of possible values depends on the OpenSSL version.

2.7.9 新版功能.

reason
A string mnemonic designating the reason this error occurred, for example `CERTIFICATE_VERIFY_FAILED`. The range of possible values depends on the OpenSSL version.

2.7.9 新版功能.

exception ssl.**SSLZeroReturnError**
A subclass of *SSLError* raised when trying to read or write and the SSL connection has been closed cleanly. Note that this doesn't mean that the underlying transport (read TCP) has been closed.

2.7.9 新版功能.

exception ssl.**SSLWantReadError**
A subclass of *SSLError* raised by a *non-blocking SSL socket* when trying to read or write data, but more data needs to be received on the underlying TCP transport before the request can be fulfilled.

2.7.9 新版功能.

exception ssl.**SSLWantWriteError**
A subclass of *SSLError* raised by a *non-blocking SSL socket* when trying to read or write data, but more data needs to be sent on the underlying TCP transport before the request can be fulfilled.

2.7.9 新版功能.

exception ssl.**SSLSyscallError**
A subclass of *SSLError* raised when a system error was encountered while trying to fulfill an operation on a SSL socket. Unfortunately, there is no easy way to inspect the original errno number.

2.7.9 新版功能.

exception ssl.**SSLEOFError**
A subclass of *SSLError* raised when the SSL connection has been terminated abruptly. Generally, you shouldn't try to reuse the underlying transport when this error is encountered.

2.7.9 新版功能.

exception ssl.**CertificateError**
Raised to signal an error with a certificate (such as mismatching hostname). Certificate errors detected by OpenSSL, though, raise an *SSLError*.

Socket creation

The following function allows for standalone socket creation. Starting from Python 2.7.9, it can be more flexible to use *SSLContext.wrap_socket()* instead.

ssl.**wrap_socket**(*sock*, *keyfile=None*, *certfile=None*, *server_side=False*, *cert_reqs=CERT_NONE*, *ssl_version={see docs}*, *ca_certs=None*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *ciphers=None*)
Takes an instance `sock` of *socket.socket*, and returns an instance of `ssl.SSLSocket`, a subtype of *socket.socket*, which wraps the underlying socket in an SSL context. `sock` must be a *SOCK_STREAM* socket; other socket types are unsupported.

For client-side sockets, the context construction is lazy; if the underlying socket isn't connected yet, the context construction will be performed after `connect()` is called on the socket. For server-side sockets, if the socket has

no remote peer, it is assumed to be a listening socket, and the server-side SSL wrapping is automatically performed on client connections accepted via the `accept()` method. *`wrap_socket()`* may raise *`SSLError`*.

The `keyfile` and `certfile` parameters specify optional files which contain a certificate to be used to identify the local side of the connection. See the discussion of *Certificates* for more information on how the certificate is stored in the `certfile`.

The parameter `server_side` is a boolean which identifies whether server-side or client-side behavior is desired from this socket.

The parameter `cert_reqs` specifies whether a certificate is required from the other side of the connection, and whether it will be validated if provided. It must be one of the three values *`CERT_NONE`* (certificates ignored), *`CERT_OPTIONAL`* (not required, but validated if provided), or *`CERT_REQUIRED`* (required and validated). If the value of this parameter is not *`CERT_NONE`*, then the `ca_certs` parameter must point to a file of CA certificates.

The `ca_certs` file contains a set of concatenated "certification authority" certificates, which are used to validate certificates passed from the other end of the connection. See the discussion of *Certificates* for more information about how to arrange the certificates in this file.

The parameter `ssl_version` specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server's choice. Most of the versions are not interoperable with the other versions. If not specified, the default is *`PROTOCOL_SSLv23`*; it provides the most compatibility with other versions.

Here's a table showing which versions in a client (down the side) can connect to which versions in a server (along the top):

client / **server**	**SSLv2**	**SSLv3**	**SSLv23**	**TLSv1**	**TLSv1.1**	**TLSv1.2**
SSLv2	是	否	是	否	否	否
SSLv3	否	是	是	否	否	否
SSLv23[1]	否	是	是	是	是	是
TLSv1	否	否	是	是	否	否
TLSv1.1	否	否	是	否	是	否
TLSv1.2	否	否	是	否	否	是

备注

注解： Which connections succeed will vary depending on the version of OpenSSL. For example, before OpenSSL 1.0.0, an SSLv23 client would always attempt SSLv2 connections.

The *ciphers* parameter sets the available ciphers for this SSL object. It should be a string in the OpenSSL cipher list format.

The parameter `do_handshake_on_connect` specifies whether to do the SSL handshake automatically after doing a `socket.connect()`, or whether the application program will call it explicitly, by invoking the *`SSLSocket.do_handshake()`* method. Calling *`SSLSocket.do_handshake()`* explicitly gives the program control over the blocking behavior of the socket I/O involved in the handshake.

The parameter `suppress_ragged_eofs` specifies how the `SSLSocket.read()` method should signal unexpected EOF from the other end of the connection. If specified as *`True`* (the default), it returns a normal EOF (an empty bytes object) in response to unexpected EOF errors raised from the underlying socket; if *`False`*, it will raise the exceptions back to the caller.

[1] TLS 1.3 protocol will be available with *`PROTOCOL_SSLv23`* in OpenSSL >= 1.1.1. There is no dedicated PROTOCOL constant for just TLS 1.3.

在 2.7 版更改: New optional argument *ciphers*.

上下文创建

A convenience function helps create `SSLContext` objects for common purposes.

ssl.**create_default_context**(*purpose=Purpose.SERVER_AUTH*, *cafile=None*, *capath=None*, *cadata=None*)

Return a new `SSLContext` object with default settings for the given *purpose*. The settings are chosen by the `ssl` module, and usually represent a higher security level than when calling the `SSLContext` constructor directly.

cafile, *capath*, *cadata* represent optional CA certificates to trust for certificate verification, as in `SSLContext.load_verify_locations()`. If all three are `None`, this function can choose to trust the system's default CA certificates instead.

The settings are: `PROTOCOL_SSLv23`, `OP_NO_SSLv2`, and `OP_NO_SSLv3` with high encryption cipher suites without RC4 and without unauthenticated cipher suites. Passing `SERVER_AUTH` as *purpose* sets `verify_mode` to `CERT_REQUIRED` and either loads CA certificates (when at least one of *cafile*, *capath* or *cadata* is given) or uses `SSLContext.load_default_certs()` to load default CA certificates.

注解: The protocol, options, cipher and other settings may change to more restrictive values anytime without prior deprecation. The values represent a fair balance between compatibility and security.

If your application needs specific settings, you should create a `SSLContext` and apply the settings yourself.

注解: If you find that when certain older clients or servers attempt to connect with a `SSLContext` created by this function that they get an error stating "Protocol or cipher suite mismatch", it may be that they only support SSL3.0 which this function excludes using the `OP_NO_SSLv3`. SSL3.0 is widely considered to be completely broken. If you still wish to continue to use this function but still allow SSL 3.0 connections you can re-enable them using:

```
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options &= ~ssl.OP_NO_SSLv3
```

2.7.9 新版功能.

在 2.7.10 版更改: RC4 was dropped from the default cipher string.

在 2.7.13 版更改: ChaCha20/Poly1305 was added to the default cipher string.

3DES was dropped from the default cipher string.

ssl.**_https_verify_certificates**(*enable=True*)

Specifies whether or not server certificates are verified when creating client HTTPS connections without specifying a particular SSL context.

Starting with Python 2.7.9, `httplib` and modules which use it, such as `urllib2` and `xmlrpclib`, default to verifying remote server certificates received when establishing client HTTPS connections. This default verification checks that the certificate is signed by a Certificate Authority in the system trust store and that the Common Name (or Subject Alternate Name) on the presented certificate matches the requested host.

Setting *enable* to `True` ensures this default behaviour is in effect.

Setting *enable* to `False` reverts the default HTTPS certificate handling to that of Python 2.7.8 and earlier, allowing connections to servers using self-signed certificates, servers using certificates signed by a Certicate Authority not present in the system trust store, and servers where the hostname does not match the presented server certificate.

The leading underscore on this function denotes that it intentionally does not exist in any implementation of Python 3 and may not be present in all Python 2.7 implementations. The portable approach to bypassing certificate checks or the system trust store when necessary is for tools to enable that on a case-by-case basis by explicitly passing in a suitably configured SSL context, rather than reverting the default behaviour of the standard library client modules.

2.7.12 新版功能.

参见:

- CVE-2014-9365 –HTTPS man-in-the-middle attack against Python clients using default settings
- **PEP 476** –Enabling certificate verification by default for HTTPS
- **PEP 493** –HTTPS verification migration tools for Python 2.7

Random generation

2.7.13 版后已移除: OpenSSL has deprecated `ssl.RAND_pseudo_bytes()`, use `ssl.RAND_bytes()` instead.

`ssl.`**`RAND_status`**`()`
Return `True` if the SSL pseudo-random number generator has been seeded with 'enough' randomness, and `False` otherwise. You can use *`ssl.RAND_egd()`* and *`ssl.RAND_add()`* to increase the randomness of the pseudo-random number generator.

`ssl.`**`RAND_egd`**`(`*path*`)`
If you are running an entropy-gathering daemon (EGD) somewhere, and *path* is the pathname of a socket connection open to it, this will read 256 bytes of randomness from the socket, and add it to the SSL pseudo-random number generator to increase the security of generated secret keys. This is typically only necessary on systems without better sources of randomness.

See http://egd.sourceforge.net/ or http://prngd.sourceforge.net/ for sources of entropy-gathering daemons.

Availability: not available with LibreSSL and OpenSSL > 1.1.0

`ssl.`**`RAND_add`**`(`*bytes*, *entropy*`)`
Mix the given *bytes* into the SSL pseudo-random number generator. The parameter *entropy* (a float) is a lower bound on the entropy contained in string (so you can always use `0.0`). See **RFC 1750** for more information on sources of entropy.

Certificate handling

`ssl.`**`match_hostname`**`(`*cert*, *hostname*`)`
Verify that *cert* (in decoded format as returned by *`SSLSocket.getpeercert()`*) matches the given *hostname*. The rules applied are those for checking the identity of HTTPS servers as outlined in **RFC 2818** and **RFC 6125**, except that IP addresses are not currently supported. In addition to HTTPS, this function should be suitable for checking the identity of servers in various SSL-based protocols such as FTPS, IMAPS, POPS and others.

`CertificateError` is raised on failure. On success, the function returns nothing:

```
>>> cert = {'subject': ((('commonName', 'example.com'),),)}
>>> ssl.match_hostname(cert, "example.com")
>>> ssl.match_hostname(cert, "example.org")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/py3k/Lib/ssl.py", line 130, in match_hostname
ssl.CertificateError: hostname 'example.org' doesn't match 'example.com'
```

2.7.9 新版功能.

ssl.**cert_time_to_seconds**(*cert_time*)
Return the time in seconds since the Epoch, given the `cert_time` string representing the "notBefore" or "notAfter" date from a certificate in `"%b %d %H:%M:%S %Y %Z"` strptime format (C locale).

Here's an example:

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

"notBefore" or "notAfter" dates must use GMT (**RFC 5280**).

在 2.7.9 版更改: Interpret the input time as a time in UTC as specified by 'GMT' timezone in the input string. Local timezone was used previously. Return an integer (no fractions of a second in the input format)

ssl.**get_server_certificate**(*addr*, *ssl_version=PROTOCOL_SSLv23*, *ca_certs=None*)
Given the address `addr` of an SSL-protected server, as a (*hostname*, *port-number*) pair, fetches the server's certificate, and returns it as a PEM-encoded string. If `ssl_version` is specified, uses that version of the SSL protocol to attempt to connect to the server. If `ca_certs` is specified, it should be a file containing a list of root certificates, the same format as used for the same parameter in *`wrap_socket()`*. The call will attempt to validate the server certificate against that set of root certificates, and will fail if the validation attempt fails.

在 2.7.9 版更改: This function is now IPv6-compatible, and the default *ssl_version* is changed from *`PROTOCOL_SSLv3`* to *`PROTOCOL_SSLv23`* for maximum compatibility with modern servers.

ssl.**DER_cert_to_PEM_cert**(*DER_cert_bytes*)
Given a certificate as a DER-encoded blob of bytes, returns a PEM-encoded string version of the same certificate.

ssl.**PEM_cert_to_DER_cert**(*PEM_cert_string*)
Given a certificate as an ASCII PEM string, returns a DER-encoded sequence of bytes for that same certificate.

ssl.**get_default_verify_paths**()
Returns a named tuple with paths to OpenSSL's default cafile and capath. The paths are the same as used by *`SSLContext.set_default_verify_paths()`*. The return value is a *named tuple* `DefaultVerifyPaths`:

- `cafile` - resolved path to cafile or `None` if the file doesn't exist,
- `capath` - resolved path to capath or `None` if the directory doesn't exist,
- `openssl_cafile_env` - OpenSSL's environment key that points to a cafile,
- `openssl_cafile` - hard coded path to a cafile,
- `openssl_capath_env` - OpenSSL's environment key that points to a capath,
- `openssl_capath` - hard coded path to a capath directory

Availability: LibreSSL ignores the environment vars `openssl_cafile_env` and `openssl_capath_env`

2.7.9 新版功能.

ssl.**enum_certificates**(*store_name*)
Retrieve certificates from Windows' system cert store. *store_name* may be one of `CA`, `ROOT` or `MY`. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either `x509_asn` for X.509 ASN.1 data or `pkcs_7_asn` for PKCS#7 ASN.1 data. Trust

specifies the purpose of the certificate as a set of OIDS or exactly `True` if the certificate is trustworthy for all purposes.

示例:

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

Availability: Windows.

2.7.9 新版功能.

ssl.**enum_crls**(*store_name*)
Retrieve CRLs from Windows' system cert store. *store_name* may be one of `CA`, `ROOT` or `MY`. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either `x509_asn` for X.509 ASN.1 data or `pkcs_7_asn` for PKCS#7 ASN.1 data.

Availability: Windows.

2.7.9 新版功能.

常量

ssl.**CERT_NONE**
Possible value for *SSLContext.verify_mode*, or the `cert_reqs` parameter to *wrap_socket()*. In this mode (the default), no certificates will be required from the other side of the socket connection. If a certificate is received from the other end, no attempt to validate it is made.

See the discussion of *Security considerations* below.

ssl.**CERT_OPTIONAL**
Possible value for *SSLContext.verify_mode*, or the `cert_reqs` parameter to *wrap_socket()*. In this mode no certificates will be required from the other side of the socket connection; but if they are provided, validation will be attempted and an *SSLError* will be raised on failure.

Use of this setting requires a valid set of CA certificates to be passed, either to *SSLContext.load_verify_locations()* or as a value of the `ca_certs` parameter to *wrap_socket()*.

ssl.**CERT_REQUIRED**
Possible value for *SSLContext.verify_mode*, or the `cert_reqs` parameter to *wrap_socket()*. In this mode, certificates are required from the other side of the socket connection; an *SSLError* will be raised if no certificate is provided, or if its validation fails.

Use of this setting requires a valid set of CA certificates to be passed, either to *SSLContext.load_verify_locations()* or as a value of the `ca_certs` parameter to *wrap_socket()*.

ssl.**VERIFY_DEFAULT**
Possible value for *SSLContext.verify_flags*. In this mode, certificate revocation lists (CRLs) are not checked. By default OpenSSL does neither require nor verify CRLs.

2.7.9 新版功能.

ssl.**VERIFY_CRL_CHECK_LEAF**
Possible value for *SSLContext.verify_flags*. In this mode, only the peer cert is check but non of the intermediate CA certificates. The mode requires a valid CRL that is signed by the peer cert's issuer (its direct ancestor CA). If no proper has been loaded *SSLContext.load_verify_locations*, validation will fail.

2.7.9 新版功能.

ssl.**VERIFY_CRL_CHECK_CHAIN**
Possible value for *SSLContext.verify_flags*. In this mode, CRLs of all certificates in the peer cert chain are checked.

2.7.9 新版功能.

ssl.**VERIFY_X509_STRICT**
Possible value for *SSLContext.verify_flags* to disable workarounds for broken X.509 certificates.

2.7.9 新版功能.

ssl.**VERIFY_X509_TRUSTED_FIRST**
Possible value for *SSLContext.verify_flags*. It instructs OpenSSL to prefer trusted certificates when building the trust chain to validate a certificate. This flag is enabled by default.

2.7.10 新版功能.

ssl.**PROTOCOL_TLS**
Selects the highest protocol version that both the client and server support. Despite the name, this option can select "TLS" protocols as well as "SSL".

2.7.13 新版功能.

ssl.**PROTOCOL_SSLv23**
Alias for PROTOCOL_TLS.

2.7.13 版后已移除: Use PROTOCOL_TLS instead.

ssl.**PROTOCOL_SSLv2**
Selects SSL version 2 as the channel encryption protocol.

This protocol is not available if OpenSSL is compiled with the OPENSSL_NO_SSL2 flag.

> **警告:** SSL version 2 is insecure. Its use is highly discouraged.

2.7.13 版后已移除: OpenSSL has removed support for SSLv2.

ssl.**PROTOCOL_SSLv3**
Selects SSL version 3 as the channel encryption protocol.

This protocol is not be available if OpenSSL is compiled with the OPENSSL_NO_SSLv3 flag.

> **警告:** SSL version 3 is insecure. Its use is highly discouraged.

2.7.13 版后已移除: OpenSSL has deprecated all version specific protocols. Use the default protocol with flags like OP_NO_SSLv3 instead.

ssl.**PROTOCOL_TLSv1**
Selects TLS version 1.0 as the channel encryption protocol.

2.7.13 版后已移除: OpenSSL has deprecated all version specific protocols. Use the default protocol with flags like OP_NO_SSLv3 instead.

ssl.**PROTOCOL_TLSv1_1**
Selects TLS version 1.1 as the channel encryption protocol. Available only with openssl version 1.0.1+.

2.7.9 新版功能.

2.7.13 版后已移除: OpenSSL has deprecated all version specific protocols. Use the default protocol with flags like OP_NO_SSLv3 instead.

ssl.**PROTOCOL_TLSv1_2**
Selects TLS version 1.2 as the channel encryption protocol. This is the most modern version, and probably the best choice for maximum protection, if both sides can speak it. Available only with openssl version 1.0.1+.

2.7.9 新版功能.

2.7.13 版后已移除: OpenSSL has deprecated all version specific protocols. Use the default protocol with flags like `OP_NO_SSLv3` instead.

ssl.**OP_ALL**
Enables workarounds for various bugs present in other SSL implementations. This option is set by default. It does not necessarily set the same flags as OpenSSL's `SSL_OP_ALL` constant.

2.7.9 新版功能.

ssl.**OP_NO_SSLv2**
Prevents an SSLv2 connection. This option is only applicable in conjunction with *PROTOCOL_SSLv23*. It prevents the peers from choosing SSLv2 as the protocol version.

2.7.9 新版功能.

ssl.**OP_NO_SSLv3**
Prevents an SSLv3 connection. This option is only applicable in conjunction with *PROTOCOL_SSLv23*. It prevents the peers from choosing SSLv3 as the protocol version.

2.7.9 新版功能.

ssl.**OP_NO_TLSv1**
Prevents a TLSv1 connection. This option is only applicable in conjunction with *PROTOCOL_SSLv23*. It prevents the peers from choosing TLSv1 as the protocol version.

2.7.9 新版功能.

ssl.**OP_NO_TLSv1_1**
Prevents a TLSv1.1 connection. This option is only applicable in conjunction with *PROTOCOL_SSLv23*. It prevents the peers from choosing TLSv1.1 as the protocol version. Available only with openssl version 1.0.1+.

2.7.9 新版功能.

ssl.**OP_NO_TLSv1_2**
Prevents a TLSv1.2 connection. This option is only applicable in conjunction with *PROTOCOL_SSLv23*. It prevents the peers from choosing TLSv1.2 as the protocol version. Available only with openssl version 1.0.1+.

2.7.9 新版功能.

ssl.**OP_NO_TLSv1_3**
Prevents a TLSv1.3 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1.3 as the protocol version. TLS 1.3 is available with OpenSSL 1.1.1 or later. When Python has been compiled against an older version of OpenSSL, the flag defaults to *0*.

2.7.15 新版功能.

ssl.**OP_CIPHER_SERVER_PREFERENCE**
Use the server's cipher ordering preference, rather than the client's. This option has no effect on client sockets and SSLv2 server sockets.

2.7.9 新版功能.

ssl.**OP_SINGLE_DH_USE**
Prevents re-use of the same DH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

2.7.9 新版功能.

ssl.**OP_SINGLE_ECDH_USE**
Prevents re-use of the same ECDH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

2.7.9 新版功能.

ssl.**OP_ENABLE_MIDDLEBOX_COMPAT**
Send dummy Change Cipher Spec (CCS) messages in TLS 1.3 handshake to make a TLS 1.3 connection look more like a TLS 1.2 connection.

This option is only available with OpenSSL 1.1.1 and later.

2.7.16 新版功能.

ssl.**OP_NO_COMPRESSION**
Disable compression on the SSL channel. This is useful if the application protocol supports its own compression scheme.

This option is only available with OpenSSL 1.0.0 and later.

2.7.9 新版功能.

ssl.**HAS_ALPN**
Whether the OpenSSL library has built-in support for the *Application-Layer Protocol Negotiation* TLS extension as described in **RFC 7301**.

2.7.10 新版功能.

ssl.**HAS_ECDH**
Whether the OpenSSL library has built-in support for Elliptic Curve-based Diffie-Hellman key exchange. This should be true unless the feature was explicitly disabled by the distributor.

2.7.9 新版功能.

ssl.**HAS_SNI**
Whether the OpenSSL library has built-in support for the *Server Name Indication* extension (as defined in **RFC 4366**).

2.7.9 新版功能.

ssl.**HAS_NPN**
Whether the OpenSSL library has built-in support for *Next Protocol Negotiation* as described in the NPN draft specification. When true, you can use the *SSLContext.set_npn_protocols()* method to advertise which protocols you want to support.

2.7.9 新版功能.

ssl.**HAS_TLSv1_3**
Whether the OpenSSL library has built-in support for the TLS 1.3 protocol.

2.7.15 新版功能.

ssl.**CHANNEL_BINDING_TYPES**
List of supported TLS channel binding types. Strings in this list can be used as arguments to *SSLSocket.get_channel_binding()*.

2.7.9 新版功能.

ssl.**OPENSSL_VERSION**
The version string of the OpenSSL library loaded by the interpreter:

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 0.9.8k 25 Mar 2009'
```

2.7 新版功能.

ssl.**OPENSSL_VERSION_INFO**
A tuple of five integers representing version information about the OpenSSL library:

```
>>> ssl.OPENSSL_VERSION_INFO
(0, 9, 8, 11, 15)
```

2.7 新版功能.

ssl.**OPENSSL_VERSION_NUMBER**
The raw version number of the OpenSSL library, as a single integer:

```
>>> ssl.OPENSSL_VERSION_NUMBER
9470143L
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x9080bfL'
```

2.7 新版功能.

ssl.**ALERT_DESCRIPTION_HANDSHAKE_FAILURE**
ssl.**ALERT_DESCRIPTION_INTERNAL_ERROR**
ALERT_DESCRIPTION_*
Alert Descriptions from **RFC 5246** and others. The IANA TLS Alert Registry contains this list and references to the RFCs where their meaning is defined.

Used as the return value of the callback function in *SSLContext.set_servername_callback()*.

2.7.9 新版功能.

Purpose.**SERVER_AUTH**
Option for *create_default_context()* and *SSLContext.load_default_certs()*. This value indicates that the context may be used to authenticate Web servers (therefore, it will be used to create client-side sockets).

2.7.9 新版功能.

Purpose.**CLIENT_AUTH**
Option for *create_default_context()* and *SSLContext.load_default_certs()*. This value indicates that the context may be used to authenticate Web clients (therefore, it will be used to create server-side sockets).

2.7.9 新版功能.

17.3.2 SSL Sockets

SSL sockets provide the following methods of 套接字对象:

- *accept()*
- *bind()*
- *close()*
- *connect()*
- *fileno()*
- *getpeername()*, *getsockname()*
- *getsockopt()*, *setsockopt()*

- *gettimeout()*, *settimeout()*, *setblocking()*
- *listen()*
- *makefile()*
- *recv()*, *recv_into()* (but passing a non-zero `flags` argument is not allowed)
- *send()*, *sendall()* (with the same limitation)
- *shutdown()*

However, since the SSL (and TLS) protocol has its own framing atop of TCP, the SSL sockets abstraction can, in certain respects, diverge from the specification of normal, OS-level sockets. See especially the *notes on non-blocking sockets*.

SSL sockets also have the following additional methods and attributes:

`SSLSocket.`**`do_handshake`**`()`
Perform the SSL setup handshake.

在 2.7.9 版更改: The handshake method also performs *match_hostname()* when the *check_hostname* attribute of the socket's *context* is true.

`SSLSocket.`**`getpeercert`**`(`*binary_form=False*`)`
If there is no certificate for the peer on the other end of the connection, return `None`. If the SSL handshake hasn't been done yet, raise *ValueError*.

If the `binary_form` parameter is *False*, and a certificate was received from the peer, this method returns a *dict* instance. If the certificate was not validated, the dict is empty. If the certificate was validated, it returns a dict with several keys, amongst them `subject` (the principal for which the certificate was issued) and `issuer` (the principal issuing the certificate). If a certificate contains an instance of the *Subject Alternative Name* extension (see **RFC 3280**), there will also be a `subjectAltName` key in the dictionary.

The `subject` and `issuer` fields are tuples containing the sequence of relative distinguished names (RDNs) given in the certificate's data structure for the respective fields, and each RDN is a sequence of name-value pairs. Here is a real-world example:

```
{'issuer': ((('countryName', 'IL'),),
            (('organizationName', 'StartCom Ltd.'),),
            (('organizationalUnitName',
              'Secure Digital Certificate Signing'),),
            (('commonName',
              'StartCom Class 2 Primary Intermediate Server CA'),)),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': ((('description', '571208-SLe257oHY9fVQ07Z'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
             (('commonName', '*.eff.org'),),
             (('emailAddress', 'hostmaster@eff.org'),)),
 'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
 'version': 3}
```

注解: To validate a certificate for a particular service, you can use the *match_hostname()* function.

If the `binary_form` parameter is *True*, and a certificate was provided, this method returns the DER-encoded form of the entire certificate as a sequence of bytes, or *None* if the peer did not provide a certificate. Whether the

peer provides a certificate depends on the SSL socket's role:

- for a client SSL socket, the server will always provide a certificate, regardless of whether validation was required;
- for a server SSL socket, the client will only provide a certificate when requested by the server; therefore *getpeercert()* will return *None* if you used *CERT_NONE* (rather than *CERT_OPTIONAL* or *CERT_REQUIRED*).

在 2.7.9 版更改: The returned dictionary includes additional items such as `issuer` and `notBefore`. Additionall *ValueError* is raised when the handshake isn't done. The returned dictionary includes additional X509v3 extension items such as `crlDistributionPoints`, `caIssuers` and `OCSP` URIs.

`SSLSocket.`**`cipher`**`()`
Returns a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used. If no connection has been established, returns `None`.

`SSLSocket.`**`compression`**`()`
Return the compression algorithm being used as a string, or `None` if the connection isn't compressed.

If the higher-level protocol supports its own compression mechanism, you can use *OP_NO_COMPRESSION* to disable SSL-level compression.

2.7.9 新版功能.

`SSLSocket.`**`get_channel_binding`**`(`*cb_type="tls-unique"*`)`
Get channel binding data for current connection, as a bytes object. Returns `None` if not connected or the handshake has not been completed.

The *cb_type* parameter allow selection of the desired channel binding type. Valid channel binding types are listed in the *CHANNEL_BINDING_TYPES* list. Currently only the 'tls-unique' channel binding, defined by **RFC 5929**, is supported. *ValueError* will be raised if an unsupported channel binding type is requested.

2.7.9 新版功能.

`SSLSocket.`**`selected_alpn_protocol`**`()`
Return the protocol that was selected during the TLS handshake. If *SSLContext.set_alpn_protocols()* was not called, if the other party does not support ALPN, if this socket does not support any of the client's proposed protocols, or if the handshake has not happened yet, `None` is returned.

2.7.10 新版功能.

`SSLSocket.`**`selected_npn_protocol`**`()`
Return the higher-level protocol that was selected during the TLS/SSL handshake. If *SSLContext.set_npn_protocols()* was not called, or if the other party does not support NPN, or if the handshake has not yet happened, this will return `None`.

2.7.9 新版功能.

`SSLSocket.`**`unwrap`**`()`
Performs the SSL shutdown handshake, which removes the TLS layer from the underlying socket, and returns the underlying socket object. This can be used to go from encrypted operation over a connection to unencrypted. The returned socket should always be used for further communication with the other side of the connection, rather than the original socket.

`SSLSocket.`**`version`**`()`
Return the actual SSL protocol version negotiated by the connection as a string, or `None` is no secure connection is established. As of this writing, possible return values include `"SSLv2"`, `"SSLv3"`, `"TLSv1"`, `"TLSv1.1"` and `"TLSv1.2"`. Recent OpenSSL versions may define more return values.

2.7.9 新版功能.

SSLSocket.**context**
The *SSLContext* object this SSL socket is tied to. If the SSL socket was created using the top-level *wrap_socket()* function (rather than *SSLContext.wrap_socket()*), this is a custom context object created for this SSL socket.

2.7.9 新版功能.

17.3.3 SSL Contexts

2.7.9 新版功能.

An SSL context holds various data longer-lived than single SSL connections, such as SSL configuration options, certificate(s) and private key(s). It also manages a cache of SSL sessions for server-side sockets, in order to speed up repeated connections from the same clients.

class ssl.**SSLContext**(*protocol*)
Create a new SSL context. You must pass *protocol* which must be one of the PROTOCOL_* constants defined in this module. *PROTOCOL_SSLv23* is currently recommended for maximum interoperability.

参见:

create_default_context() lets the *ssl* module choose security settings for a given purpose.

在 2.7.16 版更改: The context is created with secure default values. The options *OP_NO_COMPRESSION*, *OP_CIPHER_SERVER_PREFERENCE*, *OP_SINGLE_DH_USE*, *OP_SINGLE_ECDH_USE*, *OP_NO_SSLv2* (except for *PROTOCOL_SSLv2*), and *OP_NO_SSLv3* (except for *PROTOCOL_SSLv3*) are set by default. The initial cipher suite list contains only HIGH ciphers, no NULL ciphers and no MD5 ciphers (except for *PROTOCOL_SSLv2*).

SSLContext objects have the following methods and attributes:

SSLContext.**cert_store_stats**()
Get statistics about quantities of loaded X.509 certificates, count of X.509 certificates flagged as CA certificates and certificate revocation lists as dictionary.

Example for a context with one CA cert and one other cert:

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

SSLContext.**load_cert_chain**(*certfile*, *keyfile=None*, *password=None*)
Load a private key and the corresponding certificate. The *certfile* string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate's authenticity. The *keyfile* string, if present, must point to a file containing the private key in. Otherwise the private key will be taken from *certfile* as well. See the discussion of *Certificates* for more information on how the certificate is stored in the *certfile*.

The *password* argument may be a function to call to get the password for decrypting the private key. It will only be called if the private key is encrypted and a password is necessary. It will be called with no arguments, and it should return a string, bytes, or bytearray. If the return value is a string it will be encoded as UTF-8 before using it to decrypt the key. Alternatively a string, bytes, or bytearray value may be supplied directly as the *password* argument. It will be ignored if the private key is not encrypted and no password is needed.

If the *password* argument is not specified and a password is required, OpenSSL's built-in password prompting mechanism will be used to interactively prompt the user for a password.

An *SSLError* is raised if the private key doesn't match with the certificate.

SSLContext.**load_default_certs**(*purpose=Purpose.SERVER_AUTH*)
Load a set of default "certification authority" (CA) certificates from default locations. On Windows

it loads CA certs from the `CA` and `ROOT` system stores. On other systems it calls *SSLContext.set_default_verify_paths()*. In the future the method may load CA certificates from other locations, too.

The *purpose* flag specifies what kind of CA certificates are loaded. The default settings *Purpose.SERVER_AUTH* loads certificates, that are flagged and trusted for TLS web server authentication (client side sockets). *Purpose.CLIENT_AUTH* loads CA certificates for client certificate verification on the server side.

SSLContext.**load_verify_locations**(*cafile=None*, *capath=None*, *cadata=None*)

Load a set of "certification authority" (CA) certificates used to validate other peers' certificates when *verify_mode* is other than *CERT_NONE*. At least one of *cafile* or *capath* must be specified.

This method can also load certification revocation lists (CRLs) in PEM or DER format. In order to make use of CRLs, *SSLContext.verify_flags* must be configured properly.

The *cafile* string, if present, is the path to a file of concatenated CA certificates in PEM format. See the discussion of *Certificates* for more information about how to arrange the certificates in this file.

The *capath* string, if present, is the path to a directory containing several CA certificates in PEM format, following an OpenSSL specific layout.

The *cadata* object, if present, is either an ASCII string of one or more PEM-encoded certificates or a bytes-like object of DER-encoded certificates. Like with *capath* extra lines around PEM-encoded certificates are ignored but at least one certificate must be present.

SSLContext.**get_ca_certs**(*binary_form=False*)

Get a list of loaded "certification authority" (CA) certificates. If the `binary_form` parameter is *False* each list entry is a dict like the output of *SSLSocket.getpeercert()*. Otherwise the method returns a list of DER-encoded certificates. The returned list does not contain certificates from *capath* unless a certificate was requested and loaded by a SSL connection.

注解: Certificates in a capath directory aren't loaded unless they have been used at least once.

SSLContext.**set_default_verify_paths**()

Load a set of default "certification authority" (CA) certificates from a filesystem path defined when building the OpenSSL library. Unfortunately, there's no easy way to know whether this method succeeds: no error is returned if no certificates are to be found. When the OpenSSL library is provided as part of the operating system, though, it is likely to be configured properly.

SSLContext.**set_ciphers**(*ciphers*)

Set the available ciphers for sockets created with this context. It should be a string in the OpenSSL cipher list format. If no cipher can be selected (because compile-time options or other configuration forbids use of all the specified ciphers), an *SSLError* will be raised.

注解: when connected, the *SSLSocket.cipher()* method of SSL sockets will give the currently selected cipher.

OpenSSL 1.1.1 has TLS 1.3 cipher suites enabled by default. The suites cannot be disabled with *set_ciphers()*.

SSLContext.**set_alpn_protocols**(*protocols*)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of ASCII strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to **RFC 7301**. After a successful handshake, the *SSLSocket.selected_alpn_protocol()* method will return the agreed-upon protocol.

This method will raise *NotImplementedError* if *HAS_ALPN* is False.

OpenSSL 1.1.0 to 1.1.0e will abort the handshake and raise *SSLError* when both sides support ALPN but cannot agree on a protocol. 1.1.0f+ behaves like 1.0.2, *SSLSocket.selected_alpn_protocol()* returns None.

2.7.10 新版功能.

SSLContext.**set_npn_protocols**(*protocols*)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to the NPN draft specification. After a successful handshake, the *SSLSocket.selected_npn_protocol()* method will return the agreed-upon protocol.

This method will raise *NotImplementedError* if *HAS_NPN* is False.

SSLContext.**set_servername_callback**(*server_name_callback*)

Register a callback function that will be called after the TLS Client Hello handshake message has been received by the SSL/TLS server when the TLS client specifies a server name indication. The server name indication mechanism is specified in **RFC 6066** section 3 - Server Name Indication.

Only one callback can be set per `SSLContext`. If *server_name_callback* is `None` then the callback is disabled. Calling this function a subsequent time will disable the previously registered callback.

The callback function, *server_name_callback*, will be called with three arguments; the first being the `ssl.SSLSocket`, the second is a string that represents the server name that the client is intending to communicate (or *None* if the TLS Client Hello does not contain a server name) and the third argument is the original *SSLContext*. The server name argument is the IDNA decoded server name.

A typical use of this callback is to change the `ssl.SSLSocket`'s *SSLSocket.context* attribute to a new object of type *SSLContext* representing a certificate chain that matches the server name.

Due to the early negotiation phase of the TLS connection, only limited methods and attributes are usable like *SSLSocket.selected_alpn_protocol()* and *SSLSocket.context*. *SSLSocket.getpeercert()*, *SSLSocket.getpeercert()*, *SSLSocket.cipher()* and `SSLSocket.compress()` methods require that the TLS connection has progressed beyond the TLS Client Hello and therefore will not contain return meaningful values nor can they be called safely.

The *server_name_callback* function must return `None` to allow the TLS negotiation to continue. If a TLS failure is required, a constant *ALERT_DESCRIPTION_** can be returned. Other return values will result in a TLS fatal error with *ALERT_DESCRIPTION_INTERNAL_ERROR*.

If there is an IDNA decoding error on the server name, the TLS connection will terminate with an *ALERT_DESCRIPTION_INTERNAL_ERROR* fatal TLS alert message to the client.

If an exception is raised from the *server_name_callback* function the TLS connection will terminate with a fatal TLS alert message *ALERT_DESCRIPTION_HANDSHAKE_FAILURE*.

This method will raise *NotImplementedError* if the OpenSSL library had OPENSSL_NO_TLSEXT defined when it was built.

SSLContext.**load_dh_params**(*dhfile*)

Load the key generation parameters for Diffie-Helman (DH) key exchange. Using DH key exchange improves forward secrecy at the expense of computational resources (both on the server and on the client). The *dhfile* parameter should be the path to a file containing DH parameters in PEM format.

This setting doesn't apply to client sockets. You can also use the *OP_SINGLE_DH_USE* option to further improve security.

SSLContext.**set_ecdh_curve**(*curve_name*)

Set the curve name for Elliptic Curve-based Diffie-Hellman (ECDH) key exchange. ECDH is significantly faster than regular DH while arguably as secure. The *curve_name* parameter should be a string describing a well-known elliptic curve, for example `prime256v1` for a widely supported curve.

This setting doesn't apply to client sockets. You can also use the *OP_SINGLE_ECDH_USE* option to further improve security.

This method is not available if *HAS_ECDH* is `False`.

参见:

SSL/TLS & Perfect Forward Secrecy Vincent Bernat.

`SSLContext.`**`wrap_socket`**(*sock*, *server_side=False*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *server_hostname=None*)

Wrap an existing Python socket *sock* and return an `SSLSocket` object. *sock* must be a *SOCK_STREAM* socket; other socket types are unsupported.

The returned SSL socket is tied to the context, its settings and certificates. The parameters *server_side*, *do_handshake_on_connect* and *suppress_ragged_eofs* have the same meaning as in the top-level *wrap_socket()* function.

On client connections, the optional parameter *server_hostname* specifies the hostname of the service which we are connecting to. This allows a single server to host multiple SSL-based services with distinct certificates, quite similarly to HTTP virtual hosts. Specifying *server_hostname* will raise a *ValueError* if *server_side* is true.

在 2.7.9 版更改: Always allow a server_hostname to be passed, even if OpenSSL does not have SNI.

`SSLContext.`**`session_stats`**()

Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each piece of information to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

`SSLContext.`**`check_hostname`**

Wether to match the peer cert's hostname with *match_hostname()* in *SSLSocket.do_handshake()*. The context's *verify_mode* must be set to *CERT_OPTIONAL* or *CERT_REQUIRED*, and you must pass *server_hostname* to *wrap_socket()* in order to match the hostname.

示例:

```
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLS)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

注解: This features requires OpenSSL 0.9.8f or newer.

`SSLContext.`**`options`**

An integer representing the set of SSL options enabled on this context. The default value is *OP_ALL*, but you can specify other options such as *OP_NO_SSLv2* by ORing them together.

> **注解：** With versions of OpenSSL older than 0.9.8m, it is only possible to set options, not to clear them. Attempting to clear an option (by resetting the corresponding bits) will raise a `ValueError`.

`SSLContext.protocol`
The protocol version chosen when constructing the context. This attribute is read-only.

`SSLContext.verify_flags`
The flags for certificate verification operations. You can set flags like *VERIFY_CRL_CHECK_LEAF* by ORing them together. By default OpenSSL does neither require nor verify certificate revocation lists (CRLs). Available only with openssl version 0.9.8+.

`SSLContext.verify_mode`
Whether to try to verify other peers' certificates and how to behave if verification fails. This attribute must be one of *CERT_NONE*, *CERT_OPTIONAL* or *CERT_REQUIRED*.

17.3.4 Certificates

Certificates in general are part of a public-key / private-key system. In this system, each *principal*, (which may be a machine, or a person, or an organization) is assigned a unique two-part encryption key. One part of the key is public, and is called the *public key*; the other part is kept secret, and is called the *private key*. The two parts are related, in that if you encrypt a message with one of the parts, you can decrypt it with the other part, and **only** with the other part.

A certificate contains information about two principals. It contains the name of a *subject*, and the subject's public key. It also contains a statement by a second principal, the *issuer*, that the subject is who they claim to be, and that this is indeed the subject's public key. The issuer's statement is signed with the issuer's private key, which only the issuer knows. However, anyone can verify the issuer's statement by finding the issuer's public key, decrypting the statement with it, and comparing it to the other information in the certificate. The certificate also contains information about the time period over which it is valid. This is expressed as two fields, called "notBefore" and "notAfter".

In the Python use of certificates, a client or server can use a certificate to prove who they are. The other side of a network connection can also be required to produce a certificate, and that certificate can be validated to the satisfaction of the client or server that requires such validation. The connection attempt can be set to raise an exception if the validation fails. Validation is done automatically, by the underlying OpenSSL framework; the application need not concern itself with its mechanics. But the application does usually need to provide sets of certificates to allow this process to take place.

Python uses files to contain certificates. They should be formatted as "PEM" (see **RFC 1422**), which is a base-64 encoded form wrapped with a header line and a footer line:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Certificate chains

The Python files which contain certificates can contain a sequence of certificates, sometimes called a *certificate chain*. This chain should start with the specific certificate for the principal who "is" the client or server, and then the certificate for the issuer of that certificate, and then the certificate for the issuer of *that* certificate, and so on up the chain till you get to a certificate which is *self-signed*, that is, a certificate which has the same subject and issuer, sometimes called a *root certificate*. The certificates should just be concatenated together in the certificate file. For example, suppose we had a three certificate chain, from our server certificate to the certificate of the certification authority that signed our server certificate, to the root certificate of the agency which issued the certification authority's certificate:

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

CA certificates

If you are going to require validation of the other side of the connection's certificate, you need to provide a "CA certs" file, filled with the certificate chains for each issuer you are willing to trust. Again, this file just contains these chains concatenated together. For validation, Python will use the first chain it finds in the file which matches. The platform's certificates file can be used by calling *SSLContext.load_default_certs()*, this is done automatically with *create_default_context()*.

Combined key and certificate

Often the private key is stored in the same file as the certificate; in this case, only the `certfile` parameter to *SSLContext.load_cert_chain()* and *wrap_socket()* needs to be passed. If the private key is stored with the certificate, it should come before the first certificate in the certificate chain:

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Self-signed certificates

If you are going to create a server that provides SSL-encrypted connection services, you will need to acquire a certificate for that service. There are many ways of acquiring appropriate certificates, such as buying one from a certification authority. Another common practice is to generate a self-signed certificate. The simplest way to do this is with the OpenSSL package, using something like the following:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.......++++++
.............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
```

(下页继续)

(续上页)

```
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

The disadvantage of a self-signed certificate is that it is its own root certificate, and no one else will have it in their cache of known (and trusted) root certificates.

17.3.5 例子

Testing for SSL support

To test for the presence of SSL support in a Python installation, user code should use the following idiom:

```
try:
    import ssl
except ImportError:
    pass
else:
    ...  # do something that requires SSL support
```

Client-side operation

This example creates a SSL context with the recommended security settings for client sockets, including automatic certificate verification:

```
>>> context = ssl.create_default_context()
```

If you prefer to tune security settings yourself, you might create a context from scratch (but beware that you might not get the settings right):

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS)
>>> context.verify_mode = ssl.CERT_REQUIRED
>>> context.check_hostname = True
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(this snippet assumes your operating system places a bundle of all CA certificates in `/etc/ssl/certs/ca-bundle.crt`; if not, you'll get an error and have to adjust the location)

When you use the context to connect to a server, *CERT_REQUIRED* validates the server certificate: it ensures that the server certificate was signed with one of the CA certificates, and checks the signature for correctness:

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                            server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

You may then fetch the certificate:

```
>>> cert = conn.getpeercert()
```

Visual inspection shows that the certificate does identify the desired service (that is, the HTTPS host `www.python.org`):

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.crt
↪',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('organizationalUnitName', 'www.digicert.com'),),
            (('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': ((('businessCategory', 'Private Organization'),),
             (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
             (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
             (('serialNumber', '3359300'),),
             (('streetAddress', '16 Allen Rd'),),
             (('postalCode', '03894-4801'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'NH'),),
             (('localityName', 'Wolfeboro,'),),
             (('organizationName', 'Python Software Foundation'),),
             (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.python.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
                    ('DNS', 'test.pythonhosted.org'),
                    ('DNS', 'us.pycon.org'),
                    ('DNS', 'id.python.org')),
 'version': 3}
```

Now the SSL channel is established and the certificate verified, you can proceed to talk with the server:

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
```

(下页继续)

(续上页)

```
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
 b'',
 b'']
```

See the discussion of *Security considerations* below.

Server-side operation

For server operation, typically you'll need to have a server certificate, and private key, each in a file. You'll first create a context holding the key and the certificate, so that clients can check your authenticity. Then you'll open a socket, bind it to a port, call `listen()` on it, and start waiting for clients to connect:

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.mydomain.com', 10023))
bindsocket.listen(5)
```

When a client connects, you'll call `accept()` on the socket to get the new socket from the other end, and use the context's *SSLContext.wrap_socket()* method to create a server-side SSL socket for the connection:

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

Then you'll read data from the `connstream` and do something with it till you are finished with the client (or the client is finished with you):

```
def deal_with_client(connstream):
    data = connstream.read()
    # null data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.read()
    # finished with client
```

And go back to listening for new client connections (of course, a real server would probably handle each client connection in a separate thread, or put the sockets in non-blocking mode and use an event loop).

17.3.6 Notes on non-blocking sockets

When working with non-blocking sockets, there are several things you need to be aware of:

- Calling *select()* tells you that the OS-level socket can be read from (or written to), but it does not imply that there is sufficient data at the upper SSL layer. For example, only part of an SSL frame might have arrived. Therefore, you must be ready to handle `SSLSocket.recv()` and `SSLSocket.send()` failures, and retry after another call to *select()*.
- Conversely, since the SSL layer has its own framing, a SSL socket may still have data available for reading without *select()* being aware of it. Therefore, you should first call `SSLSocket.recv()` to drain any potentially available data, and then only block on a *select()* call if still necessary.

  (of course, similar provisions apply when using other primitives such as *poll()*, or those in the `selectors` module)
- The SSL handshake itself will be non-blocking: the *SSLSocket.do_handshake()* method has to be retried until it returns successfully. Here is a synopsis using *select()* to wait for the socket's readiness:

```
while True:
    try:
        sock.do_handshake()
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

17.3.7 Security considerations

Best defaults

For **client use**, if you don't have any special requirements for your security policy, it is highly recommended that you use the *create_default_context()* function to create your SSL context. It will load the system's trusted CA certificates, enable certificate validation and hostname checking, and try to choose reasonably secure protocol and cipher settings.

If a client certificate is needed for the connection, it can be added with *SSLContext.load_cert_chain()*.

By contrast, if you create the SSL context by calling the *SSLContext* constructor yourself, it will not have certificate validation nor hostname checking enabled by default. If you do so, please read the paragraphs below to achieve a good security level.

Manual settings

Verifying certificates

When calling the *SSLContext* constructor directly, *CERT_NONE* is the default. Since it does not authenticate the other peer, it can be insecure, especially in client mode where most of time you would like to ensure the authenticity of the server you're talking to. Therefore, when in client mode, it is highly recommended to use *CERT_REQUIRED*. However, it is in itself not sufficient; you also have to check that the server certificate, which can be obtained by calling *SSLSocket.getpeercert()*, matches the desired service. For many protocols and applications, the service can be identified by the hostname; in this case, the *match_hostname()* function can be used. This common check is automatically performed when *SSLContext.check_hostname* is enabled.

In server mode, if you want to authenticate your clients using the SSL layer (rather than using a higher-level authentication mechanism), you'll also have to specify *CERT_REQUIRED* and similarly check the client certificate.

> **注解：** In client mode, *CERT_OPTIONAL* and *CERT_REQUIRED* are equivalent unless anonymous ciphers are enabled (they are disabled by default).

Protocol versions

SSL versions 2 and 3 are considered insecure and are therefore dangerous to use. If you want maximum compatibility between clients and servers, it is recommended to use *PROTOCOL_SSLv23* as the protocol version and then disable SSLv2 and SSLv3 explicitly using the *SSLContext.options* attribute:

```
context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
context.options |= ssl.OP_NO_SSLv2
context.options |= ssl.OP_NO_SSLv3
```

The SSL context created above will only allow TLSv1 and later (if supported by your system) connections.

Cipher selection

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating a SSL session is possible through the *SSLContext.set_ciphers()* method. Starting from Python 2.7.9, the ssl module disables certain weak ciphers by default, but you may want to further restrict the cipher choice. Be sure to read OpenSSL's documentation about the cipher list format. If you want to check which ciphers are enabled by a given cipher list, use the `openssl ciphers` command on your system.

Multi-processing

If using this module as part of a multi-processed application (using, for example the *multiprocessing* or `concurrent.futures` modules), be aware that OpenSSL's internal random number generator does not properly handle forked processes. Applications must change the PRNG state of the parent process if they use any SSL feature with *os.fork()*. Any successful call of *RAND_add()*, `RAND_bytes()` or `RAND_pseudo_bytes()` is sufficient.

17.3.8 LibreSSL support

LibreSSL is a fork of OpenSSL 1.0.1. The ssl module has limited support for LibreSSL. Some features are not available when the ssl module is compiled with LibreSSL.

- LibreSSL >= 2.6.1 no longer supports NPN. The methods *SSLContext.set_npn_protocols()* and *SSLSocket.selected_npn_protocol()* are not available.
- *SSLContext.set_default_verify_paths()* ignores the env vars `SSL_CERT_FILE` and `SSL_CERT_PATH` although *get_default_verify_paths()* still reports them.

参见：

Class *socket.socket* Documentation of underlying *socket* class

SSL/TLS Strong Encryption: An Introduction Intro from the Apache webserver documentation

RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management Steve Kent

RFC 1750: Randomness Recommendations for Security D. Eastlake et. al.

RFC 3280: Internet X.509 Public Key Infrastructure Certificate and CRL Profile Housley et. al.

RFC 4366: Transport Layer Security (TLS) Extensions Blake-Wilson et. al.

RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2 T. Dierks et. al.

RFC 6066: Transport Layer Security (TLS) Extensions D. Eastlake

IANA TLS: Transport Layer Security (TLS) Parameters IANA

RFC 7525: Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTL IETF

Mozilla's Server Side TLS recommendations Mozilla

17.4 signal —设置异步事件处理程序

This module provides mechanisms to use signal handlers in Python. Some general rules for working with signals and their handlers:

- 一旦设置，特定信号的处理程序将保持安装，直到它被显式重置（Python 模拟 BSD 样式接口而不管底层实现），但 SIGCHLD 的处理程序除外，它遵循底层实现。
- There is no way to "block" signals temporarily from critical sections (since this is not supported by all Unix flavors).
- Although Python signal handlers are called asynchronously as far as the Python user is concerned, they can only occur between the "atomic" instructions of the Python interpreter. This means that signals arriving during long calculations implemented purely in C (such as regular expression matches on large bodies of text) may be delayed for an arbitrary amount of time.
- When a signal arrives during an I/O operation, it is possible that the I/O operation raises an exception after the signal handler returns. This is dependent on the underlying Unix system's semantics regarding interrupted system calls.
- Because the C signal handler always returns, it makes little sense to catch synchronous errors like SIGFPE or SIGSEGV.
- Python installs a small number of signal handlers by default: SIGPIPE is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and SIGINT is translated into a *KeyboardInterrupt* exception. All of these can be overridden.
- Some care must be taken if both signals and threads are used in the same program. The fundamental thing to remember in using signals and threads simultaneously is: always perform *signal()* operations in the main thread of execution. Any thread can perform an *alarm()*, *getsignal()*, *pause()*, *setitimer()* or *getitimer()*; only the main thread can set a new signal handler, and the main thread will be the only one to receive signals (this is enforced by the Python *signal* module, even if the underlying thread implementation supports sending signals to individual threads). This means that signals can't be used as a means of inter-thread communication. Use locks instead.

在*signal* 模块中定义的变量是：

signal.**SIG_DFL**
这是两种标准信号处理选项之一；它只会执行信号的默认函数。例如，在大多数系统上，对于 SIGQUIT 的默认操作是转储核心并退出，而对于 SIGCHLD 的默认操作是简单地忽略它。

signal.**SIG_IGN**
这是另一个标准信号处理程序，它将简单地忽略给定的信号。

SIG*
All the signal numbers are defined symbolically. For example, the hangup signal is defined as `signal.SIGHUP`; the variable names are identical to the names used in C programs, as found in `<signal.h>`. The Unix man page for '`signal()`' lists the existing signals (on some systems this is *signal(2)*, on others the list is in *signal(7)*). Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

signal.**CTRL_C_EVENT**
对应于 `Ctrl+C` 击键事件的信号。此信号只能用于*os.kill()* 。

Availability: Windows.

2.7 新版功能.

signal.**CTRL_BREAK_EVENT**
对应于 `Ctrl+Break` 击键事件的信号。此信号只能用于*os.kill()* 。

Availability: Windows.

2.7 新版功能.

signal.**NSIG**
比最高信号数多一。

signal.**ITIMER_REAL**
实时递减间隔计时器，并在到期时发送 `SIGALRM` 。

signal.**ITIMER_VIRTUAL**
仅在进程执行时递减间隔计时器，并在到期时发送 SIGVTALRM 。

signal.**ITIMER_PROF**
Decrements interval timer both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

The *signal* module defines one exception:

exception signal.**ItimerError**
Raised to signal an error from the underlying *setitimer()* or *getitimer()* implementation. Expect this error if an invalid interval timer or a negative time is passed to *setitimer()*. This error is a subtype of *IOError*.

The *signal* module defines the following functions:

signal.**alarm**(*time*)
If *time* is non-zero, this function requests that a `SIGALRM` signal be sent to the process in *time* seconds. Any previously scheduled alarm is canceled (only one alarm can be scheduled at any time). The returned value is then the number of seconds before any previously set alarm was to have been delivered. If *time* is zero, no alarm is scheduled, and any scheduled alarm is canceled. If the return value is zero, no alarm is currently scheduled. (See the Unix man page *alarm(2)*.) Availability: Unix.

signal.**getsignal**(*signalnum*)
Return the current signal handler for the signal *signalnum*. The returned value may be a callable Python object, or one of the special values *signal.SIG_IGN*, *signal.SIG_DFL* or *None*. Here, *signal.SIG_IGN* means that the signal was previously ignored, *signal.SIG_DFL* means that the default way of handling the signal was previously in use, and `None` means that the previous signal handler was not installed from Python.

signal.**pause**()
Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing. Not on Windows. (See the Unix man page *signal(2)*.)

signal.**setitimer**(*which*, *seconds*[, *interval*])
Sets given interval timer (one of *signal.ITIMER_REAL*, *signal.ITIMER_VIRTUAL* or *signal.*

ITIMER_PROF) specified by *which* to fire after *seconds* (float is accepted, different from *alarm()*) and after that every *interval* seconds. The interval timer specified by *which* can be cleared by setting seconds to zero.

When an interval timer fires, a signal is sent to the process. The signal sent is dependent on the timer being used; *signal.ITIMER_REAL* will deliver SIGALRM, *signal.ITIMER_VIRTUAL* sends SIGVTALRM, and *signal.ITIMER_PROF* will deliver SIGPROF.

The old values are returned as a tuple: (delay, interval).

Attempting to pass an invalid interval timer will cause an *ItimerError*. Availability: Unix.

2.6 新版功能.

signal.**getitimer**(*which*)
Returns current value of a given interval timer specified by *which*. Availability: Unix.

2.6 新版功能.

signal.**set_wakeup_fd**(*fd*)
Set the wakeup fd to *fd*. When a signal is received, a '\0' byte is written to the fd. This can be used by a library to wakeup a poll or select call, allowing the signal to be fully processed.

The old wakeup fd is returned (or -1 if file descriptor wakeup was not enabled). If *fd* is -1, file descriptor wakeup is disabled. If not -1, *fd* must be non-blocking. It is up to the library to remove any bytes from *fd* before calling poll or select again.

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a *ValueError* exception to be raised.

2.6 新版功能.

signal.**siginterrupt**(*signalnum*, *flag*)
Change system call restart behaviour: if *flag* is *False*, system calls will be restarted when interrupted by signal *signalnum*, otherwise system calls will be interrupted. Returns nothing. Availability: Unix (see the man page *siginterrupt(3)* for further information).

Note that installing a signal handler with *signal()* will reset the restart behaviour to interruptible by implicitly calling siginterrupt() with a true *flag* value for the given signal.

2.6 新版功能.

signal.**signal**(*signalnum*, *handler*)
Set the handler for signal *signalnum* to the function *handler*. *handler* can be a callable Python object taking two arguments (see below), or one of the special values *signal.SIG_IGN* or *signal.SIG_DFL*. The previous signal handler will be returned (see the description of *getsignal()* above). (See the Unix man page *signal(2)*.)

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a *ValueError* exception to be raised.

The *handler* is called with two arguments: the signal number and the current stack frame (None or a frame object; for a description of frame objects, see the description in the type hierarchy or see the attribute descriptions in the *inspect* module).

On Windows, *signal()* can only be called with SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, or SIGTERM. A *ValueError* will be raised in any other case.

17.4.1 示例

Here is a minimal example program. It uses the *alarm()* function to limit the time spent waiting to open a file; this is useful if the file is for a serial device that may not be turned on, which would normally cause the *os.open()* to hang indefinitely. The solution is to set a 5-second alarm before opening the file; if the operation takes too long, the alarm signal will be sent, and the handler raises an exception.

```
import signal, os

def handler(signum, frame):
    print 'Signal handler called with signal', signum
    raise IOError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # Disable the alarm
```

17.5 popen2 —Subprocesses with accessible I/O streams

2.6 版后已移除: This module is obsolete. Use the *subprocess* module. Check especially the *Replacing Older Functions with the subprocess Module* section.

This module allows you to spawn processes and connect to their input/output/error pipes and obtain their return codes under Unix and Windows.

The *subprocess* module provides more powerful facilities for spawning new processes and retrieving their results. Using the *subprocess* module is preferable to using the *popen2* module.

The primary interface offered by this module is a trio of factory functions. For each of these, if *bufsize* is specified, it specifies the buffer size for the I/O pipes. *mode*, if provided, should be the string `'b'` or `'t'`; on Windows this is needed to determine whether the file objects should be opened in binary or text mode. The default value for *mode* is `'t'`.

On Unix, *cmd* may be a sequence, in which case arguments will be passed directly to the program without shell intervention (as with *os.spawnv()*). If *cmd* is a string it will be passed to the shell (as with *os.system()*).

The only way to retrieve the return codes for the child processes is by using the `poll()` or `wait()` methods on the *Popen3* and *Popen4* classes; these are only available on Unix. This information is not available when using the *popen2()*, *popen3()*, and *popen4()* functions, or the equivalent functions in the *os* module. (Note that the tuples returned by the *os* module's functions are in a different order from the ones returned by the *popen2* module.)

`popen2.`**popen2**(*cmd*[, *bufsize*[, *mode*]])
Executes *cmd* as a sub-process. Returns the file objects `(child_stdout, child_stdin)`.

`popen2.`**popen3**(*cmd*[, *bufsize*[, *mode*]])
Executes *cmd* as a sub-process. Returns the file objects `(child_stdout, child_stdin, child_stderr)`.

`popen2.`**popen4**(*cmd*[, *bufsize*[, *mode*]])
Executes *cmd* as a sub-process. Returns the file objects `(child_stdout_and_stderr, child_stdin)`.

2.0 新版功能.

On Unix, a class defining the objects returned by the factory functions is also available. These are not used for the Windows implementation, and are not available on that platform.

class popen2.**Popen3**(*cmd*[, *capturestderr*[, *bufsize*]])
This class represents a child process. Normally, *Popen3* instances are created using the *popen2()* and *popen3()* factory functions described above.

If not using one of the helper functions to create *Popen3* objects, the parameter *cmd* is the shell command to execute in a sub-process. The *capturestderr* flag, if true, specifies that the object should capture standard error output of the child process. The default is false. If the *bufsize* parameter is specified, it specifies the size of the I/O buffers to/from the child process.

class popen2.**Popen4**(*cmd*[, *bufsize*])
Similar to *Popen3*, but always captures standard error into the same file object as standard output. These are typically created using *popen4()*.

2.0 新版功能.

17.5.1 Popen3 and Popen4 Objects

Instances of the *Popen3* and *Popen4* classes have the following methods:

Popen3.**poll**()
Returns -1 if child process hasn't completed yet, or its status code (see *wait()*) otherwise.

Popen3.**wait**()
Waits for and returns the status code of the child process. The status code encodes both the return code of the process and information about whether it exited using the exit() system call or died due to a signal. Functions to help interpret the status code are defined in the *os* module; see section 进程管理 for the W*() family of functions.

The following attributes are also available:

Popen3.**fromchild**
A file object that provides output from the child process. For *Popen4* instances, this will provide both the standard output and standard error streams.

Popen3.**tochild**
A file object that provides input to the child process.

Popen3.**childerr**
A file object that provides error output from the child process, if *capturestderr* was true for the constructor, otherwise None. This will always be None for *Popen4* instances.

Popen3.**pid**
The process ID of the child process.

17.5.2 Flow Control Issues

Any time you are working with any form of inter-process communication, control flow needs to be carefully thought out. This remains the case with the file objects provided by this module (or the *os* module equivalents).

When reading output from a child process that writes a lot of data to standard error while the parent is reading from the child's standard output, a deadlock can occur. A similar situation can occur with other combinations of reads and writes. The essential factors are that more than _PC_PIPE_BUF bytes are being written by one process in a blocking fashion, while the other process is reading from the first process, also in a blocking fashion.

There are several ways to deal with this situation.

The simplest application change, in many cases, will be to follow this model in the parent process:

```
import popen2

r, w, e = popen2.popen3('python slave.py')
e.readlines()
r.readlines()
r.close()
e.close()
w.close()
```

with code like this in the child:

```
import os
import sys

# note that each of these print statements
# writes a single long string

print >>sys.stderr, 400 * 'this is a test\n'
os.close(sys.stderr.fileno())
print >>sys.stdout, 400 * 'this is another test\n'
```

In particular, note that `sys.stderr` must be closed after writing all data, or `readlines()` won't return. Also note that *`os.close()`* must be used, as `sys.stderr.close()` won't close `stderr` (otherwise assigning to `sys.stderr` will silently close it, so no further errors can be printed).

Applications which need to support a more general approach should integrate I/O over pipes with their *`select()`* loops, or use separate threads to read each of the individual files provided by whichever `popen*()` function or `Popen*` class was used.

参见:

Module *`subprocess`* Module for spawning and managing subprocesses.

17.6 `asyncore` —异步 socket 处理器

源码: Lib/asyncore.py

该模块提供用于编写异步套接字服务客户端与服务端的基础构件。

只有两种方法让单个处理器上的程序“同一时间完成不止一件事”。多线程编程是最简单和最流行的方法，但是还有另一种非常不同的技术，它可以让你拥有多线程的几乎所有优点，而无需实际使用多线程。它仅仅在你的程序主要受 I/O 限制时有用，那么。如果你的程序受处理器限制，那么先发制人的预定线程可能就是你真正需要的。但是，网络服务器很少受处理器限制。

如果你的操作系统在其 I/O 库中支持 `select()` 系统调用（几乎所有操作系统），那么你可以使用它来同时处理多个通信通道；在 I/O 正在“后台”时进行其他工作。虽然这种策略看起来很奇怪和复杂，特别是起初，它在很多方面比多线程编程更容易理解和控制。*`asyncore`* 模块为您解决了许多难题，使得构建复杂的高性能网络服务器和客户端的任务变得轻而易举。对于“会话”应用程序和协议，伴侣*`asynchat`* 模块是非常宝贵的。

这两个模块背后的基本思想是创建一个或多个网络 *通道*，类的实例*`asyncore.dispatcher`* 和*`asynchat.async_chat`*。创建通道会将它们添加到全局映射中，如果你不为它提供自己的 *映射*，则由*`loop()`* 函数使用。

一旦创建了初始通道，调用*`loop()`* 函数将激活通道服务，该服务将一直持续到最后一个通道（包括在异步服务期间已添加到映射中的任何通道）关闭。

asyncore.**loop**([*timeout*[, *use_poll*[, *map*[, *count*]]]])
进入一个轮询循环，其在循环计数超出或所有打开的通道关闭后终止。所有参数都是可选的。*count* 形参默认为 None，导致循环仅在所有通道关闭时终止。*timeout* 形参为适当的*select()* 或*poll()* 调用设置超时参数，以秒为单位; 默认值为 30 秒。*use_poll* 形参，如果为 True，则表示*poll()* 应优先使用 select`（默认为 ``False`()）。

The *map* parameter is a dictionary whose items are the channels to watch. As channels are closed they are deleted from their map. If *map* is omitted, a global map is used. Channels (instances of *asyncore.dispatcher*, *asynchat.async_chat* and subclasses thereof) can freely be mixed in the map.

class asyncore.**dispatcher**
The *dispatcher* class is a thin wrapper around a low-level socket object. To make it more useful, it has a few methods for event-handling which are called from the asynchronous loop. Otherwise, it can be treated as a normal non-blocking socket object.

The firing of low-level events at certain times or in certain connection states tells the asynchronous loop that certain higher-level events have taken place. For example, if we have asked for a socket to connect to another host, we know that the connection has been made when the socket becomes writable for the first time (at this point you know that you may write to it with the expectation of success). The implied higher-level events are:

Event	描述
handle_connect()	Implied by the first read or write event
handle_close()	Implied by a read event with no data available
handle_accept()	Implied by a read event on a listening socket

During asynchronous processing, each mapped channel's *readable()* and *writable()* methods are used to determine whether the channel's socket should be added to the list of channels select()ed or poll()ed for read and write events.

Thus, the set of channel events is larger than the basic socket events. The full set of methods that can be overridden in your subclass follows:

handle_read()
Called when the asynchronous loop detects that a read() call on the channel's socket will succeed.

handle_write()
Called when the asynchronous loop detects that a writable socket can be written. Often this method will implement the necessary buffering for performance. For example:

```
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

handle_expt()
Called when there is out of band (OOB) data for a socket connection. This will almost never happen, as OOB is tenuously supported and rarely used.

handle_connect()
Called when the active opener's socket actually makes a connection. Might send a "welcome" banner, or initiate a protocol negotiation with the remote endpoint, for example.

handle_close()
Called when the socket is closed.

handle_error()
Called when an exception is raised and not otherwise handled. The default version prints a condensed traceback.

handle_accept()
Called on listening channels (passive openers) when a connection can be established with a new remote endpoint that has issued a *connect()* call for the local endpoint.

readable()
Called each time around the asynchronous loop to determine whether a channel's socket should be added to the list on which read events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in read events.

writable()
Called each time around the asynchronous loop to determine whether a channel's socket should be added to the list on which write events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in write events.

In addition, each channel delegates or extends many of the socket methods. Most of these are nearly identical to their socket partners.

create_socket(*family*, *type*)
This is identical to the creation of a normal socket, and will use the same options for creation. Refer to the *socket* documentation for information on creating sockets.

connect(*address*)
As with the normal socket object, *address* is a tuple with the first element the host to connect to, and the second the port number.

send(*data*)
Send *data* to the remote end-point of the socket.

recv(*buffer_size*)
Read at most *buffer_size* bytes from the socket's remote end-point. An empty string implies that the channel has been closed from the other end.

Note that *recv()* may raise *socket.error* with *EAGAIN* or *EWOULDBLOCK*, even though *select.select()* or *select.poll()* has reported the socket ready for reading.

listen(*backlog*)
Listen for connections made to the socket. The *backlog* argument specifies the maximum number of queued connections and should be at least 1; the maximum value is system-dependent (usually 5).

bind(*address*)
Bind the socket to *address*. The socket must not already be bound. (The format of *address* depends on the address family —refer to the *socket* documentation for more information.) To mark the socket as re-usable (setting the `SO_REUSEADDR` option), call the *dispatcher* object's `set_reuse_addr()` method.

accept()
Accept a connection. The socket must be bound to an address and listening for connections. The return value can be either `None` or a pair `(conn, address)` where *conn* is a *new* socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection. When `None` is returned it means the connection didn't take place, in which case the server should just ignore this event and keep listening for further incoming connections.

close()
Close the socket. All future operations on the socket object will fail. The remote end-point will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

class asyncore.**dispatcher_with_send**
A *dispatcher* subclass which adds simple buffered output capability, useful for simple clients. For more sophisticated usage use *asynchat.async_chat*.

class asyncore.**file_dispatcher**
A file_dispatcher takes a file descriptor or file object along with an optional map argument and wraps it for use with

the `poll()` or `loop()` functions. If provided a file object or anything with a `fileno()` method, that method will be called and passed to the *file_wrapper* constructor. Availability: UNIX.

class asyncore.**file_wrapper**
A file_wrapper takes an integer file descriptor and calls *os.dup()* to duplicate the handle so that the original handle may be closed independently of the file_wrapper. This class implements sufficient methods to emulate a socket for use by the *file_dispatcher* class. Availability: UNIX.

17.6.1 asyncore Example basic HTTP client

Here is a very basic HTTP client that uses the *dispatcher* class to implement its socket handling:

```
import asyncore, socket

class HTTPClient(asyncore.dispatcher):

    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.connect( (host, 80) )
        self.buffer = 'GET %s HTTP/1.0\r\n\r\n' % path

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()

    def handle_read(self):
        print self.recv(8192)

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]


client = HTTPClient('www.python.org', '/')
asyncore.loop()
```

17.6.2 asyncore Example basic echo server

Here is a basic echo server that uses the *dispatcher* class to accept connections and dispatches the incoming connections to a handler:

```
import asyncore
import socket

class EchoHandler(asyncore.dispatcher_with_send):

    def handle_read(self):
        data = self.recv(8192)
        if data:
```

(下页继续)

(续上页)

```
            self.send(data)

class EchoServer(asyncore.dispatcher):

    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(5)

    def handle_accept(self):
        pair = self.accept()
        if pair is not None:
            sock, addr = pair
            print 'Incoming connection from %s' % repr(addr)
            handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()
```

17.7 asynchat —异步 socket 指令/响应处理器

Source code: Lib/asynchat.py

This module builds on the *asyncore* infrastructure, simplifying asynchronous clients and servers and making it easier to handle protocols whose elements are terminated by arbitrary strings, or are of variable length. *asynchat* defines the abstract class *async_chat* that you subclass, providing implementations of the `collect_incoming_data()` and `found_terminator()` methods. It uses the same asynchronous loop as *asyncore*, and the two types of channel, *asyncore.dispatcher* and *asynchat.async_chat*, can freely be mixed in the channel map. Typically an *asyncore.dispatcher* server channel generates new *asynchat.async_chat* channel objects as it receives incoming connection requests.

class asynchat.**async_chat**

This class is an abstract subclass of *asyncore.dispatcher*. To make practical use of the code you must subclass *async_chat*, providing meaningful *collect_incoming_data()* and *found_terminator()* methods. The *asyncore.dispatcher* methods can be used, although not all make sense in a message/response context.

Like *asyncore.dispatcher*, *async_chat* defines a set of events that are generated by an analysis of socket conditions after a `select()` call. Once the polling loop has been started the *async_chat* object's methods are called by the event-processing framework with no action on the part of the programmer.

Two class attributes can be modified, to improve performance, or possibly even to conserve memory.

ac_in_buffer_size

The asynchronous input buffer size (default `4096`).

ac_out_buffer_size

The asynchronous output buffer size (default `4096`).

Unlike *asyncore.dispatcher*, *async_chat* allows you to define a first-in-first-out queue (fifo) of *producers*. A producer need have only one method, `more()`, which should return data to be transmitted on the channel. The producer indicates exhaustion (*i.e.* that it contains no more data) by having its `more()` method return the

empty string. At this point the *async_chat* object removes the producer from the fifo and starts using the next producer, if any. When the producer fifo is empty the `handle_write()` method does nothing. You use the channel object's *set_terminator()* method to describe how to recognize the end of, or an important breakpoint in, an incoming transmission from the remote endpoint.

To build a functioning *async_chat* subclass your input methods *collect_incoming_data()* and *found_terminator()* must handle the data that the channel receives asynchronously. The methods are described below.

`async_chat.`**`close_when_done`**`()`
Pushes a `None` on to the producer fifo. When this producer is popped off the fifo it causes the channel to be closed.

`async_chat.`**`collect_incoming_data`**`(`*data*`)`
Called with *data* holding an arbitrary amount of received data. The default method, which must be overridden, raises a *NotImplementedError* exception.

`async_chat.`**`discard_buffers`**`()`
In emergencies this method will discard any data held in the input and/or output buffers and the producer fifo.

`async_chat.`**`found_terminator`**`()`
Called when the incoming data stream matches the termination condition set by *set_terminator()*. The default method, which must be overridden, raises a *NotImplementedError* exception. The buffered input data should be available via an instance attribute.

`async_chat.`**`get_terminator`**`()`
Returns the current terminator for the channel.

`async_chat.`**`push`**`(`*data*`)`
Pushes data on to the channel's fifo to ensure its transmission. This is all you need to do to have the channel write the data out to the network, although it is possible to use your own producers in more complex schemes to implement encryption and chunking, for example.

`async_chat.`**`push_with_producer`**`(`*producer*`)`
Takes a producer object and adds it to the producer fifo associated with the channel. When all currently-pushed producers have been exhausted the channel will consume this producer's data by calling its `more()` method and send the data to the remote endpoint.

`async_chat.`**`set_terminator`**`(`*term*`)`
Sets the terminating condition to be recognized on the channel. `term` may be any of three types of value, corresponding to three different ways to handle incoming protocol data.

term	描述
string	Will call *found_terminator()* when the string is found in the input stream
integer	Will call *found_terminator()* when the indicated number of characters have been received
`None`	The channel continues to collect data forever

Note that any data following the terminator will be available for reading by the channel after *found_terminator()* is called.

17.7.1 asynchat - Auxiliary Classes

class asynchat.**fifo**([*list=None*])

A *fifo* holding data which has been pushed by the application but not yet popped for writing to the channel. A *fifo* is a list used to hold data and/or producers until they are required. If the *list* argument is provided then it should contain producers or data items to be written to the channel.

is_empty()

Returns `True` if and only if the fifo is empty.

first()

Returns the least-recently *push()*ed item from the fifo.

push(*data*)

Adds the given data (which may be a string or a producer object) to the producer fifo.

pop()

If the fifo is not empty, returns `True, first()`, deleting the popped item. Returns `False, None` for an empty fifo.

17.7.2 asynchat Example

The following partial example shows how HTTP requests can be read with *async_chat*. A web server might create an `http_request_handler` object for each incoming client connection. Notice that initially the channel terminator is set to match the blank line at the end of the HTTP headers, and a flag indicates that the headers are being read.

Once the headers have been read, if the request is of type POST (indicating that further data are present in the input stream) then the `Content-Length:` header is used to set a numeric terminator to read the right amount of data from the channel.

The `handle_request()` method is called once all relevant input has been marshalled, after setting the channel terminator to `None` to ensure that any extraneous data sent by the web client are ignored.

```
class http_request_handler(asynchat.async_chat):

    def __init__(self, sock, addr, sessions, log):
        asynchat.async_chat.__init__(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = ""
        self.set_terminator("\r\n\r\n")
        self.reading_headers = True
        self.handling = False
        self.cgi_data = None
        self.log = log

    def collect_incoming_data(self, data):
        """Buffer the data"""
        self.ibuffer.append(data)

    def found_terminator(self):
        if self.reading_headers:
            self.reading_headers = False
            self.parse_headers("".join(self.ibuffer))
            self.ibuffer = []
            if self.op.upper() == "POST":
```

(下页继续)

(续上页)

```
                clen = self.headers.getheader("content-length")
                self.set_terminator(int(clen))
            else:
                self.handling = True
                self.set_terminator(None)
                self.handle_request()
        elif not self.handling:
            self.set_terminator(None)  # browsers sometimes over-send
            self.cgi_data = parse(self.headers, "".join(self.ibuffer))
            self.handling = True
            self.ibuffer = []
            self.handle_request()
```

CHAPTER 18

互联网数据处理

本章介绍了支持处理互联网上常用数据格式的模块。

18.1 email —电子邮件与 MIME 处理包

2.2 新版功能.

The *email* package is a library for managing email messages, including MIME and other **RFC 2822**-based message documents. It subsumes most of the functionality in several older standard modules such as *rfc822*, *mimetools*, *multifile*, and other non-standard packages such as mimecntl. It is specifically *not* designed to do any sending of email messages to SMTP (**RFC 2821**), NNTP, or other servers; those are functions of modules such as *smtplib* and *nntplib*. The *email* package attempts to be as RFC-compliant as possible, supporting in addition to **RFC 2822**, such MIME-related RFCs as **RFC 2045**, **RFC 2046**, **RFC 2047**, and **RFC 2231**.

The primary distinguishing feature of the *email* package is that it splits the parsing and generating of email messages from the internal *object model* representation of email. Applications using the *email* package deal primarily with objects; you can add sub-objects to messages, remove sub-objects from messages, completely re-arrange the contents, etc. There is a separate parser and a separate generator which handles the transformation from flat text to the object model, and then back to flat text again. There are also handy subclasses for some common MIME object types, and a few miscellaneous utilities that help with such common tasks as extracting and parsing message field values, creating RFC-compliant dates, etc.

The following sections describe the functionality of the *email* package. The ordering follows a progression that should be common in applications: an email message is read as flat text from a file or other source, the text is parsed to produce the object structure of the email message, this structure is manipulated, and finally, the object tree is rendered back into flat text.

It is perfectly feasible to create the object structure out of whole cloth —i.e. completely from scratch. From there, a similar progression can be taken as above.

Also included are detailed specifications of all the classes and modules that the *email* package provides, the exception classes you might encounter while using the *email* package, some auxiliary utilities, and a few examples. For users of the older mimelib package, or previous versions of the *email* package, a section on differences and porting is provided.

Contents of the *email* package documentation:

18.1.1 email.message: Representing an email message

The central class in the *email* package is the *Message* class, imported from the *email.message* module. It is the base class for the *email* object model. *Message* provides the core functionality for setting and querying header fields, and for accessing message bodies.

Conceptually, a *Message* object consists of *headers* and *payloads*. Headers are **RFC 2822** style field names and values where the field name and value are separated by a colon. The colon is not part of either the field name or the field value.

Headers are stored and returned in case-preserving form but are matched case-insensitively. There may also be a single envelope header, also known as the *Unix-From* header or the `From_` header. The payload is either a string in the case of simple message objects or a list of *Message* objects for MIME container documents (e.g. *multipart/** and *message/rfc822*).

Message objects provide a mapping style interface for accessing the message headers, and an explicit interface for accessing both the headers and the payload. It provides convenience methods for generating a flat text representation of the message object tree, for accessing commonly used header parameters, and for recursively walking over the object tree.

Here are the methods of the *Message* class:

class email.message.**Message**

The constructor takes no arguments.

as_string([*unixfrom*])

Return the entire message flattened as a string. When optional *unixfrom* is `True`, the envelope header is included in the returned string. *unixfrom* defaults to `False`. Flattening the message may trigger changes to the *Message* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it mangles lines that begin with `From`. For more flexibility, instantiate a *Generator* instance and use its *flatten()* method directly. For example:

```
from cStringIO import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=False, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

__str__()

Equivalent to `as_string(unixfrom=True)`.

is_multipart()

Return `True` if the message's payload is a list of sub-*Message* objects, otherwise return `False`. When *is_multipart()* returns `False`, the payload should be a string object.

set_unixfrom(*unixfrom*)

Set the message's envelope header to *unixfrom*, which should be a string.

get_unixfrom()

Return the message's envelope header. Defaults to `None` if the envelope header was never set.

attach(*payload*)

Add the given *payload* to the current payload, which must be `None` or a list of *Message* objects before the call. After the call, the payload will always be a list of *Message* objects. If you want to set the payload to a scalar object (e.g. a string), use *set_payload()* instead.

get_payload([*i*[, *decode*]])
Return the current payload, which will be a list of *Message* objects when *is_multipart()* is True, or a string when *is_multipart()* is False. If the payload is a list and you mutate the list object, you modify the message's payload in place.

With optional argument *i*, *get_payload()* will return the *i*-th element of the payload, counting from zero, if *is_multipart()* is True. An *IndexError* will be raised if *i* is less than 0 or greater than or equal to the number of items in the payload. If the payload is a string (i.e. *is_multipart()* is False) and *i* is given, a *TypeError* is raised.

Optional *decode* is a flag indicating whether the payload should be decoded or not, according to the *Content-Transfer-Encoding* header. When True and the message is not a multipart, the payload will be decoded if this header's value is quoted-printable or base64. If some other encoding is used, or *Content-Transfer-Encoding* header is missing, or if the payload has bogus base64 data, the payload is returned as-is (undecoded). If the message is a multipart and the *decode* flag is True, then None is returned. The default for *decode* is False.

set_payload(*payload*[, *charset*])
Set the entire message object's payload to *payload*. It is the client's responsibility to ensure the payload invariants. Optional *charset* sets the message's default character set; see *set_charset()* for details.

在 2.2.2 版更改: *charset* argument added.

set_charset(*charset*)
Set the character set of the payload to *charset*, which can either be a *Charset* instance (see *email.charset*), a string naming a character set, or None. If it is a string, it will be converted to a *Charset* instance. If *charset* is None, the charset parameter will be removed from the *Content-Type* header (the message will not be otherwise modified). Anything else will generate a *TypeError*.

If there is no existing *MIME-Version* header one will be added. If there is no existing *Content-Type* header, one will be added with a value of *text/plain*. Whether the *Content-Type* header already exists or not, its charset parameter will be set to *charset.output_charset*. If *charset.input_charset* and *charset.output_charset* differ, the payload will be re-encoded to the *output_charset*. If there is no existing *Content-Transfer-Encoding* header, then the payload will be transfer-encoded, if needed, using the specified *Charset*, and a header with the appropriate value will be added. If a *Content-Transfer-Encoding* header already exists, the payload is assumed to already be correctly encoded using that *Content-Transfer-Encoding* and is not modified.

The message will be assumed to be of type *text/**, with the payload either in unicode or encoded with *charset.input_charset*. It will be encoded or converted to *charset.output_charset* and transfer encoded properly, if needed, when generating the plain text representation of the message. MIME headers (*MIME-Version*, *Content-Type*, *Content-Transfer-Encoding*) will be added as needed.

2.2.2 新版功能.

get_charset()
Return the *Charset* instance associated with the message's payload.

2.2.2 新版功能.

The following methods implement a mapping-like interface for accessing the message's **RFC 2822** headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by *keys()*, but in a *Message* object, headers are always returned in the order they appeared in the original message, or were added to the message later. Any header deleted and then re-added are always appended to the end of the header list.

These semantic differences are intentional and are biased toward maximal convenience.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

__len__()
Return the total number of headers, including duplicates.

__contains__(*name*)
Return true if the message object has a field named *name*. Matching is done case-insensitively and *name* should not include the trailing colon. Used for the `in` operator, e.g.:

```
if 'message-id' in myMessage:
    print 'Message-ID:', myMessage['message-id']
```

__getitem__(*name*)
Return the value of the named header field. *name* should not include the colon field separator. If the header is missing, `None` is returned; a *KeyError* is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the *get_all()* method to get the values of all the extant named headers.

__setitem__(*name*, *val*)
Add a header to the message with field name *name* and value *val*. The field is appended to the end of the message's existing fields.

Note that this does *not* overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name *name*, delete the field first, e.g.:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__(*name*)
Delete all occurrences of the field with name *name* from the message's headers. No exception is raised if the named field isn't present in the headers.

has_key(*name*)
Return true if the message contains a header field named *name*, otherwise return false.

keys()
Return a list of all the message's header field names.

values()
Return a list of all the message's field values.

items()
Return a list of 2-tuples containing all the message's field headers and values.

get(*name*[, *failobj*])
Return the value of the named header field. This is identical to *__getitem__()* except that optional *failobj* is returned if the named header is missing (defaults to `None`).

Here are some additional useful methods:

get_all(*name*[, *failobj*])
Return a list of all the values for the field named *name*. If there are no such named headers in the message, *failobj* is returned (defaults to `None`).

add_header(*_name*, *_value*, ***_params*)
Extended header setting. This method is similar to *__setitem__()* except that additional header parameters can be provided as keyword arguments. *_name* is the header field to add and *_value* is the *primary* value for the header.

For each item in the keyword argument dictionary *_params*, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will

be added as `key="value"` unless the value is `None`, in which case only the key will be added. If the value contains non-ASCII characters, it must be specified as a three tuple in the format `(CHARSET, LANGUAGE, VALUE)`, where `CHARSET` is a string naming the charset to be used to encode the value, `LANGUAGE` can usually be set to `None` or the empty string (see **RFC 2231** for other possibilities), and `VALUE` is the string value containing non-ASCII code points.

Here's an example:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example with non-ASCII characters:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

Which produces

```
Content-Disposition: attachment; filename*="iso-8859-1''Fu%DFballer.ppt"
```

replace_header(*_name*, *_value*)
Replace a header. Replace the first header found in the message that matches *_name*, retaining header order and field name case. If no matching header was found, a *KeyError* is raised.

2.2.2 新版功能.

get_content_type()
Return the message's content type. The returned string is coerced to lower case of the form *maintype/subtype*. If there was no *Content-Type* header in the message the default type as given by *get_default_type()* will be returned. Since according to **RFC 2045**, messages always have a default type, *get_content_type()* will always return a value.

RFC 2045 defines a message's default type to be *text/plain* unless it appears inside a *multipart/digest* container, in which case it would be *message/rfc822*. If the *Content-Type* header has an invalid type specification, **RFC 2045** mandates that the default type be *text/plain*.

2.2.2 新版功能.

get_content_maintype()
Return the message's main content type. This is the *maintype* part of the string returned by *get_content_type()*.

2.2.2 新版功能.

get_content_subtype()
Return the message's sub-content type. This is the *subtype* part of the string returned by *get_content_type()*.

2.2.2 新版功能.

get_default_type()
Return the default content type. Most messages have a default content type of *text/plain*, except for messages that are subparts of *multipart/digest* containers. Such subparts have a default content type of *message/rfc822*.

2.2.2 新版功能.

set_default_type(*ctype*)
Set the default content type. *ctype* should either be `text/plain` or `message/rfc822`, although this is not enforced. The default content type is not stored in the `Content-Type` header.

2.2.2 新版功能.

get_params([*failobj*[, *header*[, *unquote*]]])
Return the message's `Content-Type` parameters, as a list. The elements of the returned list are 2-tuples of key/value pairs, as split on the `'='` sign. The left hand side of the `'='` is the key, while the right hand side is the value. If there is no `'='` sign in the parameter the value is the empty string, otherwise the value is as described in `get_param()` and is unquoted if optional *unquote* is `True` (the default).

Optional *failobj* is the object to return if there is no `Content-Type` header. Optional *header* is the header to search instead of `Content-Type`.

在 2.2.2 版更改: *unquote* argument added.

get_param(*param*[, *failobj*[, *header*[, *unquote*]]])
Return the value of the `Content-Type` header's parameter *param* as a string. If the message has no `Content-Type` header or if there is no such parameter, then *failobj* is returned (defaults to `None`).

Optional *header* if given, specifies the message header to use instead of `Content-Type`.

Parameter keys are always compared case insensitively. The return value can either be a string, or a 3-tuple if the parameter was **RFC 2231** encoded. When it's a 3-tuple, the elements of the value are of the form `(CHARSET, LANGUAGE, VALUE)`. Note that both `CHARSET` and `LANGUAGE` can be `None`, in which case you should consider `VALUE` to be encoded in the `us-ascii` charset. You can usually ignore `LANGUAGE`.

If your application doesn't care whether the parameter was encoded as in **RFC 2231**, you can collapse the parameter value by calling `email.utils.collapse_rfc2231_value()`, passing in the return value from `get_param()`. This will return a suitably decoded Unicode string when the value is a tuple, or the original string unquoted if it isn't. For example:

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

In any case, the parameter value (either the returned string, or the `VALUE` item in the 3-tuple) is always unquoted, unless *unquote* is set to `False`.

在 2.2.2 版更改: *unquote* argument added, and 3-tuple return value possible.

set_param(*param*, *value*[, *header*[, *requote*[, *charset*[, *language*]]]])
Set a parameter in the `Content-Type` header. If the parameter already exists in the header, its value will be replaced with *value*. If the `Content-Type` header as not yet been defined for this message, it will be set to `text/plain` and the new parameter value will be appended as per **RFC 2045**.

Optional *header* specifies an alternative header to `Content-Type`, and all parameters will be quoted as necessary unless optional *requote* is `False` (the default is `True`).

If optional *charset* is specified, the parameter will be encoded according to **RFC 2231**. Optional *language* specifies the RFC 2231 language, defaulting to the empty string. Both *charset* and *language* should be strings.

2.2.2 新版功能.

del_param(*param*[, *header*[, *requote*]])
Remove the given parameter completely from the `Content-Type` header. The header will be re-written in place without the parameter or its value. All values will be quoted as necessary unless *requote* is `False` (the default is `True`). Optional *header* specifies an alternative to `Content-Type`.

2.2.2 新版功能.

set_type(*type[, header][, requote]*)
Set the main type and subtype for the *Content-Type* header. *type* must be a string in the form *maintype/subtype*, otherwise a *ValueError* is raised.

This method replaces the *Content-Type* header, keeping all the parameters in place. If *requote* is False, this leaves the existing header's quoting as is, otherwise the parameters will be quoted (the default).

An alternative header can be specified in the *header* argument. When the *Content-Type* header is set a *MIME-Version* header is also added.

2.2.2 新版功能.

get_filename([*failobj*])
Return the value of the filename parameter of the *Content-Disposition* header of the message. If the header does not have a filename parameter, this method falls back to looking for the name parameter on the *Content-Type* header. If neither is found, or the header is missing, then *failobj* is returned. The returned string will always be unquoted as per *email.utils.unquote()*.

get_boundary([*failobj*])
Return the value of the boundary parameter of the *Content-Type* header of the message, or *failobj* if either the header is missing, or has no boundary parameter. The returned string will always be unquoted as per *email.utils.unquote()*.

set_boundary(*boundary*)
Set the boundary parameter of the *Content-Type* header to *boundary*. *set_boundary()* will always quote *boundary* if necessary. A *HeaderParseError* is raised if the message object has no *Content-Type* header.

Note that using this method is subtly different than deleting the old *Content-Type* header and adding a new one with the new boundary via *add_header()*, because *set_boundary()* preserves the order of the *Content-Type* header in the list of headers. However, it does *not* preserve any continuation lines which may have been present in the original *Content-Type* header.

get_content_charset([*failobj*])
Return the charset parameter of the *Content-Type* header, coerced to lower case. If there is no *Content-Type* header, or if that header has no charset parameter, *failobj* is returned.

Note that this method differs from *get_charset()* which returns the *Charset* instance for the default encoding of the message body.

2.2.2 新版功能.

get_charsets([*failobj*])
Return a list containing the character set names in the message. If the message is a *multipart*, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the charset parameter in the *Content-Type* header for the represented subpart. However, if the subpart has no *Content-Type* header, no charset parameter, or is not of the *text* main MIME type, then that item in the returned list will be *failobj*.

walk()
The *walk()* method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use *walk()* as the iterator in a for loop; each iteration returns the next subpart.

Here's an example that prints the MIME type of every part of a multipart message structure:

```
>>> for part in msg.walk():
...     print part.get_content_type()
multipart/report
text/plain
```

(下页继续)

(续上页)

```
message/delivery-status
text/plain
text/plain
message/rfc822
```

在 2.5 版更改: The previously deprecated methods `get_type()`, `get_main_type()`, and `get_subtype()` were removed.

Message objects can also optionally contain two instance attributes, which can be used when generating the plain text of a MIME message.

preamble
The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The *preamble* attribute contains this leading extra-armor text for MIME documents. When the *Parser* discovers some text after the headers but before the first boundary string, it assigns this text to the message's *preamble* attribute. When the *Generator* is writing out the plain text representation of a MIME message, and it finds the message has a *preamble* attribute, it will write this text in the area between the headers and the first boundary. See *email.parser* and *email.generator* for details.

Note that if the message object has no preamble, the *preamble* attribute will be `None`.

epilogue
The *epilogue* attribute acts the same way as the *preamble* attribute, except that it contains text that appears between the last boundary and the end of the message.

在 2.5 版更改: You do not need to set the epilogue to the empty string in order for the *Generator* to print a newline at the end of the file.

defects
The *defects* attribute contains a list of all the problems found when parsing this message. See *email.errors* for a detailed description of the possible parsing defects.

2.4 新版功能.

18.1.2 email.parser: Parsing email messages

Message object structures can be created in one of two ways: they can be created from whole cloth by instantiating *Message* objects and stringing them together via *attach()* and *set_payload()* calls, or they can be created by parsing a flat text representation of the email message.

The *email* package provides a standard parser that understands most email document structures, including MIME documents. You can pass the parser a string or a file object, and the parser will return to you the root *Message* instance of the object structure. For simple, non-MIME messages the payload of this root object will likely be a string containing the text of the message. For MIME messages, the root object will return `True` from its *is_multipart()* method, and the subparts can be accessed via the *get_payload()* and *walk()* methods.

There are actually two parser interfaces available for use, the classic *Parser* API and the incremental *FeedParser* API. The classic *Parser* API is fine if you have the entire text of the message in memory as a string, or if the entire message lives in a file on the file system. *FeedParser* is more appropriate for when you're reading the message from a stream which might block waiting for more input (e.g. reading an email message from a socket). The *FeedParser* can consume and parse the message incrementally, and only returns the root object when you close the parser[1].

[1] As of email package version 3.0, introduced in Python 2.4, the classic *Parser* was re-implemented in terms of the *FeedParser*, so the semantics and results are identical between the two parsers.

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. There is no magical connection between the *email* package's bundled parser and the *Message* class, so your custom parser can create message object trees any way it finds necessary.

FeedParser API

2.4 新版功能.

The *FeedParser*, imported from the `email.feedparser` module, provides an API that is conducive to incremental parsing of email messages, such as would be necessary when reading the text of an email message from a source that can block (e.g. a socket). The *FeedParser* can of course be used to parse an email message fully contained in a string or a file, but the classic *Parser* API may be more convenient for such use cases. The semantics and results of the two parser APIs are identical.

The *FeedParser*'s API is simple; you create an instance, feed it a bunch of text until there's no more to feed it, then close the parser to retrieve the root message object. The *FeedParser* is extremely accurate when parsing standards-compliant messages, and it does a very good job of parsing non-compliant messages, providing information about how a message was deemed broken. It will populate a message object's *defects* attribute with a list of any problems it found in a message. See the *email.errors* module for the list of defects that it can find.

Here is the API for the *FeedParser*:

class email.parser.**FeedParser**([*_factory*])
Create a *FeedParser* instance. Optional *_factory* is a no-argument callable that will be called whenever a new message object is needed. It defaults to the *email.message.Message* class.

feed(*data*)
Feed the *FeedParser* some more data. *data* should be a string containing one or more lines. The lines can be partial and the *FeedParser* will stitch such partial lines together properly. The lines in the string can have any of the common three line endings, carriage return, newline, or carriage return and newline (they can even be mixed).

close()
Closing a *FeedParser* completes the parsing of all previously fed data, and returns the root message object. It is undefined what happens if you feed more data to a closed *FeedParser*.

Parser class API

The *Parser* class, imported from the *email.parser* module, provides an API that can be used to parse a message when the complete contents of the message are available in a string or file. The *email.parser* module also provides a second class, called `HeaderParser` which can be used if you're only interested in the headers of the message. `HeaderParser` can be much faster in these situations, since it does not attempt to parse the message body, instead setting the payload to the raw body as a string. `HeaderParser` has the same API as the *Parser* class.

class email.parser.**Parser**([*_class*])
The constructor for the *Parser* class takes an optional argument *_class*. This must be a callable factory (such as a function or a class), and it is used whenever a sub-message object needs to be created. It defaults to *Message* (see *email.message*). The factory will be called without arguments.

The optional *strict* flag is ignored.

2.4 版后已移除: Because the *Parser* class is a backward compatible API wrapper around the new-in-Python 2.4 *FeedParser*, *all* parsing is effectively non-strict. You should simply stop passing a *strict* flag to the *Parser* constructor.

在 2.2.2 版更改: The *strict* flag was added.

在 2.4 版更改: The *strict* flag was deprecated.

The other public *Parser* methods are:

parse(*fp*[, *headersonly*])
Read all the data from the file-like object *fp*, parse the resulting text, and return the root message object. *fp* must support both the *readline()* and the *read()* methods on file-like objects.

The text contained in *fp* must be formatted as a block of **RFC 2822** style headers and header continuation lines, optionally preceded by an envelope header. The header block is terminated either by the end of the data or by a blank line. Following the header block is the body of the message (which may contain MIME-encoded subparts).

Optional *headersonly* is a flag specifying whether to stop parsing after reading the headers or not. The default is `False`, meaning it parses the entire contents of the file.

在 2.2.2 版更改: The *headersonly* flag was added.

parsestr(*text*[, *headersonly*])
Similar to the *parse()* method, except it takes a string object instead of a file-like object. Calling this method on a string is exactly equivalent to wrapping *text* in a *StringIO* instance first and calling *parse()*.

Optional *headersonly* is as with the *parse()* method.

在 2.2.2 版更改: The *headersonly* flag was added.

Since creating a message object structure from a string or a file object is such a common task, two functions are provided as a convenience. They are available in the top-level *email* package namespace.

`email`.**message_from_string**(*s*[, *_class*[, *strict*]])
Return a message object structure from a string. This is exactly equivalent to `Parser().parsestr(s)`. Optional *_class* and *strict* are interpreted as with the *Parser* class constructor.

在 2.2.2 版更改: The *strict* flag was added.

`email`.**message_from_file**(*fp*[, *_class*[, *strict*]])
Return a message object structure tree from an open file object. This is exactly equivalent to `Parser().parse(fp)`. Optional *_class* and *strict* are interpreted as with the *Parser* class constructor.

在 2.2.2 版更改: The *strict* flag was added.

Here's an example of how you might use this at an interactive Python prompt:

```
>>> import email
>>> msg = email.message_from_string(myString)
```

Additional notes

Here are some notes on the parsing semantics:

- Most non-*multipart* type messages are parsed as a single message object with a string payload. These objects will return `False` for *is_multipart()*. Their *get_payload()* method will return a string object.
- All *multipart* type messages will be parsed as a container message object with a list of sub-message objects for their payload. The outer container message will return `True` for *is_multipart()* and their *get_payload()* method will return the list of *Message* subparts.
- Most messages with a content type of *message/** (e.g. *message/delivery-status* and *message/rfc822*) will also be parsed as container object containing a list payload of length 1. Their *is_multipart()* method will return `True`. The single element in the list payload will be a sub-message object.
- Some non-standards compliant messages may not be internally consistent about their *multipart*-edness. Such messages may have a *Content-Type* header of type *multipart*, but their *is_multipart()* method

may return `False`. If such messages were parsed with the *FeedParser*, they will have an instance of the `MultipartInvariantViolationDefect` class in their *defects* attribute list. See *email.errors* for details.

18.1.3 email.generator: Generating MIME documents

One of the most common tasks is to generate the flat text of the email message represented by a message object structure. You will need to do this if you want to send your message via the *smtplib* module or the *nntplib* module, or print the message on the console. Taking a message object structure and producing a flat text document is the job of the *Generator* class.

Again, as with the *email.parser* module, you aren't limited to the functionality of the bundled generator; you could write one from scratch yourself. However the bundled generator knows how to generate most email in a standards-compliant way, should handle MIME and non-MIME email messages just fine, and is designed so that the transformation from flat text, to a message structure via the *Parser* class, and back to flat text, is idempotent (the input is identical to the output)[1]. On the other hand, using the Generator on a *Message* constructed by program may result in changes to the *Message* object as defaults are filled in.

Here are the public methods of the *Generator* class, imported from the *email.generator* module:

class email.generator.**Generator**(*outfp*[, *mangle_from_*[, *maxheaderlen*]])

The constructor for the *Generator* class takes a file-like object called *outfp* for an argument. *outfp* must support the *write()* method and be usable as the output file in a Python extended print statement.

Optional *mangle_from_* is a flag that, when `True`, puts a > character in front of any line in the body that starts exactly as `From`, i.e. `From` followed by a space at the beginning of the line. This is the only guaranteed portable way to avoid having such lines be mistaken for a Unix mailbox format envelope header separator (see WHY THE CONTENT-LENGTH FORMAT IS BAD for details). *mangle_from_* defaults to `True`, but you might want to set this to `False` if you are not writing Unix mailbox format files.

Optional *maxheaderlen* specifies the longest length for a non-continued header. When a header line is longer than *maxheaderlen* (in characters, with tabs expanded to 8 spaces), the header will be split as defined in the *Header* class. Set to zero to disable header wrapping. The default is 78, as recommended (but not required) by **RFC 2822**.

The other public *Generator* methods are:

flatten(*msg*[, *unixfrom*])

Print the textual representation of the message object structure rooted at *msg* to the output file specified when the *Generator* instance was created. Subparts are visited depth-first and the resulting text will be properly MIME encoded.

Optional *unixfrom* is a flag that forces the printing of the envelope header delimiter before the first **RFC 2822** header of the root message object. If the root object has no envelope header, a standard one is crafted. By default, this is set to `False` to inhibit the printing of the envelope delimiter.

Note that for subparts, no envelope header is ever printed.

2.2.2 新版功能.

clone(*fp*)

Return an independent clone of this *Generator* instance with the exact same options.

2.2.2 新版功能.

write(*s*)

Write the string *s* to the underlying file object, i.e. *outfp* passed to *Generator*'s constructor. This provides just enough file-like API for *Generator* instances to be used in extended print statements.

[1] This statement assumes that you use the appropriate setting for the `unixfrom` argument, and that you set maxheaderlen=0 (which will preserve whatever the input line lengths were). It is also not strictly true, since in many cases runs of whitespace in headers are collapsed into single blanks. The latter is a bug that will eventually be fixed.

As a convenience, see the methods `Message.as_string()` and `str(aMessage)`, a.k.a. `Message.__str__()`, which simplify the generation of a formatted string representation of a message object. For more detail, see *email.message*.

The *email.generator* module also provides a derived class, called *DecodedGenerator* which is like the *Generator* base class, except that non-*text* parts are substituted with a format string representing the part.

class email.generator.**DecodedGenerator**(*outfp*[, *mangle_from_*[, *maxheaderlen*[, *fmt*]]])

This class, derived from *Generator* walks through all the subparts of a message. If the subpart is of main type *text*, then it prints the decoded payload of the subpart. Optional *_mangle_from_* and *maxheaderlen* are as with the *Generator* base class.

If the subpart is not of main type *text*, optional *fmt* is a format string that is used instead of the message payload. *fmt* is expanded with the following keywords, `%(keyword)s` format:

- `type` –Full MIME type of the non-*text* part
- `maintype` –Main MIME type of the non-*text* part
- `subtype` –Sub-MIME type of the non-*text* part
- `filename` –Filename of the non-*text* part
- `description` –Description associated with the non-*text* part
- `encoding` –Content transfer encoding of the non-*text* part

The default value for *fmt* is `None`, meaning

```
[Non-text (%(type)s) part of message omitted, filename %(filename)s]
```

2.2.2 新版功能.

在 2.5 版更改: The previously deprecated method `__call__()` was removed.

备注

18.1.4 email.mime: Creating email and MIME objects from scratch

Ordinarily, you get a message object structure by passing a file or some text to a parser, which parses the text and returns the root message object. However you can also build a complete message structure from scratch, or even individual *Message* objects by hand. In fact, you can also take an existing structure and add new *Message* objects, move them around, etc. This makes a very convenient interface for slicing-and-dicing MIME messages.

You can create a new object structure by creating *Message* instances, adding attachments and all the appropriate headers manually. For MIME messages though, the *email* package provides some convenient subclasses to make things easier.

Here are the classes:

class email.mime.base.**MIMEBase**(*_maintype*, *_subtype*, ***_params*)

Module: `email.mime.base`

This is the base class for all the MIME-specific subclasses of *Message*. Ordinarily you won't create instances specifically of *MIMEBase*, although you could. *MIMEBase* is provided primarily as a convenient base class for more specific MIME-aware subclasses.

_maintype is the *Content-Type* major type (e.g. *text* or *image*), and *_subtype* is the *Content-Type* minor type (e.g. *plain* or *gif*). *_params* is a parameter key/value dictionary and is passed directly to *Message.add_header*.

The *MIMEBase* class always adds a *Content-Type* header (based on *_maintype*, *_subtype*, and *_params*), and a *MIME-Version* header (always set to `1.0`).

class email.mime.nonmultipart.**MIMENonMultipart**
Module: email.mime.nonmultipart

A subclass of *MIMEBase*, this is an intermediate base class for MIME messages that are not *multipart*. The primary purpose of this class is to prevent the use of the *attach()* method, which only makes sense for *multipart* messages. If *attach()* is called, a *MultipartConversionError* exception is raised.

2.2.2 新版功能.

class email.mime.multipart.**MIMEMultipart**(*[_subtype[, boundary[, _subparts[, _params]]]]*)
Module: email.mime.multipart

A subclass of *MIMEBase*, this is an intermediate base class for MIME messages that are *multipart*. Optional *_subtype* defaults to *mixed*, but can be used to specify the subtype of the message. A *Content-Type* header of *multipart/_subtype* will be added to the message object. A *MIME-Version* header will also be added.

Optional *boundary* is the multipart boundary string. When None (the default), the boundary is calculated when needed (for example, when the message is serialized).

_subparts is a sequence of initial subparts for the payload. It must be possible to convert this sequence to a list. You can always attach new subparts to the message by using the *Message.attach* method.

Additional parameters for the *Content-Type* header are taken from the keyword arguments, or passed into the *_params* argument, which is a keyword dictionary.

2.2.2 新版功能.

class email.mime.application.**MIMEApplication**(*_data[, _subtype[, _encoder[, **_params]]]*)
Module: email.mime.application

A subclass of *MIMENonMultipart*, the *MIMEApplication* class is used to represent MIME message objects of major type *application*. *_data* is a string containing the raw byte data. Optional *_subtype* specifies the MIME subtype and defaults to *octet-stream*.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the data for transport. This callable takes one argument, which is the *MIMEApplication* instance. It should use *get_payload()* and *set_payload()* to change the payload to encoded form. It should also add any *Content-Transfer-Encoding* or other headers to the message object as necessary. The default encoding is base64. See the *email.encoders* module for a list of the built-in encoders.

_params are passed straight through to the base class constructor.

2.5 新版功能.

class email.mime.audio.**MIMEAudio**(*_audiodata[, _subtype[, _encoder[, **_params]]]*)
Module: email.mime.audio

A subclass of *MIMENonMultipart*, the *MIMEAudio* class is used to create MIME message objects of major type *audio*. *_audiodata* is a string containing the raw audio data. If this data can be decoded by the standard Python module *sndhdr*, then the subtype will be automatically included in the *Content-Type* header. Otherwise you can explicitly specify the audio subtype via the *_subtype* parameter. If the minor type could not be guessed and *_subtype* was not given, then *TypeError* is raised.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the audio data for transport. This callable takes one argument, which is the *MIMEAudio* instance. It should use *get_payload()* and *set_payload()* to change the payload to encoded form. It should also add any *Content-Transfer-Encoding* or other headers to the message object as necessary. The default encoding is base64. See the *email.encoders* module for a list of the built-in encoders.

_params are passed straight through to the base class constructor.

class email.mime.image.**MIMEImage**(*_imagedata*[, *_subtype*[, *_encoder*[, ***_params*]]])
Module: email.mime.image

A subclass of *MIMENonMultipart*, the *MIMEImage* class is used to create MIME message objects of major type *image*. *_imagedata* is a string containing the raw image data. If this data can be decoded by the standard Python module *imghdr*, then the subtype will be automatically included in the *Content-Type* header. Otherwise you can explicitly specify the image subtype via the *_subtype* parameter. If the minor type could not be guessed and *_subtype* was not given, then *TypeError* is raised.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the image data for transport. This callable takes one argument, which is the *MIMEImage* instance. It should use *get_payload()* and *set_payload()* to change the payload to encoded form. It should also add any *Content-Transfer-Encoding* or other headers to the message object as necessary. The default encoding is base64. See the *email.encoders* module for a list of the built-in encoders.

_params are passed straight through to the *MIMEBase* constructor.

class email.mime.message.**MIMEMessage**(*_msg*[, *_subtype*])
Module: email.mime.message

A subclass of *MIMENonMultipart*, the *MIMEMessage* class is used to create MIME objects of main type *message*. *_msg* is used as the payload, and must be an instance of class *Message* (or a subclass thereof), otherwise a *TypeError* is raised.

Optional *_subtype* sets the subtype of the message; it defaults to *rfc822*.

class email.mime.text.**MIMEText**(*_text*[, *_subtype*[, *_charset*]])
Module: email.mime.text

A subclass of *MIMENonMultipart*, the *MIMEText* class is used to create MIME objects of major type *text*. *_text* is the string for the payload. *_subtype* is the minor type and defaults to *plain*. *_charset* is the character set of the text and is passed as a parameter to the *MIMENonMultipart* constructor; it defaults to us-ascii. If *_text* is unicode, it is encoded using the *output_charset* of *_charset*, otherwise it is used as-is.

在 2.4 版更改: The previously deprecated *_encoding* argument has been removed. Content Transfer Encoding now happens implicitly based on the *_charset* argument.

Unless the _charset parameter is explicitly set to None, the MIMEText object created will have both a *Content-Type* header with a charset parameter, and a *Content-Transfer-Encoding* header. This means that a subsequent set_payload call will not result in an encoded payload, even if a charset is passed in the set_payload command. You can "reset" this behavior by deleting the Content-Transfer-Encoding header, after which a set_payload call will automatically encode the new payload (and add a new *Content-Transfer-Encoding* header).

18.1.5 email.header: Internationalized headers

RFC 2822 is the base standard that describes the format of email messages. It derives from the older **RFC 822** standard which came into widespread use at a time when most email was composed of ASCII characters only. **RFC 2822** is a specification written assuming email contains only 7-bit ASCII characters.

Of course, as email has been deployed worldwide, it has become internationalized, such that language specific character sets can now be used in email messages. The base standard still requires email messages to be transferred using only 7-bit ASCII characters, so a slew of RFCs have been written describing how to encode email containing non-ASCII characters into **RFC 2822**-compliant format. These RFCs include **RFC 2045**, **RFC 2046**, **RFC 2047**, and **RFC 2231**. The *email* package supports these standards in its *email.header* and *email.charset* modules.

If you want to include non-ASCII characters in your email headers, say in the *Subject* or *To* fields, you should use the *Header* class and assign the field in the *Message* object to an instance of *Header* instead of using a string for the header value. Import the *Header* class from the *email.header* module. For example:

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> print msg.as_string()
Subject: =?iso-8859-1?q?p=F6stal?=
```

Notice here how we wanted the *Subject* field to contain a non-ASCII character? We did this by creating a *Header* instance and passing in the character set that the byte string was encoded in. When the subsequent *Message* instance was flattened, the *Subject* field was properly **RFC 2047** encoded. MIME-aware mail readers would show this header using the embedded ISO-8859-1 character.

2.2.2 新版功能.

Here is the *Header* class description:

class email.header.**Header**([*s*[, *charset*[, *maxlinelen*[, *header_name*[, *continuation_ws*[, *errors*]]]]]])

Create a MIME-compliant header that can contain strings in different character sets.

Optional *s* is the initial header value. If None (the default), the initial header value is not set. You can later append to the header with *append()* method calls. *s* may be a byte string or a Unicode string, but see the *append()* documentation for semantics.

Optional *charset* serves two purposes: it has the same meaning as the *charset* argument to the *append()* method. It also sets the default character set for all subsequent *append()* calls that omit the *charset* argument. If *charset* is not provided in the constructor (the default), the us-ascii character set is used both as *s*' s initial charset and as the default for subsequent *append()* calls.

The maximum line length can be specified explicitly via *maxlinelen*. For splitting the first line to a shorter value (to account for the field header which isn' t included in *s*, e.g. *Subject*) pass in the name of the field in *header_name*. The default *maxlinelen* is 76, and the default value for *header_name* is None, meaning it is not taken into account for the first line of a long, split header.

Optional *continuation_ws* must be **RFC 2822**-compliant folding whitespace, and is usually either a space or a hard tab character. This character will be prepended to continuation lines. *continuation_ws* defaults to a single space character (" ").

Optional *errors* is passed straight through to the *append()* method.

append(*s*[, *charset*[, *errors*]])

Append the string *s* to the MIME header.

Optional *charset*, if given, should be a *Charset* instance (see *email.charset*) or the name of a character set, which will be converted to a *Charset* instance. A value of None (the default) means that the *charset* given in the constructor is used.

s may be a byte string or a Unicode string. If it is a byte string (i.e. isinstance(s, str) is true), then *charset* is the encoding of that byte string, and a *UnicodeError* will be raised if the string cannot be decoded with that character set.

If *s* is a Unicode string, then *charset* is a hint specifying the character set of the characters in the string. In this case, when producing an **RFC 2822**-compliant header using **RFC 2047** rules, the Unicode string will be encoded using the following charsets in order: us-ascii, the *charset* hint, utf-8. The first character set to not provoke a *UnicodeError* is used.

Optional *errors* is passed through to any *unicode()* or unicode.encode() call, and defaults to "strict" .

encode([*splitchars*])
Encode a message header into an RFC-compliant format, possibly wrapping long lines and encapsulating non-ASCII parts in base64 or quoted-printable encodings. Optional *splitchars* is a string containing characters to split long ASCII lines on, in rough support of **RFC 2822**'s *highest level syntactic breaks*. This doesn't affect **RFC 2047** encoded lines.

The *Header* class also provides a number of methods to support standard operators and built-in functions.

__str__()
A synonym for *Header.encode()*. Useful for `str(aHeader)`.

__unicode__()
A helper for the built-in *unicode()* function. Returns the header as a Unicode string.

__eq__(*other*)
This method allows you to compare two *Header* instances for equality.

__ne__(*other*)
This method allows you to compare two *Header* instances for inequality.

The *email.header* module also provides the following convenient functions.

email.header.**decode_header**(*header*)
Decode a message header value without converting the character set. The header value is in *header*.

This function returns a list of `(decoded_string, charset)` pairs containing each of the decoded parts of the header. *charset* is `None` for non-encoded parts of the header, otherwise a lower case string containing the name of the character set specified in the encoded string.

Here's an example:

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[('p\xf6stal', 'iso-8859-1')]
```

email.header.**make_header**(*decoded_seq*[, *maxlinelen*[, *header_name*[, *continuation_ws*]]])
Create a *Header* instance from a sequence of pairs as returned by *decode_header()*.

decode_header() takes a header value string and returns a sequence of pairs of the format `(decoded_string, charset)` where *charset* is the name of the character set.

This function takes one of those sequence of pairs and returns a *Header* instance. Optional *maxlinelen*, *header_name*, and *continuation_ws* are as in the *Header* constructor.

18.1.6 email.charset: Representing character sets

This module provides a class *Charset* for representing character sets and character set conversions in email messages, as well as a character set registry and several convenience methods for manipulating this registry. Instances of *Charset* are used in several other modules within the *email* package.

Import this class from the *email.charset* module.

2.2.2 新版功能.

class email.charset.**Charset**([*input_charset*])
Map character sets to their email properties.

This class provides information about the requirements imposed on email for a specific character set. It also provides convenience routines for converting between character sets, given the availability of the applicable codecs. Given a character set, it will do its best to provide information on how to use that character set in an email message in an RFC-compliant way.

Certain character sets must be encoded with quoted-printable or base64 when used in email headers or bodies. Certain character sets must be converted outright, and are not allowed in email.

Optional *input_charset* is as described below; it is always coerced to lower case. After being alias normalized it is also used as a lookup into the registry of character sets to find out the header encoding, body encoding, and output conversion codec to be used for the character set. For example, if *input_charset* is `iso-8859-1`, then headers and bodies will be encoded using quoted-printable and no output conversion codec is necessary. If *input_charset* is `euc-jp`, then headers will be encoded with base64, bodies will not be encoded, but output text will be converted from the `euc-jp` character set to the `iso-2022-jp` character set.

Charset instances have the following data attributes:

input_charset
The initial character set specified. Common aliases are converted to their *official* email names (e.g. `latin_1` is converted to `iso-8859-1`). Defaults to 7-bit `us-ascii`.

header_encoding
If the character set must be encoded before it can be used in an email header, this attribute will be set to `Charset.QP` (for quoted-printable), `Charset.BASE64` (for base64 encoding), or `Charset.SHORTEST` for the shortest of QP or BASE64 encoding. Otherwise, it will be `None`.

body_encoding
Same as *header_encoding*, but describes the encoding for the mail message's body, which indeed may be different than the header encoding. `Charset.SHORTEST` is not allowed for *body_encoding*.

output_charset
Some character sets must be converted before they can be used in email headers or bodies. If the *input_charset* is one of them, this attribute will contain the name of the character set output will be converted to. Otherwise, it will be `None`.

input_codec
The name of the Python codec used to convert the *input_charset* to Unicode. If no conversion codec is necessary, this attribute will be `None`.

output_codec
The name of the Python codec used to convert Unicode to the *output_charset*. If no conversion codec is necessary, this attribute will have the same value as the *input_codec*.

Charset instances also have the following methods:

get_body_encoding()
Return the content transfer encoding used for body encoding.

This is either the string `quoted-printable` or `base64` depending on the encoding used, or it is a function, in which case you should call the function with a single argument, the Message object being encoded. The function should then set the *Content-Transfer-Encoding* header itself to whatever is appropriate.

Returns the string `quoted-printable` if *body_encoding* is `QP`, returns the string `base64` if *body_encoding* is `BASE64`, and returns the string `7bit` otherwise.

convert(*s*)
Convert the string *s* from the *input_codec* to the *output_codec*.

to_splittable(*s*)
Convert a possibly multibyte string to a safely splittable format. *s* is the string to split.

Uses the *input_codec* to try and convert the string to Unicode, so it can be safely split on character boundaries (even for multibyte characters).

Returns the string as-is if it isn't known how to convert *s* to Unicode with the *input_charset*.

Characters that could not be converted to Unicode will be replaced with the Unicode replacement character `'U+FFFD'`.

from_splittable(*ustr*[, *to_output*])
Convert a splittable string back into an encoded string. *ustr* is a Unicode string to "unsplit".

This method uses the proper codec to try and convert the string from Unicode back into an encoded format. Return the string as-is if it is not Unicode, or if it could not be converted from Unicode.

Characters that could not be converted from Unicode will be replaced with an appropriate character (usually `'?'`).

If *to_output* is `True` (the default), uses *output_codec* to convert to an encoded format. If *to_output* is `False`, it uses *input_codec*.

get_output_charset()
Return the output character set.

This is the *output_charset* attribute if that is not `None`, otherwise it is *input_charset*.

encoded_header_len()
Return the length of the encoded header string, properly calculating for quoted-printable or base64 encoding.

header_encode(*s*[, *convert*])
Header-encode the string *s*.

If *convert* is `True`, the string will be converted from the input charset to the output charset automatically. This is not useful for multibyte character sets, which have line length issues (multibyte characters must be split on a character, not a byte boundary); use the higher-level *Header* class to deal with these issues (see *email.header*). *convert* defaults to `False`.

The type of encoding (base64 or quoted-printable) will be based on the *header_encoding* attribute.

body_encode(*s*[, *convert*])
Body-encode the string *s*.

If *convert* is `True` (the default), the string will be converted from the input charset to output charset automatically. Unlike *header_encode()*, there are no issues with byte boundaries and multibyte charsets in email bodies, so this is usually pretty safe.

The type of encoding (base64 or quoted-printable) will be based on the *body_encoding* attribute.

The *Charset* class also provides a number of methods to support standard operations and built-in functions.

__str__()
Returns *input_charset* as a string coerced to lower case. `__repr__()` is an alias for *__str__()*.

__eq__(*other*)
This method allows you to compare two *Charset* instances for equality.

__ne__(*other*)
This method allows you to compare two *Charset* instances for inequality.

The *email.charset* module also provides the following functions for adding new entries to the global character set, alias, and codec registries:

email.charset.**add_charset**(*charset*[, *header_enc*[, *body_enc*[, *output_charset*]]])
Add character properties to the global registry.

charset is the input character set, and must be the canonical name of a character set.

Optional *header_enc* and *body_enc* is either `Charset.QP` for quoted-printable, `Charset.BASE64` for base64 encoding, `Charset.SHORTEST` for the shortest of quoted-printable or base64 encoding, or `None` for no encoding. `SHORTEST` is only valid for *header_enc*. The default is `None` for no encoding.

Optional *output_charset* is the character set that the output should be in. Conversions will proceed from input charset, to Unicode, to the output charset when the method *Charset.convert()* is called. The default is to output in the same character set as the input.

Both *input_charset* and *output_charset* must have Unicode codec entries in the module's character set-to-codec mapping; use *add_codec()* to add codecs the module does not know about. See the *codecs* module's documentation for more information.

The global character set registry is kept in the module global dictionary CHARSETS.

email.charset.**add_alias**(*alias*, *canonical*)
Add a character set alias. *alias* is the alias name, e.g. latin-1. *canonical* is the character set's canonical name, e.g. iso-8859-1.

The global charset alias registry is kept in the module global dictionary ALIASES.

email.charset.**add_codec**(*charset*, *codecname*)
Add a codec that map characters in the given character set to and from Unicode.

charset is the canonical name of a character set. *codecname* is the name of a Python codec, as appropriate for the second argument to the *unicode()* built-in, or to the encode() method of a Unicode string.

18.1.7 email.encoders: 编码器

当创建全新的*Message*对象时，你经常需要对载荷编码以便通过兼容的邮件服务器进行传输。对于包含二进制数据的 *image/** 和 *text/** 类型的消息来说尤其如此。

email 包在其 encoders 模块中提供了一些方便的编码。这些编码器实际上由*MIMEAudio* 和*MIMEImage* 类构造函数使用，以提供默认编码。所有编码器函数只接受一个参数，即要编码的消息对象。它们通常提取有效数据，对其进行编码，并将有效数据重置为此新编码的值。他们还应该根据需要设置 *Content-Transfer-Encoding* 标头。

请注意，这些函数对于多段消息没有意义。它们必须应用到各个单独的段上面，而不是整体。如果直接传递一个多段类型的消息，会产生一个*TypeError* 错误。

下面是提供的编码函数：

email.encoders.**encode_quopri**(*msg*)
将有效数据编码为 quoted-printable 形式，并将:mailheader:*Content-Transfer-Encoding*标头设置为 *"quoted-printable*[1]. 。当大多数实际的数据是普通的可打印数据但包含少量不可打印的字符时，这是一个很好的编码。

email.encoders.**encode_base64**(*msg*)
将有效载荷编码为 base64 形式，并将 *Content-Transfer-Encoding* 标头设为 base64。当你的载荷主要包含不可打印数据时这是一种很好用的编码格式，因为它比 quoted-printable 更紧凑。base64 编码格式的缺点是它会使文本变成人类不可读的形式。

email.encoders.**encode_7or8bit**(*msg*)
这并不实际改变消息的有效载荷，但它会基于载荷数据将 *Content-Transfer-Encoding* 标头相应地设为 7bit 或 8bit。

email.encoders.**encode_noop**(*msg*)
这样什么都不会做；它甚至不会设置 *Content-Transfer-Encoding* 标头。

[1] 请注意使用*encode_quopri()* 编码格式还会对数据中的所有制表符和空格符进行编码。

备注

18.1.8 email.errors: 异常和缺陷类

以下异常类在*email.errors* 模块中定义:

exception email.errors.**MessageError**
这是*email* 包可以引发的所有异常的基类。它源自标准异常*Exception* 类，这个类没有定义其他方法。

exception email.errors.**MessageParseError**
This is the base class for exceptions raised by the *Parser* class. It is derived from *MessageError*.

exception email.errors.**HeaderParseError**
Raised under some error conditions when parsing the **RFC 2822** headers of a message, this class is derived from *MessageParseError*. It can be raised from the *Parser.parse* or *Parser.parsestr* methods.

Situations where it can be raised include finding an envelope header after the first **RFC 2822** header of the message, finding a continuation line before the first **RFC 2822** header is found, or finding a line in the headers which is neither a header or a continuation line.

exception email.errors.**BoundaryError**
Raised under some error conditions when parsing the **RFC 2822** headers of a message, this class is derived from *MessageParseError*. It can be raised from the *Parser.parse* or *Parser.parsestr* methods.

Situations where it can be raised include not being able to find the starting or terminating boundary in a *multipart/** message when strict parsing is used.

exception email.errors.**MultipartConversionError**
当使用 add_payload() 将有效负载添加到*Message* 对象时，有效负载已经是一个标量，而消息的*Content-Type* 主类型不是 *multipart* 或者缺少时触发该异常。*MultipartConversionError* 多重继承自*MessageError* 和内置的*TypeError*。

由于 Message.add_payload() 已被弃用，此异常实际上极少会被引发。但是如果在派生自*MIMENonMultipart* 的类 (例如*MIMEImage*) 的实例上调用*attach()* 方法也可以引发此异常。

Here's the list of the defects that the *FeedParser* can find while parsing messages. Note that the defects are added to the message where the problem was found, so for example, if a message nested inside a *multipart/alternative* had a malformed header, that nested message object would have a defect, but the containing messages would not.

All defect classes are subclassed from email.errors.MessageDefect, but this class is *not* an exception!

2.4 新版功能: All the defect classes were added.

- NoBoundaryInMultipartDefect –一条消息宣称有多个部分，但却没有 *boundary* 形参。
- StartBoundaryNotFoundDefect –在 *Content-Type* 标头中宣称的开始边界无法被找到。
- FirstHeaderLineIsContinuationDefect –消息以一个继续行作为其第一个标头行。
- MisplacedEnvelopeHeaderDefect - 在标头块中间发现了一个“Unix From”标头。
- MalformedHeaderDefect –找到一个缺失了冒号或格式错误的标头。
- MultipartInvariantViolationDefect –A message claimed to be a *multipart*, but no subparts were found. Note that when a message has this defect, its *is_multipart()* method may return false even though its content type claims to be *multipart*.

18.1.9 email.utils: 其他工具

There are several useful utilities provided in the *email.utils* module:

email.utils.**quote**(*str*)
Return a new string with backslashes in *str* replaced by two backslashes, and double quotes replaced by backslash-double quote.

email.utils.**unquote**(*str*)
Return a new string which is an *unquoted* version of *str*. If *str* ends and begins with double quotes, they are stripped off. Likewise if *str* ends and begins with angle brackets, they are stripped off.

email.utils.**parseaddr**(*address*)
Parse address –which should be the value of some address-containing field such as *To* or *Cc* –into its constituent *realname* and *email address* parts. Returns a tuple of that information, unless the parse fails, in which case a 2-tuple of `('', '')` is returned.

email.utils.**formataddr**(*pair*)
The inverse of *parseaddr()*, this takes a 2-tuple of the form `(realname, email_address)` and returns the string value suitable for a *To* or *Cc* header. If the first element of *pair* is false, then the second element is returned unmodified.

email.utils.**getaddresses**(*fieldvalues*)
This method returns a list of 2-tuples of the form returned by `parseaddr()`. *fieldvalues* is a sequence of header field values as might be returned by *Message.get_all*. Here's a simple example that gets all the recipients of a message:

```
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

email.utils.**parsedate**(*date*)
Attempts to parse a date according to the rules in **RFC 2822**. however, some mailers don't follow that format as specified, so *parsedate()* tries to guess correctly in such cases. *date* is a string containing an **RFC 2822** date, such as `"Mon, 20 Nov 1995 19:12:08 -0500"`. If it succeeds in parsing the date, *parsedate()* returns a 9-tuple that can be passed directly to *time.mktime()*; otherwise `None` will be returned. Note that indexes 6, 7, and 8 of the result tuple are not usable.

email.utils.**parsedate_tz**(*date*)
Performs the same function as *parsedate()*, but returns either `None` or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to *time.mktime()*, and the tenth is the offset of the date's timezone from UTC (which is the official term for Greenwich Mean Time)[1]. If the input string has no timezone, the last element of the tuple returned is `None`. Note that indexes 6, 7, and 8 of the result tuple are not usable.

email.utils.**mktime_tz**(*tuple*)
Turn a 10-tuple as returned by *parsedate_tz()* into a UTC timestamp (seconds since the Epoch). If the timezone item in the tuple is `None`, assume local time.

email.utils.**formatdate**(*[timeval[, localtime][, usegmt]]*)
Returns a date string as per **RFC 2822**, e.g.:

[1] Note that the sign of the timezone offset is the opposite of the sign of the `time.timezone` variable for the same timezone; the latter variable follows the POSIX standard while this module follows **RFC 2822**.

```
Fri, 09 Nov 2001 01:08:47 -0000
```

Optional *timeval* if given is a floating point time value as accepted by `time.gmtime()` and `time.localtime()`, otherwise the current time is used.

Optional *localtime* is a flag that when `True`, interprets *timeval*, and returns a date relative to the local timezone instead of UTC, properly taking daylight savings time into account. The default is `False` meaning UTC is used.

Optional *usegmt* is a flag that when `True`, outputs a date string with the timezone as an ascii string `GMT`, rather than a numeric `-0000`. This is needed for some protocols (such as HTTP). This only applies when *localtime* is `False`. The default is `False`.

2.4 新版功能.

`email.utils.`**`make_msgid`**(*[idstring]*)
Returns a string suitable for an **RFC 2822**-compliant `Message-ID` header. Optional *idstring* if given, is a string used to strengthen the uniqueness of the message id.

`email.utils.`**`decode_rfc2231`**(*s*)
Decode the string *s* according to **RFC 2231**.

`email.utils.`**`encode_rfc2231`**(*s*[, *charset*[, *language*]])
Encode the string *s* according to **RFC 2231**. Optional *charset* and *language*, if given is the character set name and language name to use. If neither is given, *s* is returned as-is. If *charset* is given but *language* is not, the string is encoded using the empty string for *language*.

`email.utils.`**`collapse_rfc2231_value`**(*value*[, *errors*[, *fallback_charset*]])
When a header parameter is encoded in **RFC 2231** format, `Message.get_param` may return a 3-tuple containing the character set, language, and value. `collapse_rfc2231_value()` turns this into a unicode string. Optional *errors* is passed to the *errors* argument of the built-in `unicode()` function; it defaults to `replace`. Optional *fallback_charset* specifies the character set to use if the one in the **RFC 2231** header is not known by Python; it defaults to `us-ascii`.

For convenience, if the *value* passed to `collapse_rfc2231_value()` is not a tuple, it should be a string and it is returned unquoted.

`email.utils.`**`decode_params`**(*params*)
Decode parameters list according to **RFC 2231**. *params* is a sequence of 2-tuples containing elements of the form `(content-type, string-value)`.

在 2.4 版更改: The `dump_address_pair()` function has been removed; use `formataddr()` instead.

在 2.4 版更改: The `decode()` function has been removed; use the `Header.decode_header` method instead.

在 2.4 版更改: The `encode()` function has been removed; use the `Header.encode` method instead.

备注

18.1.10 `email.iterators`: 迭代器

通过`Message.walk`方法来迭代消息对象树是相当容易的。`email.iterators`模块提供了一些适用于消息对象树的高层级迭代器。

`email.iterators.`**`body_line_iterator`**(*msg*[, *decode*])
此对象会迭代 *msg* 的所有子部分中的所有载荷，逐行返回字符串载荷。它会跳过所有子部分的标头，并且它也会跳过任何包含不为 Python 字符串的载荷的子部分。这基本上等价于使用`readline()`从一个文件读取消息的纯文本表示形式，并跳过所有中间的标头。

可选的 *decode* 会被传递给`Message.get_payload`。

email.iterators.**typed_subpart_iterator**(*msg*[, *maintype*[, *subtype*]])
此函数会迭代 *msg* 的所有子部分，只返回其中与 *maintype* 和 *subtype* 所指定的 MIME 类型相匹配的子部分。

请注意 *subtype* 是可选项；如果省略，则仅使用主类型来进行子部分 MIME 类型的匹配。*maintype* 也是可选项；它的默认值为 *text*。

因此，在默认情况下*typed_subpart_iterator()* 会返回每一个 MIME 类型为 *text/** 的子部分。

增加了以下函数作为有用的调试工具。它 不应当被视为该包所支持的公共接口的组成部分。

email.iterators.**_structure**(*msg*[, *fp*[, *level*]])
Prints an indented representation of the content types of the message object structure. For example:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
    text/plain
    text/plain
    multipart/digest
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
    text/plain
```

Optional *fp* is a file-like object to print the output to. It must be suitable for Python's extended print statement. *level* is used internally.

18.1.11 email: Examples

Here are a few examples of how to use the *email* package to read, write, and send simple email messages, as well as more complex MIME messages.

First, let's see how to create and send a simple text message:

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.mime.text import MIMEText

# Open a plain text file for reading.  For this example, assume that
# the text file contains only ASCII characters.
fp = open(textfile, 'rb')
# Create a text/plain message
msg = MIMEText(fp.read())
fp.close()

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = 'The contents of %s' % textfile
```

(下页继续)

(续上页)

```
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server, but don't include the
# envelope header.
s = smtplib.SMTP('localhost')
s.sendmail(me, [you], msg.as_string())
s.quit()
```

And parsing RFC822 headers can easily be done by the parse(filename) or parsestr(message_as_string) methods of the Parser() class:

```
# Import the email modules we'll need
from email.parser import Parser

#  If the e-mail headers are in a file, uncomment this line:
#headers = Parser().parse(open(messagefile, 'r'))

#  Or for parsing headers in a string, use:
headers = Parser().parsestr('From: <user@example.com>\n'
        'To: <someone_else@example.com>\n'
        'Subject: Test message\n'
        '\n'
        'Body would go here\n')

#  Now the header items can be accessed as a dictionary:
print 'To: %s' % headers['to']
print 'From: %s' % headers['from']
print 'Subject: %s' % headers['subject']
```

Here’s an example of how to send a MIME message containing a bunch of family pictures that may be residing in a directory:

```
# Import smtplib for the actual sending function
import smtplib

# Here are the email package modules we'll need
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart

COMMASPACE = ', '

# Create the container (outer) email message.
msg = MIMEMultipart()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = COMMASPACE.join(family)
msg.preamble = 'Our family reunion'

# Assume we know that the image files are all in PNG format
for file in pngfiles:
    # Open the files in binary mode.  Let the MIMEImage class automatically
    # guess the specific image type.
    fp = open(file, 'rb')
```

(下页继续)

(续上页)

```
    img = MIMEImage(fp.read())
    fp.close()
    msg.attach(img)

# Send the email via our own SMTP server.
s = smtplib.SMTP('localhost')
s.sendmail(me, family, msg.as_string())
s.quit()
```

Here's an example of how to send the entire contents of a directory as an email message:[1]

```
#!/usr/bin/env python

"""Send the contents of a directory as a MIME message."""

import os
import sys
import smtplib
# For guessing MIME type based on file name extension
import mimetypes

from optparse import OptionParser

from email import encoders
from email.message import Message
from email.mime.audio import MIMEAudio
from email.mime.base import MIMEBase
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

COMMASPACE = ', '


def main():
    parser = OptionParser(usage="""\
Send the contents of a directory as a MIME message.

Usage: %prog [options]

Unless the -o option is given, the email is sent by forwarding to your local
SMTP server, which then does the normal delivery process.  Your local machine
must be running an SMTP server.
""")
    parser.add_option('-d', '--directory',
                      type='string', action='store',
                      help="""Mail the contents of the specified directory,
                      otherwise use the current directory.  Only the regular
                      files in the directory are sent, and we don't recurse to
                      subdirectories.""")
    parser.add_option('-o', '--output',
                      type='string', action='store', metavar='FILE',
                      help="""Print the composed message to FILE instead of
                      sending the message to the SMTP server.""")
    parser.add_option('-s', '--sender',
```

(下页继续)

[1] Thanks to Matthew Dixon Cowles for the original inspiration and examples.

(续上页)

```
                    type='string', action='store', metavar='SENDER',
                    help='The value of the From: header (required)')
    parser.add_option('-r', '--recipient',
                    type='string', action='append', metavar='RECIPIENT',
                    default=[], dest='recipients',
                    help='A To: header value (at least one required)')
    opts, args = parser.parse_args()
    if not opts.sender or not opts.recipients:
        parser.print_help()
        sys.exit(1)
    directory = opts.directory
    if not directory:
        directory = '.'
    # Create the enclosing (outer) message
    outer = MIMEMultipart()
    outer['Subject'] = 'Contents of directory %s' % os.path.abspath(directory)
    outer['To'] = COMMASPACE.join(opts.recipients)
    outer['From'] = opts.sender
    outer.preamble = 'You will not see this in a MIME-aware mail reader.\n'

    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension.  Encoding
        # will be ignored, although we should check for simple things like
        # gzip'd or compressed files.
        ctype, encoding = mimetypes.guess_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (compressed), so
            # use a generic bag-of-bits type.
            ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        if maintype == 'text':
            fp = open(path)
            # Note: we should handle calculating the charset
            msg = MIMEText(fp.read(), _subtype=subtype)
            fp.close()
        elif maintype == 'image':
            fp = open(path, 'rb')
            msg = MIMEImage(fp.read(), _subtype=subtype)
            fp.close()
        elif maintype == 'audio':
            fp = open(path, 'rb')
            msg = MIMEAudio(fp.read(), _subtype=subtype)
            fp.close()
        else:
            fp = open(path, 'rb')
            msg = MIMEBase(maintype, subtype)
            msg.set_payload(fp.read())
            fp.close()
            # Encode the payload using Base64
            encoders.encode_base64(msg)
        # Set the filename parameter
        msg.add_header('Content-Disposition', 'attachment', filename=filename)
        outer.attach(msg)
```

(下页继续)

(续上页)

```
    # Now send or store the message
    composed = outer.as_string()
    if opts.output:
        fp = open(opts.output, 'w')
        fp.write(composed)
        fp.close()
    else:
        s = smtplib.SMTP('localhost')
        s.sendmail(opts.sender, opts.recipients, composed)
        s.quit()


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

Here’s an example of how to unpack a MIME message like the one above, into a directory of files:

```
#!/usr/bin/env python

"""Unpack a MIME message into a directory of files."""

import os
import sys
import email
import errno
import mimetypes

from optparse import OptionParser


def main():
    parser = OptionParser(usage="""\
Unpack a MIME message into a directory of files.

Usage: %prog [options] msgfile
""")
    parser.add_option('-d', '--directory',
                      type='string', action='store',
                      help="""Unpack the MIME message into the named
                      directory, which will be created if it doesn't already
                      exist.""")
    opts, args = parser.parse_args()
    if not opts.directory:
        parser.print_help()
        sys.exit(1)

    try:
        msgfile = args[0]
    except IndexError:
        parser.print_help()
        sys.exit(1)

    try:
        os.mkdir(opts.directory)
    except OSError as e:
        # Ignore directory exists error
```

(下页继续)

(续上页)

```
        if e.errno != errno.EEXIST:
            raise

    fp = open(msgfile)
    msg = email.message_from_file(fp)
    fp.close()

    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given filename so that an
        # email message can't be used to overwrite important files
        filename = part.get_filename()
        if not filename:
            ext = mimetypes.guess_extension(part.get_content_type())
            if not ext:
                # Use a generic bag-of-bits extension
                ext = '.bin'
            filename = 'part-%03d%s' % (counter, ext)
        counter += 1
        fp = open(os.path.join(opts.directory, filename), 'wb')
        fp.write(part.get_payload(decode=True))
        fp.close()


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

Here's an example of how to create an HTML message with an alternative plain text version:[2]

```
#!/usr/bin/env python

import smtplib

from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

# me == my email address
# you == recipient's email address
me = "my@email.com"
you = "your@email.com"

# Create message container - the correct MIME type is multipart/alternative.
msg = MIMEMultipart('alternative')
msg['Subject'] = "Link"
msg['From'] = me
msg['To'] = you

# Create the body of the message (a plain-text and an HTML version).
text = "Hi!\nHow are you?\nHere is the link you wanted:\nhttps://www.python.org"
html = """\
<html>
  <head></head>
```

(下页继续)

[2] Contributed by Martin Matejek.

(续上页)

```
  <body>
    <p>Hi!<br>
       How are you?<br>
       Here is the <a href="https://www.python.org">link</a> you wanted.
    </p>
  </body>
</html>
"""

# Record the MIME types of both parts - text/plain and text/html.
part1 = MIMEText(text, 'plain')
part2 = MIMEText(html, 'html')

# Attach parts into message container.
# According to RFC 2046, the last part of a multipart message, in this case
# the HTML message, is best and preferred.
msg.attach(part1)
msg.attach(part2)

# Send the message via local SMTP server.
s = smtplib.SMTP('localhost')
# sendmail function takes 3 arguments: sender's address, recipient's address
# and message to send - here it is sent as one string.
s.sendmail(me, you, msg.as_string())
s.quit()
```

参见:

Module *smtplib* SMTP protocol client

Module *nntplib* NNTP protocol client

18.1.12 Package History

This table describes the release history of the email package, corresponding to the version of Python that the package was released with. For purposes of this document, when you see a note about change or added versions, these refer to the Python version the change was made in, *not* the email package version. This table also describes the Python compatibility of each version of the package.

email version	distributed with	compatible with
`1.x`	Python 2.2.0 to Python 2.2.1	*no longer supported*
`2.5`	Python 2.2.2+ and Python 2.3	Python 2.1 to 2.5
`3.0`	Python 2.4	Python 2.3 to 2.5
`4.0`	Python 2.5	Python 2.3 to 2.5

Here are the major differences between *email* version 4 and version 3:

- All modules have been renamed according to **PEP 8** standards. For example, the version 3 module `email.Message` was renamed to *email.message* in version 4.
- A new subpackage *email.mime* was added and all the version 3 `email.MIME*` modules were renamed and situated into the *email.mime* subpackage. For example, the version 3 module `email.MIMEText` was renamed to `email.mime.text`.

  Note that the version 3 names will continue to work until Python 2.6.

- The email.mime.application module was added, which contains the *MIMEApplication* class.
- Methods that were deprecated in version 3 have been removed. These include Generator.__call__(), Message.get_type(), Message.get_main_type(), Message.get_subtype().
- Fixes have been added for **RFC 2231** support which can change some of the return types for *Message.get_param* and friends. Under some circumstances, values which used to return a 3-tuple now return simple strings (specifically, if all extended parameter segments were unencoded, there is no language and charset designation expected, so the return type is now a simple string). Also, %-decoding used to be done for both encoded and unencoded segments; this decoding is now done only for encoded segments.

Here are the major differences between *email* version 3 and version 2:

- The *FeedParser* class was introduced, and the *Parser* class was implemented in terms of the *FeedParser*. All parsing therefore is non-strict, and parsing will make a best effort never to raise an exception. Problems found while parsing messages are stored in the message's *defect* attribute.
- All aspects of the API which raised *DeprecationWarning*s in version 2 have been removed. These include the *_encoder* argument to the *MIMEText* constructor, the Message.add_payload() method, the Utils.dump_address_pair() function, and the functions Utils.decode() and Utils.encode().
- New *DeprecationWarning*s have been added to: Generator.__call__(), Message.get_type(), Message.get_main_type(), Message.get_subtype(), and the *strict* argument to the *Parser* class. These are expected to be removed in future versions.
- Support for Pythons earlier than 2.3 has been removed.

Here are the differences between *email* version 2 and version 1:

- The email.Header and email.Charset modules have been added.
- The pickle format for *Message* instances has changed. Since this was never (and still isn't) formally defined, this isn't considered a backward incompatibility. However if your application pickles and unpickles *Message* instances, be aware that in *email* version 2, *Message* instances now have private variables *_charset* and *_default_type*.
- Several methods in the *Message* class have been deprecated, or their signatures changed. Also, many new methods have been added. See the documentation for the *Message* class for details. The changes should be completely backward compatible.
- The object structure has changed in the face of *message/rfc822* content types. In *email* version 1, such a type would be represented by a scalar payload, i.e. the container message's *is_multipart()* returned false, *get_payload()* was not a list object, but a single *Message* instance.

  This structure was inconsistent with the rest of the package, so the object representation for *message/rfc822* content types was changed. In *email* version 2, the container *does* return True from *is_multipart()*, and *get_payload()* returns a list containing a single *Message* item.

  Note that this is one place that backward compatibility could not be completely maintained. However, if you're already testing the return type of *get_payload()*, you should be fine. You just need to make sure your code doesn't do a *set_payload()* with a *Message* instance on a container with a content type of *message/rfc822*.
- The *Parser* constructor's *strict* argument was added, and its *parse()* and *parsestr()* methods grew a *headersonly* argument. The *strict* flag was also added to functions *email.message_from_file()* and *email.message_from_string()*.
- Generator.__call__() is deprecated; use *Generator.flatten* instead. The *Generator* class has also grown the *clone()* method.
- The *DecodedGenerator* class in the *email.generator* module was added.

- The intermediate base classes *MIMENonMultipart* and *MIMEMultipart* have been added, and interposed in the class hierarchy for most of the other MIME-related derived classes.
- The *_encoder* argument to the *MIMEText* constructor has been deprecated. Encoding now happens implicitly based on the *_charset* argument.
- The following functions in the email.Utils module have been deprecated: dump_address_pairs(), decode(), and encode(). The following functions have been added to the module: make_msgid(), decode_rfc2231(), encode_rfc2231(), and decode_params().
- The non-public function email.Iterators._structure() was added.

18.1.13 Differences from mimelib

The *email* package was originally prototyped as a separate library called mimelib. Changes have been made so that method names are more consistent, and some methods or modules have either been added or removed. The semantics of some of the methods have also changed. For the most part, any functionality available in mimelib is still available in the *email* package, albeit often in a different way. Backward compatibility between the mimelib package and the *email* package was not a priority.

Here is a brief description of the differences between the mimelib and the *email* packages, along with hints on how to port your applications.

Of course, the most visible difference between the two packages is that the package name has been changed to *email*. In addition, the top-level package has the following differences:

- messageFromString() has been renamed to *message_from_string()*.
- messageFromFile() has been renamed to *message_from_file()*.

The *Message* class has the following differences:

- The method asString() was renamed to *as_string()*.
- The method ismultipart() was renamed to *is_multipart()*.
- The *get_payload()* method has grown a *decode* optional argument.
- The method getall() was renamed to *get_all()*.
- The method addheader() was renamed to *add_header()*.
- The method gettype() was renamed to get_type().
- The method getmaintype() was renamed to get_main_type().
- The method getsubtype() was renamed to get_subtype().
- The method getparams() was renamed to *get_params()*. Also, whereas getparams() returned a list of strings, *get_params()* returns a list of 2-tuples, effectively the key/value pairs of the parameters, split on the '=' sign.
- The method getparam() was renamed to *get_param()*.
- The method getcharsets() was renamed to *get_charsets()*.
- The method getfilename() was renamed to *get_filename()*.
- The method getboundary() was renamed to *get_boundary()*.
- The method setboundary() was renamed to *set_boundary()*.
- The method getdecodedpayload() was removed. To get similar functionality, pass the value 1 to the *decode* flag of the *get_payload()* method.

- The method `getpayloadastext()` was removed. Similar functionality is supported by the *DecodedGenerator* class in the *email.generator* module.
- The method `getbodyastext()` was removed. You can get similar functionality by creating an iterator with *typed_subpart_iterator()* in the *email.iterators* module.

The *Parser* class has no differences in its public interface. It does have some additional smarts to recognize *message/delivery-status* type messages, which it represents as a *Message* instance containing separate *Message* subparts for each header block in the delivery status notification[1].

The *Generator* class has no differences in its public interface. There is a new class in the *email.generator* module though, called *DecodedGenerator* which provides most of the functionality previously available in the `Message.getpayloadastext()` method.

The following modules and classes have been changed:

- The *MIMEBase* class constructor arguments *_major* and *_minor* have changed to *_maintype* and *_subtype* respectively.
- The `Image` class/module has been renamed to `MIMEImage`. The *_minor* argument has been renamed to *_subtype*.
- The `Text` class/module has been renamed to `MIMEText`. The *_minor* argument has been renamed to *_subtype*.
- The `MessageRFC822` class/module has been renamed to `MIMEMessage`. Note that an earlier version of `mimelib` called this class/module `RFC822`, but that clashed with the Python standard library module *rfc822* on some case-insensitive file systems.

  Also, the *MIMEMessage* class now represents any kind of MIME message with main type *message*. It takes an optional argument *_subtype* which is used to set the MIME subtype. *_subtype* defaults to *rfc822*.

`mimelib` provided some utility functions in its `address` and `date` modules. All of these functions have been moved to the *email.utils* module.

The `MsgReader` class/module has been removed. Its functionality is most closely supported in the *body_line_iterator()* function in the *email.iterators* module.

18.2 `json` —JSON 编码和解码器

2.6 新版功能.

JSON (JavaScript Object Notation)，由 **RFC 7159** (which obsoletes **RFC 4627**) 和 ECMA-404 指定，是一个受 JavaScript 的对象字面量语法启发的轻量级数据交换格式，尽管它不仅仅是一个严格意义上的 JavaScript 的字集[1]。

json 提供了与标准库*marshal* 和*pickle* 相似的 API 接口。

对基本的 Python 对象层次结构进行编码：

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print json.dumps("\"foo\bar")
"\"foo\bar"
>>> print json.dumps(u'\u1234')
"\u1234"
>>> print json.dumps('\\')
```

(下页继续)

[1] Delivery Status Notifications (DSN) are defined in **RFC 1894**.

[1] 正如 RFC 7159 的勘误表 所说明的，JSON 允许以字符串表示字面值字符 U+2028 (LINE SEPARATOR) 和 U+2029 (PARAGRAPH SEPARATOR)，而 JavaScript (在 ECMAScript 5.1 版中) 不允许。

(续上页)

```
"\\"
>>> print json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True)
{"a": 0, "b": 0, "c": 0}
>>> from StringIO import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

紧凑编码：

```
>>> import json
>>> json.dumps([1,2,3,{'4': 5, '6': 7}], separators=(',',':'))
'[1,2,3,{"4":5,"6":7}]'
```

美化输出：

```
>>> import json
>>> print json.dumps({'4': 5, '6': 7}, sort_keys=True,
...                   indent=4, separators=(',', ': '))
{
    "4": 5,
    "6": 7
}
```

JSON 解码：

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
[u'foo', {u'bar': [u'baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
u'"foo\x08ar'
>>> from StringIO import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
[u'streaming API']
```

特殊 JSON 对象解码：

```
>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')
```

扩展*JSONEncoder*：

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
```

(下页继续)

(续上页)

```
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the TypeError
...         return json.JSONEncoder.default(self, obj)
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[', '2.0', ', ', '1.0', ']']
```

Using `json.tool` from the shell to validate and pretty-print:

```
$ echo '{"json":"obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -mjson.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

注解： JSON 是 YAML 1.2 的一个子集。由该模块的默认设置生成的 JSON （尤其是默认的“分隔符”设置值）也是 YAML 1.0 and 1.1 的一个子集。因此该模块也能够用于序列化为 YAML。

18.2.1 基本使用

`json.`**`dump`**(*obj*, *fp*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *encoding="utf-8"*, *default=None*, *sort_keys=False*, ***kw*)

使用这个转换表 将 *obj* 序列化为 JSON 格式化流形式的 *fp* (支持 `.write()` 的*file-like object*)。

If *skipkeys* is true (default: `False`), then dict keys that are not of a basic type (*str*, *unicode*, *int*, *long*, *float*, *bool*, `None`) will be skipped instead of raising a *TypeError*.

If *ensure_ascii* is true (the default), all non-ASCII characters in the output are escaped with `\uXXXX` sequences, and the result is a *str* instance consisting of ASCII characters only. If *ensure_ascii* is false, some chunks written to *fp* may be *unicode* instances. This usually happens because the input contains unicode strings or the *encoding* parameter is used. Unless `fp.write()` explicitly understands *unicode* (as in *codecs.getwriter()*) this is likely to cause an error.

如果 *check_circular* 是为假值 (默认为 `True`)，那么容器类型的循环引用检验会被跳过并且循环引用会引发一个*OverflowError* (或者更糟的情况)。

如果 *allow_nan* 是 false（默认为 `True`），那么在对严格 JSON 规格范围外的*float* 类型值（`nan`、`inf` 和 `-inf`）进行序列化时会引发一个*ValueError*。如果 *allow_nan* 是 true，则使用它们的 JavaScript 等价形式（`NaN`、`Infinity` 和 `-Infinity`）。

If *indent* is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0, or negative, will only insert newlines. `None` (the default) selects the most compact representation.

注解： Since the default item separator is `', '`, the output might include trailing whitespace when *indent* is specified. You can use `separators=(',', ': ')` to avoid this.

If specified, *separators* should be an `(item_separator, key_separator)` tuple. By default, `(', ', ': ')` are used. To get the most compact JSON representation, you should specify `(',', ':')` to eliminate whitespace.

encoding is the character encoding for str instances, default is UTF-8.

当 *default* 被指定时，其应该是一个函数，每当某个对象无法被序列化时它会被调用。它应该返回该对象的一个可以被 JSON 编码的版本或者引发一个*TypeError*。如果没有被指定，则会直接引发*TypeError*。

如果 *sort_keys* 是 true（默认为 `False`），那么字典的输出会以键的顺序排序。

为了使用一个自定义的*JSONEncoder* 子类（比如：覆盖了 `default()` 方法来序列化额外的类型），通过 *cls* 关键字参数来指定；否则将使用*JSONEncoder*。

注解： Unlike *pickle* and *marshal*, JSON is not a framed protocol so trying to serialize more objects with repeated calls to *dump()* and the same *fp* will result in an invalid JSON file.

---

json.**dumps**(*obj*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *encoding="utf-8"*, *default=None*, *sort_keys=False*, ***kw*)

Serialize *obj* to a JSON formatted *str* using this *conversion table*. If *ensure_ascii* is false, the result may contain non-ASCII characters and the return value may be a *unicode* instance.

The arguments have the same meaning as in *dump()*.

注解： JSON 中的键-值对中的键永远是*str* 类型的。当一个对象被转化为 JSON 时，字典中所有的键都会被强制转换为字符串。这所造成的结果是字典被转换为 JSON 然后转换回字典时可能和原来的不相等。换句话说，如果 x 具有非字符串的键，则有 `loads(dumps(x)) != x`。

---

json.**load**(*fp*[, *encoding*[, *cls*[, *object_hook*[, *parse_float*[, *parse_int*[, *parse_constant*[, *object_pairs_hook*[, ***kw*]]]]]]]])

Deserialize *fp* (a `.read()`-supporting *file-like object* containing a JSON document) to a Python object using this *conversion table*.

If the contents of *fp* are encoded with an ASCII based encoding other than UTF-8 (e.g. latin-1), then an appropriate *encoding* name must be specified. Encodings that are not ASCII based (such as UCS-2) are not allowed, and should be wrapped with `codecs.getreader(encoding)(fp)`, or simply decoded to a *unicode* object and passed to *loads()*.

object_hook 是一个可选的函数，它会被调用于每一个解码出的对象字面量（即一个*dict*）。*object_hook* 的返回值会取代原本的*dict*。这一特性能够被用于实现自定义解码器（如 JSON-RPC 的类型提示）。

object_pairs_hook is an optional function that will be called with the result of any object literal decoded with an ordered list of pairs. The return value of *object_pairs_hook* will be used instead of the *dict*. This feature can be used to implement custom decoders that rely on the order that the key and value pairs are decoded (for example, *collections.OrderedDict()* will remember the order of insertion). If *object_hook* is also defined, the *object_pairs_hook* takes priority.

在 2.7 版更改: 添加了对 *object_pairs_hook* 的支持。

parse_float , 如果指定, 将与每个要解码 JSON 浮点数一同调用。默认状态下, 相当于 `float(num_str)` 。可以用于对 JSON 浮点数使用其它数据类型和解析器（比如*decimal.Decimal* ）。

parse_int ， 如果指定，将与每个要解码 JSON 整数的字符串一同调用。默认状态下，相当于 `int(num_str)` 。可以用于对 JSON 整数使用其它数据类型和语法分析程序（比如*float* ）。

parse_constant ， 如果指定，将要与以下字符串中的一个一同调用：`'-Infinity'` ， `'Infinity'` ， `'NaN'` 。如果遇到无效的 JSON 数字它会被用于抛出异常。

在 2.7 版更改: *parse_constant* 不再调用 'null', 'true', 'false'。

要使用自定义的`JSONDecoder`子类，用 `cls` 指定他；否则使用`JSONDecoder`。额外的关键词参数会通过类的构造函数传递。

`json.loads`(*s*[, *encoding*[, *cls*[, *object_hook*[, *parse_float*[, *parse_int*[, *parse_constant*[, *object_pairs_hook*[, ***kw*]]]]]]]])

Deserialize *s* (a `str` or `unicode` instance containing a JSON document) to a Python object using this *conversion table*.

If *s* is a `str` instance and is encoded with an ASCII based encoding other than UTF-8 (e.g. latin-1), then an appropriate *encoding* name must be specified. Encodings that are not ASCII based (such as UCS-2) are not allowed and should be decoded to `unicode` first.

The other arguments have the same meaning as in `load()`.

18.2.2 编码器和解码器

class `json.JSONDecoder`([*encoding*[, *object_hook*[, *parse_float*[, *parse_int*[, *parse_constant*[, *strict*[, *object_pairs_hook*]]]]]]])

简单的 JSON 解码器。

默认情况下，解码执行以下翻译:

JSON	Python
object	dict
array	list
string	unicode
number (int)	int, long
number (real)	float
true	True
false	False
null	None

它还将 "NaN"、"Infinity" 和 "-Infinity" 理解为它们对应的 "float" 值，这超出了 JSON 规范。

encoding determines the encoding used to interpret any `str` objects decoded by this instance (UTF-8 by default). It has no effect when decoding `unicode` objects.

Note that currently only encodings that are a superset of ASCII work, strings of other encodings should be passed in as `unicode`.

object_hook，如果指定，会被每个解码的 JSON 对象的结果调用，并且返回值会替代给定`dict`。它可被用于提供自定义反序列化（比如去支持 JSON-RPC 类的暗示）。

object_pairs_hook, if specified will be called with the result of every JSON object decoded with an ordered list of pairs. The return value of *object_pairs_hook* will be used instead of the `dict`. This feature can be used to implement custom decoders that rely on the order that the key and value pairs are decoded (for example, `collections.OrderedDict()` will remember the order of insertion). If *object_hook* is also defined, the *object_pairs_hook* takes priority.

在 2.7 版更改: 添加了对 *object_pairs_hook* 的支持。

parse_float, 如果指定, 将与每个要解码 JSON 浮点数一同调用。默认状态下, 相当于 `float(num_str)`。可以用于对 JSON 浮点数使用其它数据类型和解析器（比如`decimal.Decimal`）。

parse_int，如果指定，将与每个要解码 JSON 整数的字符串一同调用。默认状态下，相当于 `int(num_str)`。可以用于对 JSON 整数使用其它数据类型和语法分析程序（比如`float`）。

parse_constant，如果指定，将要与以下字符串中的一个一同调用：`'-Infinity'`，`'Infinity'`，`'NaN'`。如果遇到无效的 JSON 数字它会被用于抛出异常。

如果 *strict* 为 false （默认为 `True`），那么控制字符将被允许在字符串内。在此上下文中的控制字符是那些编码于范围 0–31 的字符，包括 `'\t'`(制表符)，`'\n'` 和 `'\0'`。

If the data being deserialized is not a valid JSON document, a *ValueError* will be raised.

decode(*s*)
Return the Python representation of *s* (a *str* or *unicode* instance containing a JSON document).

raw_decode(*s*)
Decode a JSON document from *s* (a *str* or *unicode* beginning with a JSON document) and return a 2-tuple of the Python representation and the index in *s* where the document ended.

这可以用于从一个字符串解码 JSON 文档，该字符串的末尾可能有无关的数据。

class json.**JSONEncoder**([*skipkeys*[, *ensure_ascii*[, *check_circular*[, *allow_nan*[, *sort_keys*[, *indent*[, *separators*[, *encoding*[, *default*]]]]]]]]])
用于 Python 数据结构的可扩展 JSON 编码器。

默认支持以下对象和类型：

Python	JSON
dict	object
list, tuple	array
str, unicode	string
int, long, float	number
True	true
False	false
None	null

为了将其拓展至识别其他对象，需要子类化并实现*default()* 方法于另一种返回 o 的可序列化对象的方法如果可行，否则它应该调用超类实现（来引发*TypeError*）。

If *skipkeys* is false (the default), then it is a *TypeError* to attempt encoding of keys that are not str, int, long, float or `None`. If *skipkeys* is true, such items are simply skipped.

If *ensure_ascii* is true (the default), all non-ASCII characters in the output are escaped with `\uXXXX` sequences, and the results are *str* instances consisting of ASCII characters only. If *ensure_ascii* is false, a result may be a *unicode* instance. This usually happens if the input contains unicode strings or the *encoding* parameter is used.

如果 *check_circular* 为 true （默认），那么列表，字典，和自定义编码的对象在编码期间会被检查重复循环引用防止无限递归（无限递归将导致*OverflowError*）。否则，这样进行检查。

如果 *allow_nan* 为 true （默认），那么 `NaN`，`Infinity`，和 `-Infinity` 进行编码。此行为不符合 JSON 规范，但与大多数的基于 Javascript 的编码器和解码器一致。否则，它将是一个*ValueError* 来编码这些浮点数。

如果 *sort_keys* 为 true （默认为：`False`），那么字典的输出是按照键排序；这对回归测试很有用，以确保可以每天比较 JSON 序列化。

If *indent* is a non-negative integer (it is `None` by default), then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. `None` is the most compact representation.

注解： Since the default item separator is `', '`, the output might include trailing whitespace when *indent* is specified. You can use `separators=(',', ': ')` to avoid this.

If specified, *separators* should be an `(item_separator, key_separator)` tuple. By default, `(', ', ': ')` are used. To get the most compact JSON representation, you should specify `(',', ':')` to eliminate whitespace.

当 *default* 被指定时，其应该是一个函数，每当某个对象无法被序列化时它会被调用。它应该返回该对象的一个可以被 JSON 编码的版本或者引发一个*TypeError*。如果没有被指定，则会直接引发*TypeError*。

If *encoding* is not `None`, then all input strings will be transformed into unicode using that encoding prior to JSON-encoding. The default is UTF-8.

default(*o*)
在子类中实现这种方法使其返回 *o* 的可序列化对象，或者调用基础实现（引发*TypeError* ）

比如说，为了支持任意迭代器，你可以像这样实现默认设置:

```
def default(self, o):
   try:
       iterable = iter(o)
   except TypeError:
       pass
   else:
       return list(iterable)
   # Let the base class default method raise the TypeError
   return JSONEncoder.default(self, o)
```

encode(*o*)
返回 Python *o* 数据结构的 JSON 字符串表达方式。比如说:

```
>>> JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*)
编码给定对象 *o* ，并且让每个可用的字符串表达方式。例如:

```
for chunk in JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

18.2.3 标准符合性和互操作性

JSON 格式由 **RFC 7159** 和 ECMA-404 指定。此段落详细讲了这个模块符合 RFC 的级别。简单来说，*JSONEncoder* 和*JSONDecoder* 子类，和明确提到的参数以外的参数，不作考虑。

此模块不严格遵循于 RFC ，它实现了一些扩展是有效的 Javascript 但不是有效的 JSON。尤其是:

- 无限和 NaN 数值是可接受的并输出;
- 对象内的重复名称是接受的，并且仅使用最后一对名称-值对。

自从 RFC 允许符合 RFC 的语法分析程序接收不符合 RFC 的输入文本以来，这个模块的解串器在默认状态下默认符合 RFC 。

字符编码

The RFC requires that JSON be represented using either UTF-8, UTF-16, or UTF-32, with UTF-8 being the recommended default for maximum interoperability. Accordingly, this module uses UTF-8 as the default for its *encoding* parameter.

This module's deserializer only directly works with ASCII-compatible encodings; UTF-16, UTF-32, and other ASCII-incompatible encodings require the use of workarounds described in the documentation for the deserializer's *encoding* parameter.

RFC 允许，尽管不是必须的，这个模块的序列化默认设置为 *ensure_ascii=True* ，这样消除输出以便结果字符串至容纳 ASCII 字符。

RFC 禁止添加字符顺序标记（BOM ）在 JSON 文本的开头，这个模块的序列化器不添加 BOM 标记在它的输出上。RFC，准许 JSON 反序列化器忽略它们输入中的初始 BOM 标记，但不要求。此模块的反序列化器引发*ValueError* 当出现初始 BOM 标记。

RFC 不会明确禁止包含字节序列的 JSON 字符串这不对应有效的 Unicode 字符（比如不成对的 UTF-16 的替代物），但是它确实指出它们可能会导致互操作性问题。默认下，模块对这样的序列接受和输出（当在原始*str* 存在时）代码点。

Infinite 和 NaN 数值。

RFC 不允许 infinite 或者 NaN 数值的表达方式。尽管这样，默认情况下，此模块接受并且输出 `Infinity` ，`-Infinity`，和 `NaN` 好像它们是有效的 JSON 数字字面值

```
>>> # Neither of these calls raises an exception, but the results are not valid JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

序列化器中，*allow_nan* 参数可用于替代这个行为。反序列化器中，*parse_constant* 参数，可用于替代这个行为。

对象中的重复名称

RFC 具体说明了在 JSON 对象里的名字应该是唯一的，但没有规定如何处理 JSON 对象中的重复名称。默认下，此模块不引发异常；作为替代，对于给定名它将忽略除姓-值对之外的所有对:

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{u'x': 3}
```

The *object_pairs_hook* parameter can be used to alter this behavior.

顶级非对象，非数组值

过时的 RFC 4627 指定的旧版本 JSON 要求 JSON 文本顶级值必须是 JSON 对象或数组（Python *dict* 或 `list`），并且不能是 JSON null 值，布尔值，数值或者字符串值。RFC 7159 移除这个限制，此模块没有并且从未在序列化器和反序列化器中实现这个限制。

无论如何，为了最大化地获取互操作性，你可能希望自己遵守该原则。

实现限制

一些 JSON 反序列化器的实现应该在以下方面做出限制：

- 接受的 JSON 文本大小
- 嵌套 JSON 对象和数组的最高水平
- JSON 数字的范围和精度
- JSON 字符串的内容和最大长度

此模块不强制执行任何上述限制，除了相关的 Python 数据类型本身或者 Python 解释器本身的限制以外。

当序列化为 JSON ，在应用中当心此类限制这可能破坏你的 JSON 。特别是，通常将 JSON 数字反序列化为 IEEE 754 双精度数字，从而受到该表示形式的范围和精度限制。这是特别相关的，当序列化非常大的 Python *int* 值时，或者当序列化“exotic”数值类型的实例时比如*decimal.Decimal* 。

备注

18.3 `mailcap` —Mailcap 文件处理

源代码: Lib/mailcap.py

Mailcap 文件可用来配置支持 MIME 的应用例如邮件阅读器和 Web 浏览器如何响应具有不同 MIME 类型的文件。(” mailcap” 这个名称源自短语” mail capability”。）例如，一个 mailcap 文件可能包含 `video/mpeg; xmpeg %s` 这样的行。然后，如果用户遇到 MIME 类型为 *video/mpeg* 的邮件消息或 Web 文档时，`%s` 将被替换为一个文件名（通常是一个临时文件）并且将自动启动 **xmpeg** 程序来查看该文件。

The mailcap format is documented in RFC 1524, “A User Agent Configuration Mechanism For Multimedia Mail Format Information,” but is not an Internet standard. However, mailcap files are supported on most Unix systems.

`mailcap.`**`findmatch`**(*caps*, *MIMEtype*[, *key*[, *filename*[, *plist*]]])

返回一个 2 元组；其中第一个元素是包含所要执行命令的字符串 (它可被传递给*os.system()*)，第二个元素是对应于给定 MIME 类型的 mailcap 条目。如果找不到匹配的 MIME 类型，则将返回 `(None, None)`。

key 是所需字段的名称，它代表要执行的活动类型；默认值是‘view’，因为在最通常的情况下你只是想要查看 MIME 类型数据的正文。其他可能的值还有‘compose’和‘edit’，分别用于想要创建给定 MIME 类型正文或修改现有正文数据的情况。请参阅 RFC 1524 获取这些字段的完整列表。

filename 是在命令行中用来替换 `%s` 的文件名；默认值 `'/dev/null'` 几乎肯定不是你想要的，因此通常你要通过指定一个文件名来重载它。

plist 可以是一个包含命名形参的列表；默认值只是一个空列表。列表中的每个条目必须为包含形参名称的字符串、等于号 (`'='`) 以及形参的值。Mailcap 条目可以包含形如 `%{foo}` 的命名形参，它将由名为‘foo’的形参的值所替换。例如，如果命令行 `showpartial %{id} %{number} %{total}` 是

在一个 mailcap 文件中，并且 *plist* 被设为 `['id=1', 'number=2', 'total=3']`，则结果命令行将为 `'showpartial 1 2 3'`。

在 mailcap 文件中，可以指定可选的“test”字段来检测某些外部条件（例如所使用的机器架构或窗口系统）来确定是否要应用 mailcap 行。*findmatch()* 将自动检查此类条件并在检查未通过时跳过条目。

mailcap.**getcaps**()
返回一个将 MIME 类型映射到 mailcap 文件条目列表的字典。此字典必须被传给*findmatch()* 函数。条目会被存储为字典列表，但并不需要了解此表示形式的细节。

此信息来自在系统中找到的所有 mailcap 文件。用户的 mailcap 文件 `$HOME/.mailcap` 中的设置将覆盖系统 mailcap 文件 `/etc/mailcap`, `/usr/etc/mailcap` 和 `/usr/local/etc/mailcap` 中的设置。

一个用法示例:

```
>>> import mailcap
>>> d = mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='tmp1223')
('xmpeg tmp1223', {'view': 'xmpeg %s'})
```

18.4 mailbox —Manipulate mailboxes in various formats

This module defines two classes, *Mailbox* and *Message*, for accessing and manipulating on-disk mailboxes and the messages they contain. *Mailbox* offers a dictionary-like mapping from keys to messages. *Message* extends the *email.message* module's *Message* class with format-specific state and behavior. Supported mailbox formats are Maildir, mbox, MH, Babyl, and MMDF.

参见:

模块*email* Represent and manipulate messages.

18.4.1 Mailbox 对象

class mailbox.**Mailbox**
A mailbox, which may be inspected and modified.

The *Mailbox* class defines an interface and is not intended to be instantiated. Instead, format-specific subclasses should inherit from *Mailbox* and your code should instantiate a particular subclass.

The *Mailbox* interface is dictionary-like, with small keys corresponding to messages. Keys are issued by the *Mailbox* instance with which they will be used and are only meaningful to that *Mailbox* instance. A key continues to identify a message even if the corresponding message is modified, such as by replacing it with another message.

Messages may be added to a *Mailbox* instance using the set-like method *add()* and removed using a `del` statement or the set-like methods *remove()* and *discard()*.

Mailbox interface semantics differ from dictionary semantics in some noteworthy ways. Each time a message is requested, a new representation (typically a *Message* instance) is generated based upon the current state of the mailbox. Similarly, when a message is added to a *Mailbox* instance, the provided message representation's contents are copied. In neither case is a reference to the message representation kept by the *Mailbox* instance.

The default *Mailbox* iterator iterates over message representations, not keys as the default dictionary iterator does. Moreover, modification of a mailbox during iteration is safe and well-defined. Messages added to the mailbox after an iterator is created will not be seen by the iterator. Messages removed from the mailbox before the iterator yields

them will be silently skipped, though using a key from an iterator may result in a *KeyError* exception if the corresponding message is subsequently removed.

> **警告:** Be very cautious when modifying mailboxes that might be simultaneously changed by some other process. The safest mailbox format to use for such tasks is Maildir; try to avoid using single-file formats such as mbox for concurrent writing. If you're modifying a mailbox, you *must* lock it by calling the *lock()* and *unlock()* methods *before* reading any messages in the file or making any changes by adding or deleting a message. Failing to lock the mailbox runs the risk of losing messages or corrupting the entire mailbox.

Mailbox instances have the following methods:

add(*message*)
Add *message* to the mailbox and return the key that has been assigned to it.

Parameter *message* may be a *Message* instance, an *email.message.Message* instance, a string, or a file-like object (which should be open in text mode). If *message* is an instance of the appropriate format-specific *Message* subclass (e.g., if it's an *mboxMessage* instance and this is an *mbox* instance), its format-specific information is used. Otherwise, reasonable defaults for format-specific information are used.

remove(*key*)
__delitem__(*key*)
discard(*key*)
Delete the message corresponding to *key* from the mailbox.

If no such message exists, a *KeyError* exception is raised if the method was called as *remove()* or *__delitem__()* but no exception is raised if the method was called as *discard()*. The behavior of *discard()* may be preferred if the underlying mailbox format supports concurrent modification by other processes.

__setitem__(*key*, *message*)
Replace the message corresponding to *key* with *message*. Raise a *KeyError* exception if no message already corresponds to *key*.

As with *add()*, parameter *message* may be a *Message* instance, an *email.message.Message* instance, a string, or a file-like object (which should be open in text mode). If *message* is an instance of the appropriate format-specific *Message* subclass (e.g., if it's an *mboxMessage* instance and this is an *mbox* instance), its format-specific information is used. Otherwise, the format-specific information of the message that currently corresponds to *key* is left unchanged.

iterkeys()
keys()
Return an iterator over all keys if called as *iterkeys()* or return a list of keys if called as *keys()*.

itervalues()
__iter__()
values()
Return an iterator over representations of all messages if called as *itervalues()* or *__iter__()* or return a list of such representations if called as *values()*. The messages are represented as instances of the appropriate format-specific *Message* subclass unless a custom message factory was specified when the *Mailbox* instance was initialized.

> **注解:** The behavior of *__iter__()* is unlike that of dictionaries, which iterate over keys.

iteritems()

items()
Return an iterator over (*key*, *message*) pairs, where *key* is a key and *message* is a message representation, if called as `iteritems()` or return a list of such pairs if called as `items()`. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

get(*key*, *default=None*)
__getitem__(*key*)
Return a representation of the message corresponding to *key*. If no such message exists, *default* is returned if the method was called as `get()` and a `KeyError` exception is raised if the method was called as `__getitem__()`. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

get_message(*key*)
Return a representation of the message corresponding to *key* as an instance of the appropriate format-specific `Message` subclass, or raise a `KeyError` exception if no such message exists.

get_string(*key*)
Return a string representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists.

get_file(*key*)
Return a file-like representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists. The file-like object behaves as if open in binary mode. This file should be closed once it is no longer needed.

注解: Unlike other representations of messages, file-like representations are not necessarily independent of the `Mailbox` instance that created them or of the underlying mailbox. More specific documentation is provided by each subclass.

---

has_key(*key*)
__contains__(*key*)
Return `True` if *key* corresponds to a message, `False` otherwise.

__len__()
Return a count of messages in the mailbox.

clear()
Delete all messages from the mailbox.

pop(*key*[, *default*])
Return a representation of the message corresponding to *key* and delete the message. If no such message exists, return *default* if it was supplied or else raise a `KeyError` exception. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

popitem()
Return an arbitrary (*key*, *message*) pair, where *key* is a key and *message* is a message representation, and delete the corresponding message. If the mailbox is empty, raise a `KeyError` exception. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

update(*arg*)
Parameter *arg* should be a *key*-to-*message* mapping or an iterable of (*key*, *message*) pairs. Updates the mailbox so that, for each given *key* and *message*, the message corresponding to *key* is set to *message* as if by using `__setitem__()`. As with `__setitem__()`, each *key* must already correspond to a message in the

mailbox or else a *KeyError* exception will be raised, so in general it is incorrect for *arg* to be a *Mailbox* instance.

注解: Unlike with dictionaries, keyword arguments are not supported.

flush()
Write any pending changes to the filesystem. For some *Mailbox* subclasses, changes are always written immediately and *flush()* does nothing, but you should still make a habit of calling this method.

lock()
Acquire an exclusive advisory lock on the mailbox so that other processes know not to modify it. An *ExternalClashError* is raised if the lock is not available. The particular locking mechanisms used depend upon the mailbox format. You should *always* lock the mailbox before making any modifications to its contents.

unlock()
Release the lock on the mailbox, if any.

close()
Flush the mailbox, unlock it if necessary, and close any open files. For some *Mailbox* subclasses, this method does nothing.

Maildir

class mailbox.**Maildir**(*dirname*, *factory=rfc822.Message*, *create=True*)
A subclass of *Mailbox* for mailboxes in Maildir format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *MaildirMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

It is for historical reasons that *factory* defaults to *rfc822.Message* and that *dirname* is named as such rather than *path*. For a *Maildir* instance that behaves like instances of other *Mailbox* subclasses, set *factory* to `None`.

Maildir is a directory-based mailbox format invented for the qmail mail transfer agent and now widely supported by other programs. Messages in a Maildir mailbox are stored in separate files within a common directory structure. This design allows Maildir mailboxes to be accessed and modified by multiple unrelated programs without data corruption, so file locking is unnecessary.

Maildir mailboxes contain three subdirectories, namely: `tmp`, `new`, and `cur`. Messages are created momentarily in the `tmp` subdirectory and then moved to the `new` subdirectory to finalize delivery. A mail user agent may subsequently move the message to the `cur` subdirectory and store information about the state of the message in a special "info" section appended to its file name.

Folders of the style introduced by the Courier mail transfer agent are also supported. Any subdirectory of the main mailbox is considered a folder if `'.'` is the first character in its name. Folder names are represented by *Maildir* without the leading `'.'`. Each folder is itself a Maildir mailbox but should not contain other folders. Instead, a logical nesting is indicated using `'.'` to delimit levels, e.g., "Archived.2005.07".

注解: The Maildir specification requires the use of a colon (`':'`) in certain message file names. However, some operating systems do not permit this character in file names, If you wish to use a Maildir-like format on such an operating system, you should specify another character to use instead. The exclamation point (`'!'`) is a popular choice. For example:

```
import mailbox
mailbox.Maildir.colon = '!'
```

The `colon` attribute may also be set on a per-instance basis.

Maildir instances have all of the methods of *Mailbox* in addition to the following:

list_folders()
Return a list of the names of all folders.

get_folder(*folder*)
Return a *Maildir* instance representing the folder whose name is *folder*. A *NoSuchMailboxError* exception is raised if the folder does not exist.

add_folder(*folder*)
Create a folder whose name is *folder* and return a *Maildir* instance representing it.

remove_folder(*folder*)
Delete the folder whose name is *folder*. If the folder contains any messages, a *NotEmptyError* exception will be raised and the folder will not be deleted.

clean()
Delete temporary files from the mailbox that have not been accessed in the last 36 hours. The Maildir specification says that mail-reading programs should do this occasionally.

Some *Mailbox* methods implemented by *Maildir* deserve special remarks:

add(*message*)
__setitem__(*key*, *message*)
update(*arg*)

> **警告:** These methods generate unique file names based upon the current process ID. When using multiple threads, undetected name clashes may occur and cause corruption of the mailbox unless threads are coordinated to avoid using these methods to manipulate the same mailbox simultaneously.

flush()
All changes to Maildir mailboxes are immediately applied, so this method does nothing.

lock()
unlock()
Maildir mailboxes do not support (or require) locking, so these methods do nothing.

close()
Maildir instances do not keep any open files and the underlying mailboxes do not support locking, so this method does nothing.

get_file(*key*)
Depending upon the host platform, it may not be possible to modify or remove the underlying message while the returned file remains open.

参见:

maildir man page from qmail The original specification of the format.

Using maildir format Notes on Maildir by its inventor. Includes an updated name-creation scheme and details on "info" semantics.

maildir man page from Courier Another specification of the format. Describes a common extension for supporting folders.

mbox

class mailbox.**mbox**(*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in mbox format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *mboxMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

The mbox format is the classic format for storing mail on Unix systems. All messages in an mbox mailbox are stored in a single file with the beginning of each message indicated by a line whose first five characters are "From ".

Several variations of the mbox format exist to address perceived shortcomings in the original. In the interest of compatibility, *mbox* implements the original format, which is sometimes referred to as *mboxo*. This means that the *Content-Length* header, if present, is ignored and that any occurrences of "From " at the beginning of a line in a message body are transformed to ">From " when storing the message, although occurrences of ">From " are not transformed to "From " when reading the message.

Some *Mailbox* methods implemented by *mbox* deserve special remarks:

get_file(*key*)

Using the file after calling `flush()` or `close()` on the *mbox* instance may yield unpredictable results or raise an exception.

lock()
unlock()

Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

参见:

mbox man page from qmail A specification of the format and its variations.

mbox man page from tin Another specification of the format, with details on locking.

Configuring Netscape Mail on Unix: Why The Content-Length Format is Bad An argument for using the original mbox format rather than a variation.

"mbox" is a family of several mutually incompatible mailbox formats A history of mbox variations.

MH

class mailbox.**MH**(*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in MH format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *MHMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

MH is a directory-based mailbox format invented for the MH Message Handling System, a mail user agent. Each message in an MH mailbox resides in its own file. An MH mailbox may contain other MH mailboxes (called *folders*) in addition to messages. Folders may be nested indefinitely. MH mailboxes also support *sequences*, which are named lists used to logically group messages without moving them to sub-folders. Sequences are defined in a file called `.mh_sequences` in each folder.

The *MH* class manipulates MH mailboxes, but it does not attempt to emulate all of **mh**'s behaviors. In particular, it does not modify and is not affected by the `context` or `.mh_profile` files that are used by **mh** to store its state and configuration.

MH instances have all of the methods of *Mailbox* in addition to the following:

list_folders()
Return a list of the names of all folders.

get_folder(*folder*)
Return an *MH* instance representing the folder whose name is *folder*. A *NoSuchMailboxError* exception is raised if the folder does not exist.

add_folder(*folder*)
Create a folder whose name is *folder* and return an *MH* instance representing it.

remove_folder(*folder*)
Delete the folder whose name is *folder*. If the folder contains any messages, a *NotEmptyError* exception will be raised and the folder will not be deleted.

get_sequences()
Return a dictionary of sequence names mapped to key lists. If there are no sequences, the empty dictionary is returned.

set_sequences(*sequences*)
Re-define the sequences that exist in the mailbox based upon *sequences*, a dictionary of names mapped to key lists, like returned by *get_sequences()*.

pack()
Rename messages in the mailbox as necessary to eliminate gaps in numbering. Entries in the sequences list are updated correspondingly.

注解: Already-issued keys are invalidated by this operation and should not be subsequently used.

Some *Mailbox* methods implemented by *MH* deserve special remarks:

remove(*key*)
__delitem__(*key*)
discard(*key*)
These methods immediately delete the message. The MH convention of marking a message for deletion by prepending a comma to its name is not used.

lock()
unlock()
Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls. For MH mailboxes, locking the mailbox means locking the `.mh_sequences` file and, only for the duration of any operations that affect them, locking individual message files.

get_file(*key*)
Depending upon the host platform, it may not be possible to remove the underlying message while the returned file remains open.

flush()
All changes to MH mailboxes are immediately applied, so this method does nothing.

close()
MH instances do not keep any open files, so this method is equivalent to *unlock()*.

参见:

nmh - Message Handling System Home page of **nmh**, an updated version of the original **mh**.

MH & nmh: Email for Users & Programmers A GPL-licensed book on **mh** and **nmh**, with some information on the mailbox format.

Babyl

class mailbox.**Babyl**(*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in Babyl format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *BabylMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

Babyl is a single-file mailbox format used by the Rmail mail user agent included with Emacs. The beginning of a message is indicated by a line containing the two characters Control-Underscore (`'\037'`) and Control-L (`'\014'`). The end of a message is indicated by the start of the next message or, in the case of the last message, a line containing a Control-Underscore (`'\037'`) character.

Messages in a Babyl mailbox have two sets of headers, original headers and so-called visible headers. Visible headers are typically a subset of the original headers that have been reformatted or abridged to be more attractive. Each message in a Babyl mailbox also has an accompanying list of *labels*, or short strings that record extra information about the message, and a list of all user-defined labels found in the mailbox is kept in the Babyl options section.

Babyl instances have all of the methods of *Mailbox* in addition to the following:

get_labels()

Return a list of the names of all user-defined labels used in the mailbox.

> **注解:** The actual messages are inspected to determine which labels exist in the mailbox rather than consulting the list of labels in the Babyl options section, but the Babyl section is updated whenever the mailbox is modified.

Some *Mailbox* methods implemented by *Babyl* deserve special remarks:

get_file(*key*)

In Babyl mailboxes, the headers of a message are not stored contiguously with the body of the message. To generate a file-like representation, the headers and body are copied together into a *StringIO* instance (from the *StringIO* module), which has an API identical to that of a file. As a result, the file-like object is truly independent of the underlying mailbox but does not save memory compared to a string representation.

lock()
unlock()

Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

参见:

Format of Version 5 Babyl Files A specification of the Babyl format.

Reading Mail with Rmail The Rmail manual, with some information on Babyl semantics.

MMDF

class mailbox.**MMDF**(*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in MMDF format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *MMDFMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

MMDF is a single-file mailbox format invented for the Multichannel Memorandum Distribution Facility, a mail transfer agent. Each message is in the same form as an mbox message but is bracketed before and after by lines containing four Control-A (`'\001'`) characters. As with the mbox format, the beginning of each message is

indicated by a line whose first five characters are "From ", but additional occurrences of "From " are not transformed to ">From " when storing messages because the extra message separator lines prevent mistaking such occurrences for the starts of subsequent messages.

Some *Mailbox* methods implemented by *MMDF* deserve special remarks:

get_file(*key*)
Using the file after calling `flush()` or `close()` on the *MMDF* instance may yield unpredictable results or raise an exception.

lock()
unlock()
Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

参见:

mmdf man page from tin A specification of MMDF format from the documentation of tin, a newsreader.

MMDF A Wikipedia article describing the Multichannel Memorandum Distribution Facility.

18.4.2 Message objects

class mailbox.**Message**([*message*])
A subclass of the *email.message* module's *Message*. Subclasses of *mailbox.Message* add mailbox-format-specific state and behavior.

If *message* is omitted, the new instance is created in a default, empty state. If *message* is an *email.message.Message* instance, its contents are copied; furthermore, any format-specific information is converted insofar as possible if *message* is a *Message* instance. If *message* is a string or a file, it should contain an **RFC 2822**-compliant message, which is read and parsed.

The format-specific state and behaviors offered by subclasses vary, but in general it is only the properties that are not specific to a particular mailbox that are supported (although presumably the properties are specific to a particular mailbox format). For example, file offsets for single-file mailbox formats and file names for directory-based mailbox formats are not retained, because they are only applicable to the original mailbox. But state such as whether a message has been read by the user or marked as important is retained, because it applies to the message itself.

There is no requirement that *Message* instances be used to represent messages retrieved using *Mailbox* instances. In some situations, the time and memory required to generate *Message* representations might not be acceptable. For such situations, *Mailbox* instances also offer string and file-like representations, and a custom message factory may be specified when a *Mailbox* instance is initialized.

MaildirMessage

class mailbox.**MaildirMessage**([*message*])
A message with Maildir-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Typically, a mail user agent application moves all of the messages in the `new` subdirectory to the `cur` subdirectory after the first time the user opens and closes the mailbox, recording that the messages are old whether or not they've actually been read. Each message in `cur` has an "info" section added to its file name to store information about its state. (Some mail readers may also add an "info" section to messages in `new`.) The "info" section may take one of two forms: it may contain "2," followed by a list of standardized flags (e.g., "2,FR") or it may contain "1," followed by so-called experimental information. Standard flags for Maildir messages are as follows:

标志	含义	解释
D	草稿	Under composition
F	已标记	标记为重要
P	已读	转发，重新发送或退回
R	已回复	回复给
S	查看	读取
T	已删除	标记为以后删除

MaildirMessage 实例提供以下方法：

get_subdir()
Return either "new" (if the message should be stored in the `new` subdirectory) or "cur" (if the message should be stored in the `cur` subdirectory).

注解： A message is typically moved from `new` to `cur` after its mailbox has been accessed, whether or not the message is has been read. A message `msg` has been read if `"S" in msg.get_flags()` is `True`.

---

set_subdir(*subdir*)
Set the subdirectory the message should be stored in. Parameter *subdir* must be either "new" or "cur".

get_flags()
Return a string specifying the flags that are currently set. If the message complies with the standard Maildir format, the result is the concatenation in alphabetical order of zero or one occurrence of each of `'D'`, `'F'`, `'P'`, `'R'`, `'S'`, and `'T'`. The empty string is returned if no flags are set or if "info" contains experimental semantics.

set_flags(*flags*)
Set the flags specified by *flags* and unset all others.

add_flag(*flag*)
Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character. The current "info" is overwritten whether or not it contains experimental information rather than flags.

remove_flag(*flag*)
Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character. If "info" contains experimental information rather than flags, the current "info" is not modified.

get_date()
Return the delivery date of the message as a floating-point number representing seconds since the epoch.

set_date(*date*)
Set the delivery date of the message to *date*, a floating-point number representing seconds since the epoch.

get_info()
Return a string containing the "info" for a message. This is useful for accessing and modifying "info" that is experimental (i.e., not a list of flags).

set_info(*info*)
Set "info" to *info*, which should be a string.

When a *MaildirMessage* instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

结果状态	*mboxMessage* 或 *MMDFMessage* 状态
“cur” 子目录	O 标记
F 标记	F 标记
R 标记	A 标记
S 标记	R 标记
T 标记	D 标记

When a *MaildirMessage* instance is created based upon an *MHMessage* instance, the following conversions take place:

结果状态	*MHMessage* 状态
“cur” 子目录	“unseen” 序列
“cur” subdirectory and S flag	非 “unseen” 序列
F 标记	“flagged” 序列
R 标记	“replied” 序列

When a *MaildirMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place:

结果状态	*BabylMessage* 状态
“cur” 子目录	“unseen” 标签
“cur” subdirectory and S flag	非 “unseen” 标签
P 标记	“forwarded” 或 “resent” 标签
R 标记	“answered” 标签
T 标记	“deleted” 标签

mboxMessage

class mailbox.**mboxMessage**([*message*])

A message with mbox-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Messages in an mbox mailbox are stored together in a single file. The sender’s envelope address and the time of delivery are typically stored in a line beginning with “From ” that is used to indicate the start of a message, though there is considerable variation in the exact format of this data among mbox implementations. Flags that indicate the state of the message, such as whether it has been read or marked as important, are typically stored in *Status* and *X-Status* headers.

Conventional flags for mbox messages are as follows:

标志	含义	解释
R	读取	读取
O	Old	以前由 MUA 检测
D	已删除	标记为以后删除
F	已标记	标记为重要
A	已回复	回复给

The “R” and “O” flags are stored in the *Status* header, and the “D”, “F”, and “A” flags are stored in the *X-Status* header. The flags and headers typically appear in the order mentioned.

mboxMessage instances offer the following methods:

get_from()
Return a string representing the "From " line that marks the start of the message in an mbox mailbox. The leading "From " and the trailing newline are excluded.

set_from(*from_*, *time_*=*None*)
Set the "From " line to *from_*, which should be specified without a leading "From " or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a *time.struct_time* instance, a tuple suitable for passing to *time.strftime()*, or `True` (to use *time.gmtime()*).

get_flags()
Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

set_flags(*flags*)
Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

add_flag(*flag*)
Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag(*flag*)
Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an *mboxMessage* instance is created based upon a *MaildirMessage* instance, a "From " line is generated based upon the *MaildirMessage* instance's delivery date, and the following conversions take place:

结果状态	*MaildirMessage* 状态
R 标记	S 标记
O 标记	"cur" 子目录
D 标记	T 标记
F 标记	F 标记
A 标记	R 标记

When an *mboxMessage* instance is created based upon an *MHMessage* instance, the following conversions take place:

结果状态	*MHMessage* 状态
R 标记和 O 标记	非 "unseen" 序列
O 标记	"unseen" 序列
F 标记	"flagged" 序列
A 标记	"replied" 序列

When an *mboxMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place:

结果状态	*BabylMessage* 状态
R 标记和 O 标记	非 "unseen" 标签
O 标记	"unseen" 标签
D 标记	"deleted" 标签
A 标记	"answered" 标签

When a *Message* instance is created based upon an *MMDFMessage* instance, the "From " line is copied and all flags directly correspond:

结果状态	*MMDFMessage* 状态
R 标记	R 标记
O 标记	O 标记
D 标记	D 标记
F 标记	F 标记
A 标记	A 标记

MHMessage

class mailbox.**MHMessage**([*message*])

A message with MH-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

MH messages do not support marks or flags in the traditional sense, but they do support sequences, which are logical groupings of arbitrary messages. Some mail reading programs (although not the standard **mh** and **nmh**) use sequences in much the same way flags are used with other formats, as follows:

序列	解释
未读	未读取，但先前被 MUA 检测到
已回复	回复给
已标记	标记为重要

MHMessage instances offer the following methods:

get_sequences()

Return a list of the names of sequences that include this message.

set_sequences(*sequences*)

Set the list of sequences that include this message.

add_sequence(*sequence*)

Add *sequence* to the list of sequences that include this message.

remove_sequence(*sequence*)

Remove *sequence* from the list of sequences that include this message.

When an *MHMessage* instance is created based upon a *MaildirMessage* instance, the following conversions take place:

结果状态	*MaildirMessage* 状态
"unseen" 序列	非 S 标记
"replied" 序列	R 标记
"flagged" 序列	F 标记

When an *MHMessage* instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

结果状态	*mboxMessage* 或*MMDFMessage* 状态
"unseen" 序列	非 R 标记
"replied" 序列	A 标记
"flagged" 序列	F 标记

When an *MHMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place:

结果状态	*BabylMessage* 状态
“unseen” 序列	“unseen” 标签
“replied” 序列	“answered” 标签

BabylMessage

class mailbox.**BabylMessage**([*message*])

A message with Babyl-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Certain message labels, called *attributes*, are defined by convention to have special meanings. The attributes are as follows:

标签	解释
未读	未读取，但先前被 MUA 检测到
deleted	标记为以后删除
filed	复制到另一个文件或邮箱
answered	回复给
forwarded	已转发
edited	由用户修改
resent	已重发

By default, Rmail displays only visible headers. The *BabylMessage* class, though, uses the original headers because they are more complete. Visible headers may be accessed explicitly if desired.

BabylMessage instances offer the following methods:

get_labels()
返回邮件上的标签列表。

set_labels(*labels*)
将消息上的标签列表设置为 *labels* 。

add_label(*label*)
将 *label* 添加到消息上的标签列表中。

remove_label(*label*)
从消息上的标签列表中删除 *label* 。

get_visible()
Return an *Message* instance whose headers are the message’s visible headers and whose body is empty.

set_visible(*visible*)
Set the message’s visible headers to be the same as the headers in *message*. Parameter *visible* should be a *Message* instance, an *email.message.Message* instance, a string, or a file-like object (which should be open in text mode).

update_visible()
When a *BabylMessage* instance’s original headers are modified, the visible headers are not automatically modified to correspond. This method updates the visible headers as follows: each visible header with a corresponding original header is set to the value of the original header, each visible header without a corresponding original header is removed, and any of *Date*, *From*, *Reply-To*, *To*, *CC*, and *Subject* that are present in the original headers but not the visible headers are added to the visible headers.

When a *BabylMessage* instance is created based upon a *MaildirMessage* instance, the following conversions take place:

结果状态	*MaildirMessage* 状态
“unseen” 标签	非 S 标记
“deleted” 标签	T 标记
“answered” 标签	R 标记
“forwarded” 标签	P 标记

When a *BabylMessage* instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

结果状态	*mboxMessage* 或*MMDFMessage* 状态
“unseen” 标签	非 R 标记
“deleted” 标签	D 标记
“answered” 标签	A 标记

When a *BabylMessage* instance is created based upon an *MHMessage* instance, the following conversions take place:

结果状态	*MHMessage* 状态
“unseen” 标签	“unseen” 序列
“answered” 标签	“replied” 序列

MMDFMessage

class mailbox.**MMDFMessage**([*message*])

A message with MMDF-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

As with message in an mbox mailbox, MMDF messages are stored with the sender’s address and the delivery date in an initial line beginning with “From ”. Likewise, flags that indicate the state of the message are typically stored in *Status* and *X-Status* headers.

Conventional flags for MMDF messages are identical to those of mbox message and are as follows:

标志	含义	解释
R	读取	读取
O	Old	以前由 MUA 检测
D	已删除	标记为以后删除
F	已标记	标记为重要
A	已回复	回复给

The “R” and “O” flags are stored in the *Status* header, and the “D”, “F”, and “A” flags are stored in the *X-Status* header. The flags and headers typically appear in the order mentioned.

MMDFMessage instances offer the following methods, which are identical to those offered by *mboxMessage*:

get_from()

Return a string representing the “From ” line that marks the start of the message in an mbox mailbox. The leading “From ” and the trailing newline are excluded.

set_from(*from_*, *time_*=*None*)

Set the “From ” line to *from_*, which should be specified without a leading “From ” or trailing newline.

For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a *time.struct_time* instance, a tuple suitable for passing to *time.strftime()*, or `True` (to use *time.gmtime()*).

get_flags()
Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

set_flags(*flags*)
Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

add_flag(*flag*)
Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag(*flag*)
Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an *MMDFMessage* instance is created based upon a *MaildirMessage* instance, a "From " line is generated based upon the *MaildirMessage* instance's delivery date, and the following conversions take place:

结果状态	*MaildirMessage* 状态
R 标记	S 标记
O 标记	"cur" 子目录
D 标记	T 标记
F 标记	F 标记
A 标记	R 标记

When an *MMDFMessage* instance is created based upon an *MHMessage* instance, the following conversions take place:

结果状态	*MHMessage* 状态
R 标记和 O 标记	非 "unseen" 序列
O 标记	"unseen" 序列
F 标记	"flagged" 序列
A 标记	"replied" 序列

When an *MMDFMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place:

结果状态	*BabylMessage* 状态
R 标记和 O 标记	非 "unseen" 标签
O 标记	"unseen" 标签
D 标记	"deleted" 标签
A 标记	"answered" 标签

When an *MMDFMessage* instance is created based upon an *mboxMessage* instance, the "From " line is copied and all flags directly correspond:

结果状态	*mboxMessage* 状态
R 标记	R 标记
O 标记	O 标记
D 标记	D 标记
F 标记	F 标记
A 标记	A 标记

18.4.3 异常

The following exception classes are defined in the *mailbox* module:

exception mailbox.**Error**
The based class for all other module-specific exceptions.

exception mailbox.**NoSuchMailboxError**
Raised when a mailbox is expected but is not found, such as when instantiating a *Mailbox* subclass with a path that does not exist (and with the *create* parameter set to `False`), or when opening a folder that does not exist.

exception mailbox.**NotEmptyError**
Raised when a mailbox is not empty but is expected to be, such as when deleting a folder that contains messages.

exception mailbox.**ExternalClashError**
Raised when some mailbox-related condition beyond the control of the program causes it to be unable to proceed, such as when failing to acquire a lock that another program already holds a lock, or when a uniquely-generated file name already exists.

exception mailbox.**FormatError**
Raised when the data in a file cannot be parsed, such as when an *MH* instance attempts to read a corrupted `.mh_sequences` file.

18.4.4 Deprecated classes and methods

2.6 版后已移除.

Older versions of the *mailbox* module do not support modification of mailboxes, such as adding or removing message, and do not provide classes to represent format-specific message properties. For backward compatibility, the older mailbox classes are still available, but the newer classes should be used in preference to them. The old classes have been removed in Python 3.

Older mailbox objects support only iteration and provide a single public method:

oldmailbox.**next**()
Return the next message in the mailbox, created with the optional *factory* argument passed into the mailbox object's constructor. By default this is an *rfc822.Message* object (see the *rfc822* module). Depending on the mailbox implementation the *fp* attribute of this object may be a true file object or a class instance simulating a file object, taking care of things like message boundaries if multiple mail messages are contained in a single file, etc. If no more messages are available, this method returns `None`.

Most of the older mailbox classes have names that differ from the current mailbox class names, except for *Maildir*. For this reason, the new *Maildir* class defines a `next()` method and its constructor differs slightly from those of the other new mailbox classes.

The older mailbox classes whose names are not the same as their newer counterparts are as follows:

class mailbox.**UnixMailbox**(*fp*[, *factory*])
Access to a classic Unix-style mailbox, where all messages are contained in a single file and separated by `From`

(a.k.a. From_) lines. The file object *fp* points to the mailbox file. The optional *factory* parameter is a callable that should create new message objects. *factory* is called with one argument, *fp* by the next() method of the mailbox object. The default is the *rfc822.Message* class (see the *rfc822* module –and the note below).

注解: For reasons of this module's internal implementation, you will probably want to open the *fp* object in binary mode. This is especially important on Windows.

For maximum portability, messages in a Unix-style mailbox are separated by any line that begins exactly with the string 'From ' (note the trailing space) if preceded by exactly two newlines. Because of the wide-range of variations in practice, nothing else on the From_ line should be considered. However, the current implementation doesn't check for the leading two newlines. This is usually fine for most applications.

The *UnixMailbox* class implements a more strict version of From_ line checking, using a regular expression that usually correctly matched From_ delimiters. It considers delimiter line to be separated by From name time lines. For maximum portability, use the *PortableUnixMailbox* class instead. This class is identical to *UnixMailbox* except that individual messages are separated by only From lines.

class mailbox.**PortableUnixMailbox**(*fp*[, *factory*])
A less-strict version of *UnixMailbox*, which considers only the From at the beginning of the line separating messages. The "*name time*" portion of the From line is ignored, to protect against some variations that are observed in practice. This works since lines in the message which begin with 'From ' are quoted by mail handling software at delivery-time.

class mailbox.**MmdfMailbox**(*fp*[, *factory*])
Access an MMDF-style mailbox, where all messages are contained in a single file and separated by lines consisting of 4 control-A characters. The file object *fp* points to the mailbox file. Optional *factory* is as with the *UnixMailbox* class.

class mailbox.**MHMailbox**(*dirname*[, *factory*])
Access an MH mailbox, a directory with each message in a separate file with a numeric name. The name of the mailbox directory is passed in *dirname*. *factory* is as with the *UnixMailbox* class.

class mailbox.**BabylMailbox**(*fp*[, *factory*])
Access a Babyl mailbox, which is similar to an MMDF mailbox. In Babyl format, each message has two sets of headers, the *original* headers and the *visible* headers. The original headers appear before a line containing only '*** EOOH ***' (End-Of-Original-Headers) and the visible headers appear after the EOOH line. Babyl-compliant mail readers will show you only the visible headers, and *BabylMailbox* objects will return messages containing only the visible headers. You'll have to do your own parsing of the mailbox file to get at the original headers. Mail messages start with the EOOH line and end with a line containing only '\037\014'. *factory* is as with the *UnixMailbox* class.

If you wish to use the older mailbox classes with the *email* module rather than the deprecated *rfc822* module, you can do so as follows:

```
import email
import email.Errors
import mailbox

def msgfactory(fp):
    try:
        return email.message_from_file(fp)
    except email.Errors.MessageParseError:
        # Don't return None since that will
        # stop the mailbox iterator
        return ''

mbox = mailbox.UnixMailbox(fp, msgfactory)
```

Alternatively, if you know your mailbox contains only well-formed MIME messages, you can simplify this to:

```
import email
import mailbox

mbox = mailbox.UnixMailbox(fp, email.message_from_file)
```

18.4.5 例子

A simple example of printing the subjects of all messages in a mailbox that seem interesting:

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']       # Could possibly be None.
    if subject and 'python' in subject.lower():
        print subject
```

To copy all mail from a Babyl mailbox to an MH mailbox, converting all of the format-specific information that can be converted:

```
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

This example sorts mail from several mailing lists into different mailboxes, being careful to avoid mail corruption due to concurrent modification by other programs, mail loss due to interruption of the program, or premature termination due to malformed messages in the mailbox:

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = dict((name, mailbox.mbox('~/email/%s' % name)) for name in list_names)
inbox = mailbox.Maildir('~/Maildir', factory=None)

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.errors.MessageParseError:
        continue                # The message is malformed. Just leave it.

    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # Get mailbox to use
            box = boxes[name]

            # Write copy to disk before removing original.
            # If there's a crash, you might duplicate a message, but
            # that's better than losing a message completely.
            box.lock()
```

(下页继续)

(续上页)

```
            box.add(message)
            box.flush()
            box.unlock()

            # Remove original message
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break               # Found destination, so stop looking.

for box in boxes.itervalues():
    box.close()
```

18.5 mhlib —Access to MH mailboxes

2.6 版后已移除: The *mhlib* module has been removed in Python 3. Use the *mailbox* instead.

The *mhlib* module provides a Python interface to MH folders and their contents.

The module contains three basic classes, *MH*, which represents a particular collection of folders, *Folder*, which represents a single folder, and *Message*, which represents a single message.

class mhlib.**MH**(*path*[, *profile*])
MH represents a collection of MH folders.

class mhlib.**Folder**(*mh*, *name*)
The *Folder* class represents a single folder and its messages.

class mhlib.**Message**(*folder*, *number*[, *name*])
Message objects represent individual messages in a folder. The Message class is derived from *mimetools.Message*.

18.5.1 MH Objects

MH instances have the following methods:

MH.**error**(*format*[, ...])
Print an error message –can be overridden.

MH.**getprofile**(*key*)
Return a profile entry (`None` if not set).

MH.**getpath**()
Return the mailbox pathname.

MH.**getcontext**()
Return the current folder name.

MH.**setcontext**(*name*)
Set the current folder name.

MH.**listfolders**()
Return a list of top-level folders.

MH.**listallfolders**()
Return a list of all folders.

MH.**listsubfolders**(*name*)
Return a list of direct subfolders of the given folder.

MH.**listallsubfolders**(*name*)
Return a list of all subfolders of the given folder.

MH.**makefolder**(*name*)
Create a new folder.

MH.**deletefolder**(*name*)
Delete a folder –must have no subfolders.

MH.**openfolder**(*name*)
Return a new open folder object.

18.5.2 Folder Objects

Folder instances represent open folders and have the following methods:

Folder.**error**(*format*[, ...])
Print an error message –can be overridden.

Folder.**getfullname**()
Return the folder's full pathname.

Folder.**getsequencesfilename**()
Return the full pathname of the folder's sequences file.

Folder.**getmessagefilename**(*n*)
Return the full pathname of message *n* of the folder.

Folder.**listmessages**()
Return a list of messages in the folder (as numbers).

Folder.**getcurrent**()
Return the current message number.

Folder.**setcurrent**(*n*)
Set the current message number to *n*.

Folder.**parsesequence**(*seq*)
Parse msgs syntax into list of messages.

Folder.**getlast**()
Get last message, or 0 if no messages are in the folder.

Folder.**setlast**(*n*)
Set last message (internal use only).

Folder.**getsequences**()
Return dictionary of sequences in folder. The sequence names are used as keys, and the values are the lists of message numbers in the sequences.

Folder.**putsequences**(*dict*)
Return dictionary of sequences in folder name: list.

Folder.**removemessages**(*list*)
Remove messages in list from folder.

Folder.**refilemessages**(*list*, *tofolder*)
Move messages in list to other folder.

Folder.**movemessage**(*n*, *tofolder*, *ton*)
Move one message to a given destination in another folder.

Folder.**copymessage**(*n*, *tofolder*, *ton*)
Copy one message to a given destination in another folder.

18.5.3 Message Objects

The *Message* class adds one method to those of *mimetools.Message*:

Message.**openmessage**(*n*)
Return a new open message object (costs a file descriptor).

18.6 mimetools —Tools for parsing MIME messages

2.3 版后已移除: The *email* package should be used in preference to the *mimetools* module. This module is present only to maintain backward compatibility, and it has been removed in 3.x.

This module defines a subclass of the *rfc822* module's *Message* class and a number of utility functions that are useful for the manipulation for MIME multipart or encoded message.

It defines the following items:

class mimetools.**Message**(*fp*[, *seekable*])
Return a new instance of the *Message* class. This is a subclass of the *rfc822.Message* class, with some additional methods (see below). The *seekable* argument has the same meaning as for *rfc822.Message*.

mimetools.**choose_boundary**()
Return a unique string that has a high likelihood of being usable as a part boundary. The string has the form `'hostipaddr.uid.pid.timestamp.random'`.

mimetools.**decode**(*input*, *output*, *encoding*)
Read data encoded using the allowed MIME *encoding* from open file object *input* and write the decoded data to open file object *output*. Valid values for *encoding* include `'base64'`, `'quoted-printable'`, `'uuencode'`, `'x-uuencode'`, `'uue'`, `'x-uue'`, `'7bit'`, and `'8bit'`. Decoding messages encoded in `'7bit'` or `'8bit'` has no effect. The input is simply copied to the output.

mimetools.**encode**(*input*, *output*, *encoding*)
Read data from open file object *input* and write it encoded using the allowed MIME *encoding* to open file object *output*. Valid values for *encoding* are the same as for *decode()*.

mimetools.**copyliteral**(*input*, *output*)
Read lines from open file *input* until EOF and write them to open file *output*.

mimetools.**copybinary**(*input*, *output*)
Read blocks until EOF from open file *input* and write them to open file *output*. The block size is currently fixed at 8192.

参见:

Module *email* Comprehensive email handling package; supersedes the *mimetools* module.

Module *rfc822* Provides the base class for *mimetools.Message*.

Module *multifile* Support for reading files which contain distinct parts, such as MIME data.

http://faqs.cs.uu.nl/na-dir/mail/mime-faq/.html The MIME Frequently Asked Questions document. For an overview of MIME, see the answer to question 1.1 in Part 1 of this document.

18.6.1 Additional Methods of Message Objects

The *Message* class defines the following methods in addition to the *rfc822.Message* methods:

`Message.`**`getplist`**`()`
Return the parameter list of the *Content-Type* header. This is a list of strings. For parameters of the form `key=value`, *key* is converted to lower case but *value* is not. For example, if the message contains the header `Content-type: text/html; spam=1; Spam=2; Spam` then *getplist()* will return the Python list `['spam=1', 'spam=2', 'Spam']`.

`Message.`**`getparam`**`(`*name*`)`
Return the *value* of the first parameter (as returned by *getplist()*) of the form `name=value` for the given *name*. If *value* is surrounded by quotes of the form '<…>' or '"…"', these are removed.

`Message.`**`getencoding`**`()`
Return the encoding specified in the *Content-Transfer-Encoding* message header. If no such header exists, return `'7bit'`. The encoding is converted to lower case.

`Message.`**`gettype`**`()`
Return the message type (of the form `type/subtype`) as specified in the *Content-Type* header. If no such header exists, return `'text/plain'`. The type is converted to lower case.

`Message.`**`getmaintype`**`()`
Return the main type as specified in the *Content-Type* header. If no such header exists, return `'text'`. The main type is converted to lower case.

`Message.`**`getsubtype`**`()`
Return the subtype as specified in the *Content-Type* header. If no such header exists, return `'plain'`. The subtype is converted to lower case.

18.7 `mimetypes` —Map filenames to MIME types

Source code: Lib/mimetypes.py

The *mimetypes* module converts between a filename or URL and the MIME type associated with the filename extension. Conversions are provided from filename to MIME type and from MIME type to filename extension; encodings are not supported for the latter conversion.

The module provides one class and a number of convenience functions. The functions are the normal interface to this module, but some applications may be interested in the class as well.

The functions described below provide the primary interface for this module. If the module has not been initialized, they will call *init()* if they rely on the information *init()* sets up.

`mimetypes.`**`guess_type`**`(`*url*, *strict=True*`)`
Guess the type of a file based on its filename or URL, given by *url*. The return value is a tuple `(type, encoding)` where *type* is `None` if the type can't be guessed (missing or unknown suffix) or a string of the form `'type/subtype'`, usable for a MIME *content-type* header.

encoding is `None` for no encoding or the name of the program used to encode (e.g. **`compress`** or **`gzip`**). The encoding is suitable for use as a *Content-Encoding* header, **not** as a *Content-Transfer-Encoding*

header. The mappings are table driven. Encoding suffixes are case sensitive; type suffixes are first tried case sensitively, then case insensitively.

The optional *strict* argument is a flag specifying whether the list of known MIME types is limited to only the official types registered with IANA. When *strict* is `True` (the default), only the IANA types are supported; when *strict* is `False`, some additional non-standard but commonly used MIME types are also recognized.

`mimetypes.`**`guess_all_extensions`**(*type*, *strict=True*)
Guess the extensions for a file based on its MIME type, given by *type*. The return value is a list of strings giving all possible filename extensions, including the leading dot (`'.'`). The extensions are not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by *guess_type()*.

The optional *strict* argument has the same meaning as with the *guess_type()* function.

`mimetypes.`**`guess_extension`**(*type*, *strict=True*)
Guess the extension for a file based on its MIME type, given by *type*. The return value is a string giving a filename extension, including the leading dot (`'.'`). The extension is not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by *guess_type()*. If no extension can be guessed for *type*, `None` is returned.

The optional *strict* argument has the same meaning as with the *guess_type()* function.

Some additional functions and data items are available for controlling the behavior of the module.

`mimetypes.`**`init`**(*files=None*)
Initialize the internal data structures. If given, *files* must be a sequence of file names which should be used to augment the default type map. If omitted, the file names to use are taken from *knownfiles*; on Windows, the current registry settings are loaded. Each file named in *files* or *knownfiles* takes precedence over those named before it. Calling *init()* repeatedly is allowed.

Specifying an empty list for *files* will prevent the system defaults from being applied: only the well-known values will be present from a built-in list.

在 2.7 版更改: Previously, Windows registry settings were ignored.

`mimetypes.`**`read_mime_types`**(*filename*)
Load the type map given in the file *filename*, if it exists. The type map is returned as a dictionary mapping filename extensions, including the leading dot (`'.'`), to strings of the form `'type/subtype'`. If the file *filename* does not exist or cannot be read, `None` is returned.

`mimetypes.`**`add_type`**(*type*, *ext*, *strict=True*)
Add a mapping from the MIME type *type* to the extension *ext*. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When *strict* is `True` (the default), the mapping will be added to the official MIME types, otherwise to the non-standard ones.

`mimetypes.`**`inited`**
Flag indicating whether or not the global data structures have been initialized. This is set to `True` by *init()*.

`mimetypes.`**`knownfiles`**
List of type map file names commonly installed. These files are typically named `mime.types` and are installed in different locations by different packages.

`mimetypes.`**`suffix_map`**
Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately.

`mimetypes.`**`encodings_map`**
Dictionary mapping filename extensions to encoding types.

mimetypes.**types_map**
Dictionary mapping filename extensions to MIME types.

mimetypes.**common_types**
Dictionary mapping filename extensions to non-standard, but commonly found MIME types.

An example usage of the module:

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

18.7.1 MimeTypes Objects

The *MimeTypes* class may be useful for applications which may want more than one MIME-type database; it provides an interface similar to the one of the *mimetypes* module.

class mimetypes.**MimeTypes**(*filenames=()*, *strict=True*)
This class represents a MIME-types database. By default, it provides access to the same database as the rest of this module. The initial database is a copy of that provided by the module, and may be extended by loading additional `mime.types`-style files into the database using the *read()* or *readfp()* methods. The mapping dictionaries may also be cleared before loading additional data if the default data is not desired.

The optional *filenames* parameter can be used to cause additional files to be loaded "on top" of the default database.

suffix_map
Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately. This is initially a copy of the global *suffix_map* defined in the module.

encodings_map
Dictionary mapping filename extensions to encoding types. This is initially a copy of the global *encodings_map* defined in the module.

types_map
Tuple containing two dictionaries, mapping filename extensions to MIME types: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by *common_types* and *types_map*.

types_map_inv
Tuple containing two dictionaries, mapping MIME types to a list of filename extensions: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by *common_types* and *types_map*.

guess_extension(*type*, *strict=True*)
Similar to the *guess_extension()* function, using the tables stored as part of the object.

guess_type(*url*, *strict=True*)
Similar to the *guess_type()* function, using the tables stored as part of the object.

guess_all_extensions(*type*, *strict=True*)
Similar to the *guess_all_extensions()* function, using the tables stored as part of the object.

read(*filename*, *strict=True*)
Load MIME information from a file named *filename*. This uses *readfp()* to parse the file.

If *strict* is `True`, information will be added to list of standard types, else to the list of non-standard types.

readfp(*fp*, *strict=True*)
Load MIME type information from an open file *fp*. The file must have the format of the standard `mime.types` files.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

read_windows_registry(*strict=True*)
Load MIME type information from the Windows registry. Availability: Windows.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

2.7 新版功能.

18.8 MimeWriter —Generic MIME file writer

2.3 版后已移除: The *email* package should be used in preference to the *MimeWriter* module. This module is present only to maintain backward compatibility.

This module defines the class *MimeWriter*. The *MimeWriter* class implements a basic formatter for creating MIME multi-part files. It doesn't seek around the output file nor does it use large amounts of buffer space. You must write the parts out in the order that they should occur in the final file. *MimeWriter* does buffer the headers you add, allowing you to rearrange their order.

class MimeWriter.**MimeWriter**(*fp*)
Return a new instance of the *MimeWriter* class. The only argument passed, *fp*, is a file object to be used for writing. Note that a *StringIO* object could also be used.

18.8.1 MimeWriter Objects

MimeWriter instances have the following methods:

MimeWriter.**addheader**(*key*, *value*[, *prefix*])
Add a header line to the MIME message. The *key* is the name of the header, where the *value* obviously provides the value of the header. The optional argument *prefix* determines where the header is inserted; `0` means append at the end, `1` is insert at the start. The default is to append.

MimeWriter.**flushheaders**()
Causes all headers accumulated so far to be written out (and forgotten). This is useful if you don't need a body part at all, e.g. for a subpart of type *message/rfc822* that's (mis)used to store some header-like information.

MimeWriter.**startbody**(*ctype*[, *plist*[, *prefix*]])
Returns a file-like object which can be used to write to the body of the message. The content-type is set to the provided *ctype*, and the optional parameter *plist* provides additional parameters for the content-type declaration. *prefix* functions as in *addheader()* except that the default is to insert at the start.

MimeWriter.**startmultipartbody**(*subtype*[, *boundary*[, *plist*[, *prefix*]]])
Returns a file-like object which can be used to write to the body of the message. Additionally, this method initializes the multi-part code, where *subtype* provides the multipart subtype, *boundary* may provide a user-defined boundary specification, and *plist* provides optional parameters for the subtype. *prefix* functions as in *startbody()*. Subparts should be created using *nextpart()*.

MimeWriter.**nextpart**()
Returns a new instance of *MimeWriter* which represents an individual part in a multipart message. This may be used to write the part as well as used for creating recursively complex multipart messages. The message must first be initialized with *startmultipartbody()* before using *nextpart()*.

MimeWriter.**lastpart**()
This is used to designate the last part of a multipart message, and should *always* be used when writing multipart messages.

18.9 mimify —MIME processing of mail messages

2.3 版后已移除: The *email* package should be used in preference to the *mimify* module. This module is present only to maintain backward compatibility.

The *mimify* module defines two functions to convert mail messages to and from MIME format. The mail message can be either a simple message or a so-called multipart message. Each part is treated separately. Mimifying (a part of) a message entails encoding the message as quoted-printable if it contains any characters that cannot be represented using 7-bit ASCII. Unmimifying (a part of) a message entails undoing the quoted-printable encoding. Mimify and unmimify are especially useful when a message has to be edited before being sent. Typical use would be:

```
unmimify message
edit message
mimify message
send message
```

The modules defines the following user-callable functions and user-settable variables:

mimify.**mimify**(*infile*, *outfile*)
Copy the message in *infile* to *outfile*, converting parts to quoted-printable and adding MIME mail headers when necessary. *infile* and *outfile* can be file objects (actually, any object that has a *readline()* method (for *infile*) or a `write()` method (for *outfile*)) or strings naming the files. If *infile* and *outfile* are both strings, they may have the same value.

mimify.**unmimify**(*infile*, *outfile*[, *decode_base64*])
Copy the message in *infile* to *outfile*, decoding all quoted-printable parts. *infile* and *outfile* can be file objects (actually, any object that has a *readline()* method (for *infile*) or a `write()` method (for *outfile*)) or strings naming the files. If *infile* and *outfile* are both strings, they may have the same value. If the *decode_base64* argument is provided and tests true, any parts that are coded in the base64 encoding are decoded as well.

mimify.**mime_decode_header**(*line*)
Return a decoded version of the encoded header line in *line*. This only supports the ISO 8859-1 charset (Latin-1).

mimify.**mime_encode_header**(*line*)
Return a MIME-encoded version of the header line in *line*.

mimify.**MAXLEN**
By default, a part will be encoded as quoted-printable when it contains any non-ASCII characters (characters with the 8th bit set), or if there are any lines longer than *MAXLEN* characters (default value 200).

mimify.**CHARSET**
When not specified in the mail headers, a character set must be filled in. The string used is stored in *CHARSET*, and the default value is ISO-8859-1 (also known as Latin1 (latin-one)).

This module can also be used from the command line. Usage is as follows:

```
mimify.py -e [-l length] [infile [outfile]]
mimify.py -d [-b] [infile [outfile]]
```

to encode (mimify) and decode (unmimify) respectively. *infile* defaults to standard input, *outfile* defaults to standard output. The same file can be specified for input and output.

If the **-l** option is given when encoding, if there are any lines longer than the specified *length*, the containing part will be encoded.

If the **-b** option is given when decoding, any base64 parts will be decoded as well.

参见:

Module ***quopri*** Encode and decode MIME quoted-printable files.

18.10 multifile —Support for files containing distinct parts

2.5 版后已移除: The *email* package should be used in preference to the *multifile* module. This module is present only to maintain backward compatibility.

The *MultiFile* object enables you to treat sections of a text file as file-like input objects, with `''` being returned by *readline()* when a given delimiter pattern is encountered. The defaults of this class are designed to make it useful for parsing MIME multipart messages, but by subclassing it and overriding methods it can be easily adapted for more general use.

class multifile.**MultiFile**(*fp*[, *seekable*])

Create a multi-file. You must instantiate this class with an input object argument for the *MultiFile* instance to get lines from, such as a file object returned by *open()*.

MultiFile only ever looks at the input object's *readline()*, *seek()* and *tell()* methods, and the latter two are only needed if you want random access to the individual MIME parts. To use *MultiFile* on a non-seekable stream object, set the optional *seekable* argument to false; this will prevent using the input object's *seek()* and *tell()* methods.

It will be useful to know that in *MultiFile*'s view of the world, text is composed of three kinds of lines: data, section-dividers, and end-markers. MultiFile is designed to support parsing of messages that may have multiple nested message parts, each with its own pattern for section-divider and end-marker lines.

参见:

Module ***email*** Comprehensive email handling package; supersedes the *multifile* module.

18.10.1 MultiFile Objects

A *MultiFile* instance has the following methods:

MultiFile.**readline**(*str*)

Read a line. If the line is data (not a section-divider or end-marker or real EOF) return it. If the line matches the most-recently-stacked boundary, return `''` and set `self.last` to 1 or 0 according as the match is or is not an end-marker. If the line matches any other stacked boundary, raise an error. On encountering end-of-file on the underlying stream object, the method raises `Error` unless all boundaries have been popped.

MultiFile.**readlines**(*str*)

Return all lines remaining in this part as a list of strings.

MultiFile.**read**()

Read all lines, up to the next section. Return them as a single (multiline) string. Note that this doesn't take a size argument!

MultiFile.**seek**(*pos*[, *whence*])
Seek. Seek indices are relative to the start of the current section. The *pos* and *whence* arguments are interpreted as for a file seek.

MultiFile.**tell**()
Return the file position relative to the start of the current section.

MultiFile.**next**()
Skip lines to the next section (that is, read lines until a section-divider or end-marker has been consumed). Return true if there is such a section, false if an end-marker is seen. Re-enable the most-recently-pushed boundary.

MultiFile.**is_data**(*str*)
Return true if *str* is data and false if it might be a section boundary. As written, it tests for a prefix other than `'--'` at start of line (which all MIME boundaries have) but it is declared so it can be overridden in derived classes.

Note that this test is used intended as a fast guard for the real boundary tests; if it always returns false it will merely slow processing, not cause it to fail.

MultiFile.**push**(*str*)
Push a boundary string. When a decorated version of this boundary is found as an input line, it will be interpreted as a section-divider or end-marker (depending on the decoration, see **RFC 2045**). All subsequent reads will return the empty string to indicate end-of-file, until a call to *pop()* removes the boundary a or *next()* call reenables it.

It is possible to push more than one boundary. Encountering the most-recently-pushed boundary will return EOF; encountering any other boundary will raise an error.

MultiFile.**pop**()
Pop a section boundary. This boundary will no longer be interpreted as EOF.

MultiFile.**section_divider**(*str*)
Turn a boundary into a section-divider line. By default, this method prepends `'--'` (which MIME section boundaries have) but it is declared so it can be overridden in derived classes. This method need not append LF or CR-LF, as comparison with the result ignores trailing whitespace.

MultiFile.**end_marker**(*str*)
Turn a boundary string into an end-marker line. By default, this method prepends `'--'` and appends `'--'` (like a MIME-multipart end-of-message marker) but it is declared so it can be overridden in derived classes. This method need not append LF or CR-LF, as comparison with the result ignores trailing whitespace.

Finally, *MultiFile* instances have two public instance variables:

MultiFile.**level**
Nesting depth of the current part.

MultiFile.**last**
True if the last end-of-file was for an end-of-message marker.

18.10.2 MultiFile Example

```
import mimetools
import multifile
import StringIO

def extract_mime_part_matching(stream, mimetype):
    """Return the first element in a multipart MIME message on stream
    matching mimetype."""

    msg = mimetools.Message(stream)
```

(下页继续)

(续上页)

```
    msgtype = msg.gettype()
    params = msg.getplist()

    data = StringIO.StringIO()
    if msgtype[:10] == "multipart/":

        file = multifile.MultiFile(stream)
        file.push(msg.getparam("boundary"))
        while file.next():
            submsg = mimetools.Message(file)
            try:
                data = StringIO.StringIO()
                mimetools.decode(file, data, submsg.getencoding())
            except ValueError:
                continue
            if submsg.gettype() == mimetype:
                break
        file.pop()
    return data.getvalue()
```

18.11 rfc822 —Parse RFC 2822 mail headers

2.3 版后已移除: The *email* package should be used in preference to the *rfc822* module. This module is present only to maintain backward compatibility, and has been removed in Python 3.

This module defines a class, *Message*, which represents an "email message" as defined by the Internet standard **RFC 2822**.[1] Such messages consist of a collection of message headers, and a message body. This module also defines a helper class *AddressList* for parsing **RFC 2822** addresses. Please refer to the RFC for information on the specific syntax of **RFC 2822** messages.

The *mailbox* module provides classes to read mailboxes produced by various end-user mail programs.

class rfc822.**Message**(*file*[, *seekable*])

A *Message* instance is instantiated with an input object as parameter. Message relies only on the input object having a *readline()* method; in particular, ordinary file objects qualify. Instantiation reads headers from the input object up to a delimiter line (normally a blank line) and stores them in the instance. The message body, following the headers, is not consumed.

This class can work with any input object that supports a *readline()* method. If the input object has seek and tell capability, the *rewindbody()* method will work; also, illegal lines will be pushed back onto the input stream. If the input object lacks seek but has an `unread()` method that can push back a line of input, *Message* will use that to push back illegal lines. Thus this class can be used to parse messages coming from a buffered stream.

The optional *seekable* argument is provided as a workaround for certain stdio libraries in which `tell()` discards buffered data before discovering that the `lseek()` system call doesn't work. For maximum portability, you should set the seekable argument to zero to prevent that initial `tell()` when passing in an unseekable object such as a file object created from a socket object.

Input lines as read from the file may either be terminated by CR-LF or by a single linefeed; a terminating CR-LF is replaced by a single linefeed before the line is stored.

All header matching is done independent of upper or lower case; e.g. `m['From']`, `m['from']` and `m['FROM']` all yield the same result.

[1] This module originally conformed to **RFC 822**, hence the name. Since then, **RFC 2822** has been released as an update to **RFC 822**. This module should be considered **RFC 2822**-conformant, especially in cases where the syntax or semantics have changed since **RFC 822**.

class rfc822.**AddressList**(*field*)
You may instantiate the *AddressList* helper class using a single string parameter, a comma-separated list of **RFC 2822** addresses to be parsed. (The parameter None yields an empty list.)

rfc822.**quote**(*str*)
Return a new string with backslashes in *str* replaced by two backslashes and double quotes replaced by backslash-double quote.

rfc822.**unquote**(*str*)
Return a new string which is an *unquoted* version of *str*. If *str* ends and begins with double quotes, they are stripped off. Likewise if *str* ends and begins with angle brackets, they are stripped off.

rfc822.**parseaddr**(*address*)
Parse *address*, which should be the value of some address-containing field such as *To* or *Cc*, into its constituent "realname" and "email address" parts. Returns a tuple of that information, unless the parse fails, in which case a 2-tuple (None, None) is returned.

rfc822.**dump_address_pair**(*pair*)
The inverse of *parseaddr()*, this takes a 2-tuple of the form (realname, email_address) and returns the string value suitable for a *To* or *Cc* header. If the first element of *pair* is false, then the second element is returned unmodified.

rfc822.**parsedate**(*date*)
Attempts to parse a date according to the rules in **RFC 2822**. however, some mailers don't follow that format as specified, so *parsedate()* tries to guess correctly in such cases. *date* is a string containing an **RFC 2822** date, such as 'Mon, 20 Nov 1995 19:12:08 -0500'. If it succeeds in parsing the date, *parsedate()* returns a 9-tuple that can be passed directly to *time.mktime()*; otherwise None will be returned. Note that indexes 6, 7, and 8 of the result tuple are not usable.

rfc822.**parsedate_tz**(*date*)
Performs the same function as *parsedate()*, but returns either None or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to *time.mktime()*, and the tenth is the offset of the date's timezone from UTC (which is the official term for Greenwich Mean Time). (Note that the sign of the timezone offset is the opposite of the sign of the time.timezone variable for the same timezone; the latter variable follows the POSIX standard while this module follows **RFC 2822**.) If the input string has no timezone, the last element of the tuple returned is None. Note that indexes 6, 7, and 8 of the result tuple are not usable.

rfc822.**mktime_tz**(*tuple*)
Turn a 10-tuple as returned by *parsedate_tz()* into a UTC timestamp. If the timezone item in the tuple is None, assume local time. Minor deficiency: this first interprets the first 8 elements as a local time and then compensates for the timezone difference; this may yield a slight error around daylight savings time switch dates. Not enough to worry about for common use.

参见:

Module *email* Comprehensive email handling package; supersedes the *rfc822* module.

Module *mailbox* Classes to read various mailbox formats produced by end-user mail programs.

Module *mimetools* Subclass of *rfc822.Message* that handles MIME encoded messages.

18.11.1 Message Objects

A *Message* instance has the following methods:

Message.**rewindbody**()
Seek to the start of the message body. This only works if the file object is seekable.

Message.**isheader**(*line*)
Returns a line's canonicalized fieldname (the dictionary key that will be used to index it) if the line is a legal **RFC 2822** header; otherwise returns None (implying that parsing should stop here and the line be pushed back on the input stream). It is sometimes useful to override this method in a subclass.

Message.**islast**(*line*)
Return true if the given line is a delimiter on which Message should stop. The delimiter line is consumed, and the file object's read location positioned immediately after it. By default this method just checks that the line is blank, but you can override it in a subclass.

Message.**iscomment**(*line*)
Return True if the given line should be ignored entirely, just skipped. By default this is a stub that always returns False, but you can override it in a subclass.

Message.**getallmatchingheaders**(*name*)
Return a list of lines consisting of all headers matching *name*, if any. Each physical line, whether it is a continuation line or not, is a separate list item. Return the empty list if no header matches *name*.

Message.**getfirstmatchingheader**(*name*)
Return a list of lines comprising the first header matching *name*, and its continuation line(s), if any. Return None if there is no header matching *name*.

Message.**getrawheader**(*name*)
Return a single string consisting of the text after the colon in the first header matching *name*. This includes leading whitespace, the trailing linefeed, and internal linefeeds and whitespace if there any continuation line(s) were present. Return None if there is no header matching *name*.

Message.**getheader**(*name*[, *default*])
Return a single string consisting of the last header matching *name*, but strip leading and trailing whitespace. Internal whitespace is not stripped. The optional *default* argument can be used to specify a different default to be returned when there is no header matching *name*; it defaults to None. This is the preferred way to get parsed headers.

Message.**get**(*name*[, *default*])
An alias for *getheader()*, to make the interface more compatible with regular dictionaries.

Message.**getaddr**(*name*)
Return a pair (full name, email address) parsed from the string returned by getheader(name). If no header matching *name* exists, return (None, None); otherwise both the full name and the address are (possibly empty) strings.

Example: If *m*'s first *From* header contains the string 'jack@cwi.nl (Jack Jansen)', then m.getaddr('From') will yield the pair ('Jack Jansen', 'jack@cwi.nl'). If the header contained 'Jack Jansen <jack@cwi.nl>' instead, it would yield the exact same result.

Message.**getaddrlist**(*name*)
This is similar to getaddr(list), but parses a header containing a list of email addresses (e.g. a *To* header) and returns a list of (full name, email address) pairs (even if there was only one address in the header). If there is no header matching *name*, return an empty list.

If multiple headers exist that match the named header (e.g. if there are several *Cc* headers), all are parsed for addresses. Any continuation lines the named headers contain are also parsed.

Message.**getdate**(*name*)
Retrieve a header using *getheader()* and parse it into a 9-tuple compatible with *time.mktime()*; note that fields 6, 7, and 8 are not usable. If there is no header matching *name*, or it is unparsable, return None.

Date parsing appears to be a black art, and not all mailers adhere to the standard. While it has been tested and found correct on a large collection of email from many sources, it is still possible that this function may occasionally yield an incorrect result.

Message.**getdate_tz**(*name*)
Retrieve a header using *getheader()* and parse it into a 10-tuple; the first 9 elements will make a tuple compatible with *time.mktime()*, and the 10th is a number giving the offset of the date's timezone from UTC. Note that fields 6, 7, and 8 are not usable. Similarly to *getdate()*, if there is no header matching *name*, or it is unparsable, return None.

Message instances also support a limited mapping interface. In particular: m[name] is like m.getheader(name) but raises *KeyError* if there is no matching header; and len(m), m.get(name[, default]), name in m, m.keys(), m.values() m.items(), and m.setdefault(name[, default]) act as expected, with the one difference that setdefault() uses an empty string as the default value. *Message* instances also support the mapping writable interface m[name] = value and del m[name]. *Message* objects do not support the clear(), *copy()*, popitem(), or update() methods of the mapping interface. (Support for get() and setdefault() was only added in Python 2.2.)

Finally, *Message* instances have some public instance variables:

Message.**headers**
A list containing the entire set of header lines, in the order in which they were read (except that setitem calls may disturb this order). Each line contains a trailing newline. The blank line terminating the headers is not contained in the list.

Message.**fp**
The file or file-like object passed at instantiation time. This can be used to read the message content.

Message.**unixfrom**
The Unix From line, if the message had one, or an empty string. This is needed to regenerate the message in some contexts, such as an mbox-style mailbox file.

18.11.2 AddressList Objects

An *AddressList* instance has the following methods:

AddressList.**__len__**()
Return the number of addresses in the address list.

AddressList.**__str__**()
Return a canonicalized string representation of the address list. Addresses are rendered in "name" <host@domain> form, comma-separated.

AddressList.**__add__**(*alist*)
Return a new *AddressList* instance that contains all addresses in both *AddressList* operands, with duplicates removed (set union).

AddressList.**__iadd__**(*alist*)
In-place version of *__add__()*; turns this *AddressList* instance into the union of itself and the right-hand instance, *alist*.

AddressList.**__sub__**(*alist*)
Return a new *AddressList* instance that contains every address in the left-hand *AddressList* operand that is not present in the right-hand address operand (set difference).

AddressList.**__isub__**(*alist*)
In-place version of *__sub__()*, removing addresses in this list which are also in *alist*.

Finally, *AddressList* instances have one public instance variable:

AddressList.**addresslist**
A list of tuple string pairs, one per address. In each member, the first is the canonicalized name part, the second is the actual route-address ('@'-separated username-host.domain pair).

18.12 base64 —RFC 3548: Base16, Base32, Base64 Data Encodings

This module provides data encoding and decoding as specified in **RFC 3548**. This standard defines the Base16, Base32, and Base64 algorithms for encoding and decoding arbitrary binary strings into text strings that can be safely sent by email, used as parts of URLs, or included as part of an HTTP POST request. The encoding algorithm is not the same as the **uuencode** program.

There are two interfaces provided by this module. The modern interface supports encoding and decoding string objects using both base-64 alphabets defined in **RFC 3548** (normal, and URL- and filesystem-safe). The legacy interface provides for encoding and decoding to and from file-like objects as well as strings, but only using the Base64 standard alphabet.

The modern interface, which was introduced in Python 2.4, provides:

base64.**b64encode**(*s*[, *altchars*])
Encode a string using Base64.

s is the string to encode. Optional *altchars* must be a string of at least length 2 (additional characters are ignored) which specifies an alternative alphabet for the + and / characters. This allows an application to e.g. generate URL or filesystem safe Base64 strings. The default is None, for which the standard Base64 alphabet is used.

The encoded string is returned.

base64.**b64decode**(*s*[, *altchars*])
Decode a Base64 encoded string.

s is the string to decode. Optional *altchars* must be a string of at least length 2 (additional characters are ignored) which specifies the alternative alphabet used instead of the + and / characters.

The decoded string is returned. A *TypeError* is raised if *s* is incorrectly padded. Characters that are neither in the normal base-64 alphabet nor the alternative alphabet are discarded prior to the padding check.

base64.**standard_b64encode**(*s*)
Encode string *s* using the standard Base64 alphabet.

base64.**standard_b64decode**(*s*)
Decode string *s* using the standard Base64 alphabet.

base64.**urlsafe_b64encode**(*s*)
Encode string *s* using the URL- and filesystem-safe alphabet, which substitutes - instead of + and _ instead of / in the standard Base64 alphabet. The result can still contain =.

base64.**urlsafe_b64decode**(*s*)
Decode string *s* using the URL- and filesystem-safe alphabet, which substitutes - instead of + and _ instead of / in the standard Base64 alphabet.

base64.**b32encode**(*s*)
Encode a string using Base32. *s* is the string to encode. The encoded string is returned.

base64.**b32decode**(*s*[, *casefold*[, *map01*]])
Decode a Base32 encoded string.

s is the string to decode. Optional *casefold* is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is `False`.

RFC 3548 允许将字母 0(zero) 映射为字母 O(oh)，并可以选择是否将字母 1(one) 映射为 I(eye) 或 L(el)。可选参数 *map01* 不是 `None` 时，它的值指定 1 的映射目标 (I 或 l)，当 *map01* 非 `None` 时，0 总是被映射为 O。为了安全考虑，默认值被设为 `None`，如果是这样，0 和 1 不允许被作为输入。

The decoded string is returned. A *TypeError* is raised if *s* is incorrectly padded or if there are non-alphabet characters present in the string.

`base64.`**`b16encode`**(*s*)
Encode a string using Base16.

s is the string to encode. The encoded string is returned.

`base64.`**`b16decode`**(*s*[, *casefold*])
Decode a Base16 encoded string.

s is the string to decode. Optional *casefold* is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is `False`.

The decoded string is returned. A *TypeError* is raised if *s* were incorrectly padded or if there are non-alphabet characters present in the string.

旧式接口:

`base64.`**`decode`**(*input*, *output*)
Decode the contents of the *input* file and write the resulting binary data to the *output* file. *input* and *output* must either be file objects or objects that mimic the file object interface. *input* will be read until `input.read()` returns an empty string.

`base64.`**`decodestring`**(*s*)
Decode the string *s*, which must contain one or more lines of base64 encoded data, and return a string containing the resulting binary data.

`base64.`**`encode`**(*input*, *output*)
Encode the contents of the *input* file and write the resulting base64 encoded data to the *output* file. *input* and *output* must either be file objects or objects that mimic the file object interface. *input* will be read until `input.read()` returns an empty string. *encode()* returns the encoded data plus a trailing newline character (`'\n'`).

`base64.`**`encodestring`**(*s*)
Encode the string *s*, which can contain arbitrary binary data, and return a string containing one or more lines of base64-encoded data. *encodestring()* returns a string containing one or more lines of base64-encoded data always including an extra trailing newline (`'\n'`).

此模块的一个使用示例:

```
>>> import base64
>>> encoded = base64.b64encode('data to be encoded')
>>> encoded
'ZGF0YSB0byBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
'data to be encoded'
```

参见:

模块*binascii* 支持模块，包含 ASCII 到二进制和二进制到 ASCII 转换。

RFC 1521 - MIME (Multipurpose Internet Mail Extensions) 第一部分：规定并描述因特网消息体的格式的机制。 第 5.2 节，“Base64 内容转换编码格式”提供了 base64 编码格式的定义。

18.13 binhex —对 binhex4 文件进行编码和解码

This module encodes and decodes files in binhex4 format, a format allowing representation of Macintosh files in ASCII. On the Macintosh, both forks of a file and the finder information are encoded (or decoded), on other platforms only the data fork is handled.

注解: In Python 3.x, special Macintosh support has been removed.

binhex 模块定义了以下功能:

binhex.**binhex**(*input*, *output*)
将带有文件名 输入的二进制文件转换为 binhex 文件 输出。输出参数可以是文件名或类文件对象(`write()` 和 `close()` 方法的任何对象)。

binhex.**hexbin**(*input*[, *output*])
Decode a binhex file *input*. *input* may be a filename or a file-like object supporting `read()` and `close()` methods. The resulting file is written to a file named *output*, unless the argument is omitted in which case the output filename is read from the binhex file.

还定义了以下异常:

exception binhex.**Error**
当无法使用 binhex 格式编码某些内容时(例如,文件名太长而无法放入文件名字段中),或者输入未正确编码的 binhex 数据时,会引发异常。

参见:

模块*binascii* 支持模块,包含 ASCII 到二进制和二进制到 ASCII 转换。

18.13.1 注释

还有一个替代的、功能更强大的编码器和解码器接口,详细信息请参见源代码。

如果您在非 Macintosh 平台上编码或解码文本文件,它们仍将使用旧的 Macintosh 换行符约定(回车符作为行尾)。

As of this writing, *hexbin()* appears to not work in all cases.

18.14 binascii —二进制和 ASCII 码互转

binascii 模块包含很多在二进制和二进制表示的各种 ASCII 码之间转换的方法。通常情况不会直接使用这些函数,而是使用像*uu*, *base64*, 或*binhex* 这样的封装模块。为了执行效率高,*binascii* 模块含有许多用 C 写的低级函数,这些底层函数被一些高级模块所使用。

binascii 模块定义了以下函数:

binascii.**a2b_uu**(*string*)
将单行 uu 编码数据转换成二进制数据并返回。uu 编码每行的数据通常包含 45 个(二进制)字节,最后一行除外。每行数据后面可能跟有空格。

binascii.**b2a_uu**(*data*)
Convert binary data to a line of ASCII characters, the return value is the converted line, including a newline char. The length of *data* should be at most 45.

binascii.**a2b_base64**(*string*)
将 base64 数据块转换成二进制并以二进制数据形式返回。一次可以传递多行数据。

binascii.**b2a_base64**(*data*)
Convert binary data to a line of ASCII characters in base64 coding. The return value is the converted line, including a newline char. The newline is added because the original use case for this function was to feed it a series of 57 byte input lines to get output lines that conform to the MIME-base64 standard. Otherwise the output conforms to **RFC 3548**.

binascii.**a2b_qp**(*string*[, *header*])
将一个引号可打印的数据块转换成二进制数据并返回。一次可以转换多行。如果可选参数 *header* 存在且为 true，则数据中的下划线将被解码成空格。

binascii.**b2a_qp**(*data*[, *quotetabs*, *istext*, *header*])
Convert binary data to a line(s) of ASCII characters in quoted-printable encoding. The return value is the converted line(s). If the optional argument *quotetabs* is present and true, all tabs and spaces will be encoded. If the optional argument *istext* is present and true, newlines are not encoded but trailing whitespace will be encoded. If the optional argument *header* is present and true, spaces will be encoded as underscores per RFC1522. If the optional argument *header* is present and false, newline characters will be encoded as well; otherwise linefeed conversion might corrupt the binary data stream.

binascii.**a2b_hqx**(*string*)
将 binhex4 格式的 ASCII 数据不进行 RLE 解压缩直接转换为二进制数据。该字符串应包含完整数量的二进制字节，或者（在 binhex4 数据最后部分）剩余位为零。

binascii.**rledecode_hqx**(*data*)
根据 binhex4 标准对数据执行 RLE 解压缩。该算法在一个字节的数据后使用 `0x90` 作为重复指示符，然后计数。计数 `0` 指定字节值 `0x90` 。该例程返回解压缩的数据，输入数据以孤立的重复指示符结束的情况下，将引发 *Incomplete* 异常。

binascii.**rlecode_hqx**(*data*)
在 *data* 上执行 binhex4 游程编码压缩并返回结果。

binascii.**b2a_hqx**(*data*)
执行 hexbin4 类型二进制到 ASCII 码的转换并返回结果字符串。输入数据应经过 RLE 编码，且数据长度可被 3 整除（除了最后一个片段）。

binascii.**crc_hqx**(*data*, *crc*)
Compute a 16-bit CRC value of *data*, starting with an initial *crc* and returning the result. This uses the CRC-CCITT polynomial $x^{16} + x^{12} + x^5 + 1$, often represented as 0x1021. This CRC is used in the binhex4 format.

binascii.**crc32**(*data*[, *crc*])
Compute CRC-32, the 32-bit checksum of data, starting with an initial crc. This is consistent with the ZIP file checksum. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm. Use as follows:

```
print binascii.crc32("hello world")
# Or, in two pieces:
crc = binascii.crc32("hello")
crc = binascii.crc32(" world", crc) & 0xffffffff
print 'crc32 = 0x%08x' % crc
```

注解： To generate the same numeric value across all Python versions and platforms use crc32(data) & 0xffffffff. If you are only using the checksum in packed binary format this is not necessary as the return value is the correct 32bit binary representation regardless of sign.

在 2.6 版更改: The return value is in the range [-2**31, 2**31-1] regardless of platform. In the past the value would be signed on some platforms and unsigned on others. Use & 0xffffffff on the value if you want it to match Python 3 behavior.

在 3.0 版更改: The return value is unsigned and in the range [0, 2**32-1] regardless of platform.

binascii.**b2a_hex**(*data*)
binascii.**hexlify**(*data*)
Return the hexadecimal representation of the binary *data*. Every byte of *data* is converted into the corresponding 2-digit hex representation. The resulting string is therefore twice as long as the length of *data*.

binascii.**a2b_hex**(*hexstr*)
binascii.**unhexlify**(*hexstr*)
Return the binary data represented by the hexadecimal string *hexstr*. This function is the inverse of *b2a_hex()*. *hexstr* must contain an even number of hexadecimal digits (which can be upper or lower case), otherwise a *TypeError* is raised.

exception binascii.**Error**
通常是因为编程错误引发的异常。

exception binascii.**Incomplete**
数据不完整引发的异常。通常不是编程错误导致的，可以通过读取更多的数据并再次尝试来处理该异常。

参见:

模块*base64* Support for RFC compliant base64-style encoding in base 16, 32, and 64.

模块*binhex* 支持在 Macintosh 上使用的 binhex 格式。

模块*uu* 支持在 Unix 上使用的 UU 编码。

模块*quopri* 支持在 MIME 版本电子邮件中使用引号可打印编码。

18.15 quopri —编码与解码经过 MIME 转码的可打印数据

源代码: Lib/quopri.py

此模块会执行转换后可打印的传输编码与解码，具体定义见 **RFC 1521**: “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies”。转换后可打印的编码格式被设计用于只包含相对较少的不可打印字符的数据；如果存在大量这样的字符，通过*base64*模块所提供的 base64 编码方案会更为紧凑，例如当发送图片文件时。

quopri.**decode**(*input*, *output*[, *header*])
Decode the contents of the *input* file and write the resulting decoded binary data to the *output* file. *input* and *output* must either be file objects or objects that mimic the file object interface. *input* will be read until `input.readline()` returns an empty string. If the optional argument *header* is present and true, underscore will be decoded as space. This is used to decode “Q”-encoded headers as described in **RFC 1522**: “MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text” .

quopri.**encode**(*input*, *output*, *quotetabs*)
Encode the contents of the *input* file and write the resulting quoted-printable data to the *output* file. *input* and *output* must either be file objects or objects that mimic the file object interface. *input* will be read until `input.readline()` returns an empty string. *quotetabs* is a flag which controls whether to encode embedded spaces and tabs; when true it encodes such embedded whitespace, and when false it leaves them unencoded. Note that spaces and tabs appearing at the end of lines are always encoded, as per **RFC 1521**.

quopri.**decodestring**(*s*[, *header*])
Like *decode()*, except that it accepts a source string and returns the corresponding decoded string.

quopri.**encodestring**(*s*[, *quotetabs*])
Like *encode()*, except that it accepts a source string and returns the corresponding encoded string. *quotetabs* is optional (defaulting to 0), and is passed straight through to *encode()*.

参见:

Module *mimify* General utilities for processing of MIME messages.

模块*base64* 编码与解码 MIME base64 数据

18.16 uu —对 uuencode 文件进行编码与解码

源代码: Lib/uu.py

此模块使用 uuencode 格式来编码和解码文件，以便任意二进制数据可通过仅限 ASCII 码的连接进行传输。在任何要求文件参数的地方，这些方法都接受文件类对象。为了保持向下兼容，也接受包含路径名称的字符串，并且将打开相应的文件进行读写；路径名称 '-' 被解读为标准输入或输出。但是，此接口已被弃用；在 Windows 中调用者最好是自行打开文件，并在需要时确保模式为 'rb' or 'wb'。

此代码由 Lance Ellinghouse 贡献，并由 Jack Jansen 修改。

uu 模块定义了以下函数:

uu.**encode**(*in_file*, *out_file*[, *name*[, *mode*]])
Uuencode file *in_file* into file *out_file*. The uuencoded file will have the header specifying *name* and *mode* as the defaults for the results of decoding the file. The default defaults are taken from *in_file*, or '-' and 0666 respectively.

uu.**decode**(*in_file*[, *out_file*[, *mode*[, *quiet*]]])
调用此函数会解码 uuencod 编码的 *in_file* 文件并将结果放入 *out_file* 文件。如果 *out_file* 是一个路径名称，*mode* 会在必须创建文件时用于设置权限位。*out_file* 和 *mode* 的默认值会从 uuencode 标头中提取。但是，如果标头中指定的文件已存在，则会引发*uu.Error*。

如果输入由不正确的 uuencode 编码器生成，*decode()* 可能会打印一条警告到标准错误，这样 Python 可以从该错误中恢复。将 *quiet* 设为真值可以屏蔽此警告。

exception uu.**Error**
Exception 的子类，此异常可由*uu.decode()* 在多种情况下引发，如上文所述，此外还包括格式错误的标头或被截断的输入文件等。

参见:

模块*binascii* 支持模块，包含 ASCII 到二进制和二进制到 ASCII 转换。

CHAPTER 19

结构化标记处理工具

Python 支持各种模块，以处理各种形式的结构化数据标记。这包括使用标准通用标记语言（SGML）和超文本标记语言（HTML）的模块，以及使用可扩展标记语言（XML）的几个接口。

It is important to note that modules in the *xml* package require that there be at least one SAX-compliant XML parser available. Starting with Python 2.3, the Expat parser is included with Python, so the *xml.parsers.expat* module will always be available. You may still want to be aware of the PyXML add-on package; that package provides an extended set of XML libraries for Python.

The documentation for the *xml.dom* and *xml.sax* packages are the definition of the Python bindings for the DOM and SAX interfaces.

19.1 HTMLParser —Simple HTML and XHTML parser

注解： The *HTMLParser* module has been renamed to `html.parser` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

2.2 新版功能.

Source code: Lib/HTMLParser.py

This module defines a class *HTMLParser* which serves as the basis for parsing text files formatted in HTML (HyperText Mark-up Language) and XHTML. Unlike the parser in *htmllib*, this parser is not based on the SGML parser in *sgmllib*.

class HTMLParser.**HTMLParser**

An *HTMLParser* instance is fed HTML data and calls handler methods when start tags, end tags, text, comments, and other markup elements are encountered. The user should subclass *HTMLParser* and override its methods to implement the desired behavior.

The *HTMLParser* class is instantiated without arguments.

Unlike the parser in *htmllib*, this parser does not check that end tags match start tags or call the end-tag handler for elements which are closed implicitly by closing an outer element.

An exception is defined as well:

exception HTMLParser.**HTMLParseError**
HTMLParser is able to handle broken markup, but in some cases it might raise this exception when it encounters an error while parsing. This exception provides three attributes: `msg` is a brief message explaining the error, `lineno` is the number of the line on which the broken construct was detected, and `offset` is the number of characters into the line at which the construct starts.

19.1.1 Example HTML Parser Application

As a basic example, below is a simple HTML parser that uses the *HTMLParser* class to print out start tags, end tags and data as they are encountered:

```
from HTMLParser import HTMLParser

# create a subclass and override the handler methods
class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print "Encountered a start tag:", tag

    def handle_endtag(self, tag):
        print "Encountered an end tag :", tag

    def handle_data(self, data):
        print "Encountered some data  :", data

# instantiate the parser and fed it some HTML
parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
            '<body><h1>Parse me!</h1></body></html>')
```

The output will then be:

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data  : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data  : Parse me!
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```

19.1.2 HTMLParser Methods

HTMLParser instances have the following methods:

HTMLParser.**feed**(*data*)
Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or *close()* is called. *data* can be either *unicode* or *str*, but passing *unicode* is advised.

HTMLParser.**close**()
Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call the *HTMLParser* base class method *close()*.

HTMLParser.**reset**()
Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

HTMLParser.**getpos**()
Return current line number and offset.

HTMLParser.**get_starttag_text**()
Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML "as deployed" or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

The following methods are called when data or markup elements are encountered and they are meant to be overridden in a subclass. The base class implementations do nothing (except for *handle_startendtag()*):

HTMLParser.**handle_starttag**(*tag*, *attrs*)
This method is called to handle the start of a tag (e.g. `<div id="main">`).

The *tag* argument is the name of the tag converted to lower case. The *attrs* argument is a list of `(name, value)` pairs containing the attributes found inside the tag's <> brackets. The *name* will be translated to lower case, and quotes in the *value* have been removed, and character and entity references have been replaced.

For instance, for the tag `<A HREF="https://www.cwi.nl/">`, this method would be called as `handle_starttag('a', [('href', 'https://www.cwi.nl/')])`.

在 2.6 版更改: All entity references from *htmlentitydefs* are now replaced in the attribute values.

HTMLParser.**handle_endtag**(*tag*)
This method is called to handle the end tag of an element (e.g. `</div>`).

The *tag* argument is the name of the tag converted to lower case.

HTMLParser.**handle_startendtag**(*tag*, *attrs*)
Similar to *handle_starttag()*, but called when the parser encounters an XHTML-style empty tag (`<img ... />`). This method may be overridden by subclasses which require this particular lexical information; the default implementation simply calls *handle_starttag()* and *handle_endtag()*.

HTMLParser.**handle_data**(*data*)
This method is called to process arbitrary data (e.g. text nodes and the content of `<script>...</script>` and `<style>...</style>`).

HTMLParser.**handle_entityref**(*name*)
This method is called to process a named character reference of the form `&name;` (e.g. `&gt;`), where *name* is a general entity reference (e.g. `'gt'`).

HTMLParser.**handle_charref**(*name*)
This method is called to process decimal and hexadecimal numeric character references of the form `&#NNN;` and `&#xNNN;`. For example, the decimal equivalent for `&gt;` is `&#62;`, whereas the hexadecimal is `&#x3E;`; in this case the method will receive `'62'` or `'x3E'`.

HTMLParser.**handle_comment**(*data*)
This method is called when a comment is encountered (e.g. `<!--comment-->`).

For example, the comment `<!-- comment -->` will cause this method to be called with the argument `' comment '`.

The content of Internet Explorer conditional comments (condcoms) will also be sent to this method, so, for `<!--[if IE 9]>IE9-specific content<![endif]-->`, this method will receive `'[if IE 9]>IE9-specific content<![endif]'`.

HTMLParser.**handle_decl**(*decl*)
This method is called to handle an HTML doctype declaration (e.g. `<!DOCTYPE html>`).

The *decl* parameter will be the entire contents of the declaration inside the `<!...>` markup (e.g. `'DOCTYPE html'`).

HTMLParser.**handle_pi**(*data*)
This method is called when a processing instruction is encountered. The *data* parameter will contain the entire processing instruction. For example, for the processing instruction `<?proc color='red'>`, this method would be called as `handle_pi("proc color='red'")`.

注解： The *HTMLParser* class uses the SGML syntactic rules for processing instructions. An XHTML processing instruction using the trailing `'?'` will cause the `'?'` to be included in *data*.

HTMLParser.**unknown_decl**(*data*)
This method is called when an unrecognized declaration is read by the parser.

The *data* parameter will be the entire contents of the declaration inside the `<![...]>` markup. It is sometimes useful to be overridden by a derived class.

19.1.3 Examples

The following class implements a parser that will be used to illustrate more examples:

```
from HTMLParser import HTMLParser
from htmlentitydefs import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print "Start tag:", tag
        for attr in attrs:
            print "     attr:", attr

    def handle_endtag(self, tag):
        print "End tag  :", tag

    def handle_data(self, data):
        print "Data     :", data

    def handle_comment(self, data):
        print "Comment  :", data

    def handle_entityref(self, name):
        c = unichr(name2codepoint[name])
        print "Named ent:", c
```

(下页继续)

(续上页)

```
    def handle_charref(self, name):
        if name.startswith('x'):
            c = unichr(int(name[1:], 16))
        else:
            c = unichr(int(name))
        print "Num ent  :", c

    def handle_decl(self, data):
        print "Decl     :", data

parser = MyHTMLParser()
```

Parsing a doctype:

```
>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl     : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
↪html4/strict.dtd"
```

Parsing an element with a few attributes and a title:

```
>>> parser.feed('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
     attr: ('src', 'python-logo.png')
     attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data     : Python
End tag  : h1
```

The content of `script` and `style` elements is returned as is, without further parsing:

```
>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
     attr: ('type', 'text/css')
Data     : #python { color: green }
End tag  : style

>>> parser.feed('<script type="text/javascript">'
...             'alert("<strong>hello!</strong>");</script>')
Start tag: script
     attr: ('type', 'text/javascript')
Data     : alert("<strong>hello!</strong>");
End tag  : script
```

Parsing comments:

```
>>> parser.feed('<!-- a comment -->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment  :  a comment
Comment  : [if IE 9]>IE-specific content<![endif]
```

Parsing named and numeric character references and converting them to the correct char (note: these 3 references are all equivalent to `'>'`):

```
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent  : >
Num ent  : >
```

Feeding incomplete chunks to *feed()* works, but *handle_data()* might be called more than once:

```
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', 'pan>']:
...     parser.feed(chunk)
...
Start tag: span
Data     : buff
Data     : ered
Data     : text
End tag  : span
```

Parsing invalid HTML (e.g. unquoted attributes) also works:

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
     attr: ('class', 'link')
     attr: ('href', '#main')
Data     : tag soup
End tag  : p
End tag  : a
```

19.2 sgmllib —Simple SGML parser

2.6 版后已移除: The *sgmllib* module has been removed in Python 3.

This module defines a class *SGMLParser* which serves as the basis for parsing text files formatted in SGML (Standard Generalized Mark-up Language). In fact, it does not provide a full SGML parser —it only parses SGML insofar as it is used by HTML, and the module only exists as a base for the *htmllib* module. Another HTML parser which supports XHTML and offers a somewhat different interface is available in the *HTMLParser* module.

class sgmllib.**SGMLParser**

The *SGMLParser* class is instantiated without arguments. The parser is hardcoded to recognize the following constructs:

- Opening and closing tags of the form `<tag attr="value" ...>` and `</tag>`, respectively.
- Numeric character references of the form `&#name;`.
- Entity references of the form `&name;`.
- SGML comments of the form `<!--text-->`. Note that spaces, tabs, and newlines are allowed between the trailing > and the immediately preceding `--`.

A single exception is defined as well:

exception sgmllib.**SGMLParseError**

Exception raised by the *SGMLParser* class when it encounters an error while parsing.

2.1 新版功能.

SGMLParser instances have the following methods:

SGMLParser.**reset**()
Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

SGMLParser.**setnomoretags**()
Stop processing tags. Treat all following input as literal input (CDATA). (This is only provided so the HTML tag <PLAINTEXT> can be implemented.)

SGMLParser.**setliteral**()
Enter literal mode (CDATA mode).

SGMLParser.**feed**(*data*)
Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or *close()* is called.

SGMLParser.**close**()
Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call *close()*.

SGMLParser.**get_starttag_text**()
Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML "as deployed" or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

SGMLParser.**handle_starttag**(*tag*, *method*, *attributes*)
This method is called to handle start tags for which either a start_tag() or do_tag() method has been defined. The *tag* argument is the name of the tag converted to lower case, and the *method* argument is the bound method which should be used to support semantic interpretation of the start tag. The *attributes* argument is a list of (name, value) pairs containing the attributes found inside the tag's <> brackets.

The *name* has been translated to lower case. Double quotes and backslashes in the *value* have been interpreted, as well as known character references and known entity references terminated by a semicolon (normally, entity references can be terminated by any non-alphanumerical character, but this would break the very common case of <A HREF="url?spam=1&eggs=2"> when eggs is a valid entity name).

For instance, for the tag <A HREF="http://www.cwi.nl/">, this method would be called as unknown_starttag('a', [('href', 'http://www.cwi.nl/')]). The base implementation simply calls *method* with *attributes* as the only argument.

2.5 新版功能: Handling of entity and character references within attribute values.

SGMLParser.**handle_endtag**(*tag*, *method*)
This method is called to handle endtags for which an end_tag() method has been defined. The *tag* argument is the name of the tag converted to lower case, and the *method* argument is the bound method which should be used to support semantic interpretation of the end tag. If no end_tag() method is defined for the closing element, this handler is not called. The base implementation simply calls *method*.

SGMLParser.**handle_data**(*data*)
This method is called to process arbitrary data. It is intended to be overridden by a derived class; the base class implementation does nothing.

SGMLParser.**handle_charref**(*ref*)
This method is called to process a character reference of the form &#ref;. The base implementation uses *convert_charref()* to convert the reference to a string. If that method returns a string, it is passed to *handle_data()*, otherwise unknown_charref(ref) is called to handle the error.

在 2.5 版更改: Use *convert_charref()* instead of hard-coding the conversion.

SGMLParser.**convert_charref**(*ref*)
Convert a character reference to a string, or None. *ref* is the reference passed in as a string. In the base implementation, *ref* must be a decimal number in the range 0–255. It converts the code point found using the

convert_codepoint() method. If *ref* is invalid or out of range, this method returns None. This method is called by the default *handle_charref()* implementation and by the attribute value parser.

2.5 新版功能.

SGMLParser.**convert_codepoint**(*codepoint*)
Convert a code point to a *str* value. Encodings can be handled here if appropriate, though the rest of *sgmllib* is oblivious on this matter.

2.5 新版功能.

SGMLParser.**handle_entityref**(*ref*)
This method is called to process a general entity reference of the form &ref; where *ref* is a general entity reference. It converts *ref* by passing it to *convert_entityref()*. If a translation is returned, it calls the method *handle_data()* with the translation; otherwise, it calls the method unknown_entityref(ref). The default entitydefs defines translations for &, ', >, <, and ".

在 2.5 版更改: Use *convert_entityref()* instead of hard-coding the conversion.

SGMLParser.**convert_entityref**(*ref*)
Convert a named entity reference to a *str* value, or None. The resulting value will not be parsed. *ref* will be only the name of the entity. The default implementation looks for *ref* in the instance (or class) variable entitydefs which should be a mapping from entity names to corresponding translations. If no translation is available for *ref*, this method returns None. This method is called by the default *handle_entityref()* implementation and by the attribute value parser.

2.5 新版功能.

SGMLParser.**handle_comment**(*comment*)
This method is called when a comment is encountered. The *comment* argument is a string containing the text between the <!-- and --> delimiters, but not the delimiters themselves. For example, the comment <!--text--> will cause this method to be called with the argument 'text'. The default method does nothing.

SGMLParser.**handle_decl**(*data*)
Method called when an SGML declaration is read by the parser. In practice, the DOCTYPE declaration is the only thing observed in HTML, but the parser does not discriminate among different (or broken) declarations. Internal subsets in a DOCTYPE declaration are not supported. The *data* parameter will be the entire contents of the declaration inside the <!…> markup. The default implementation does nothing.

SGMLParser.**report_unbalanced**(*tag*)
This method is called when an end tag is found which does not correspond to any open element.

SGMLParser.**unknown_starttag**(*tag*, *attributes*)
This method is called to process an unknown start tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

SGMLParser.**unknown_endtag**(*tag*)
This method is called to process an unknown end tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

SGMLParser.**unknown_charref**(*ref*)
This method is called to process unresolvable numeric character references. Refer to *handle_charref()* to determine what is handled by default. It is intended to be overridden by a derived class; the base class implementation does nothing.

SGMLParser.**unknown_entityref**(*ref*)
This method is called to process an unknown entity reference. It is intended to be overridden by a derived class; the base class implementation does nothing.

Apart from overriding or extending the methods listed above, derived classes may also define methods of the following form to define processing of specific tags. Tag names in the input stream are case independent; the *tag* occurring in

method names must be in lower case:

SGMLParser.**start_tag**(*attributes*)
This method is called to process an opening tag *tag*. It has preference over do_tag(). The *attributes* argument has the same meaning as described for *handle_starttag()* above.

SGMLParser.**do_tag**(*attributes*)
This method is called to process an opening tag *tag* for which no start_tag() method is defined. The *attributes* argument has the same meaning as described for *handle_starttag()* above.

SGMLParser.**end_tag**()
This method is called to process a closing tag *tag*.

Note that the parser maintains a stack of open elements for which no end tag has been found yet. Only tags processed by start_tag() are pushed on this stack. Definition of an end_tag() method is optional for these tags. For tags processed by do_tag() or by unknown_tag(), no end_tag() method must be defined; if defined, it will not be used. If both start_tag() and do_tag() methods exist for a tag, the start_tag() method takes precedence.

19.3 htmllib —A parser for HTML documents

2.6 版后已移除: The *htmllib* module has been removed in Python 3. Use *HTMLParser* instead in Python 2, and the equivalent, html.parser, in Python 3.

This module defines a class which can serve as a base for parsing text files formatted in the HyperText Mark-up Language (HTML). The class is not directly concerned with I/O —it must be provided with input in string form via a method, and makes calls to methods of a "formatter" object in order to produce output. The *HTMLParser* class is designed to be used as a base class for other classes in order to add functionality, and allows most of its methods to be extended or overridden. In turn, this class is derived from and extends the SGMLParser class defined in module *sgmllib*. The *HTMLParser* implementation supports the HTML 2.0 language as described in **RFC 1866**. Two implementations of formatter objects are provided in the *formatter* module; refer to the documentation for that module for information on the formatter interface.

The following is a summary of the interface defined by *sgmllib.SGMLParser*:

- The interface to feed data to an instance is through the feed() method, which takes a string argument. This can be called with as little or as much text at a time as desired; p.feed(a); p.feed(b) has the same effect as p.feed(a+b). When the data contains complete HTML markup constructs, these are processed immediately; incomplete constructs are saved in a buffer. To force processing of all unprocessed data, call the close() method.

  For example, to parse the entire contents of a file, use:

```
parser.feed(open('myfile.html').read())
parser.close()
```

- The interface to define semantics for HTML tags is very simple: derive a class and define methods called start_tag(), end_tag(), or do_tag(). The parser will call these at appropriate moments: start_tag() or do_tag() is called when an opening tag of the form <tag ...> is encountered; end_tag() is called when a closing tag of the form <tag> is encountered. If an opening tag requires a corresponding closing tag, like <H1> ···</H1>, the class should define the start_tag() method; if a tag requires no closing tag, like <P>, the class should define the do_tag() method.

The module defines a parser class and an exception:

class htmllib.**HTMLParser**(*formatter*)
This is the basic HTML parser class. It supports all entity names required by the XHTML 1.0 Recommendation (https://www.w3.org/TR/xhtml1). It also defines handlers for all HTML 2.0 and many HTML 3.0 and 3.2 elements.

exception htmllib.**HTMLParseError**
Exception raised by the *HTMLParser* class when it encounters an error while parsing.

2.4 新版功能.

参见:

Module ***formatter*** Interface definition for transforming an abstract flow of formatting events into specific output events on writer objects.

Module ***HTMLParser*** Alternate HTML parser that offers a slightly lower-level view of the input, but is designed to work with XHTML, and does not implement some of the SGML syntax not used in "HTML as deployed" and which isn't legal for XHTML.

Module ***htmlentitydefs*** Definition of replacement text for XHTML 1.0 entities.

Module ***sgmllib*** Base class for *HTMLParser*.

19.3.1 HTMLParser Objects

In addition to tag methods, the *HTMLParser* class provides some additional methods and instance variables for use within tag methods.

HTMLParser.**formatter**
This is the formatter instance associated with the parser.

HTMLParser.**nofill**
Boolean flag which should be true when whitespace should not be collapsed, or false when it should be. In general, this should only be true when character data is to be treated as "preformatted" text, as within a <PRE> element. The default value is false. This affects the operation of handle_data() and *save_end()*.

HTMLParser.**anchor_bgn**(*href*, *name*, *type*)
This method is called at the start of an anchor region. The arguments correspond to the attributes of the <A> tag with the same names. The default implementation maintains a list of hyperlinks (defined by the HREF attribute for <A> tags) within the document. The list of hyperlinks is available as the data attribute anchorlist.

HTMLParser.**anchor_end**()
This method is called at the end of an anchor region. The default implementation adds a textual footnote marker using an index into the list of hyperlinks created by *anchor_bgn()*.

HTMLParser.**handle_image**(*source*, *alt*[, *ismap*[, *align*[, *width*[, *height*]]]])
This method is called to handle images. The default implementation simply passes the *alt* value to the handle_data() method.

HTMLParser.**save_bgn**()
Begins saving character data in a buffer instead of sending it to the formatter object. Retrieve the stored data via *save_end()*. Use of the *save_bgn()* / *save_end()* pair may not be nested.

HTMLParser.**save_end**()
Ends buffering character data and returns all data saved since the preceding call to *save_bgn()*. If the *nofill* flag is false, whitespace is collapsed to single spaces. A call to this method without a preceding call to *save_bgn()* will raise a *TypeError* exception.

19.4 htmlentitydefs —Definitions of HTML general entities

注解: The *htmlentitydefs* module has been renamed to `html.entities` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/htmlentitydefs.py

This module defines three dictionaries, `name2codepoint`, `codepoint2name`, and `entitydefs`. `entitydefs` is used by the *htmllib* module to provide the *entitydefs* attribute of the *HTMLParser* class. The definition provided here contains all the entities defined by XHTML 1.0 that can be handled using simple textual substitution in the Latin-1 character set (ISO-8859-1).

`htmlentitydefs.`**`entitydefs`**
A dictionary mapping XHTML 1.0 entity definitions to their replacement text in ISO Latin-1.

`htmlentitydefs.`**`name2codepoint`**
A dictionary that maps HTML entity names to the Unicode code points.

2.3 新版功能.

`htmlentitydefs.`**`codepoint2name`**
A dictionary that maps Unicode code points to HTML entity names.

2.3 新版功能.

19.5 XML 处理模块

用于处理 XML 的 Python 接口分组在 `xml` 包中。

警告: The XML modules are not secure against erroneous or maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML 漏洞*.

值得注意的是*xml* 包中的模块要求至少有一个 SAX 兼容的 XML 解析器可用。在 Pythonm 中包含 Expat 解析器，因此*xml.parsers.expat* 模块将始终可用。

xml.dom 和*xml.sax* 包的文档是 DOM 和 SAX 接口的 Python 绑定的定义。

XML 处理子模块包括:

- *xml.etree.ElementTree*：ElementTree API，一个简单而轻量级的 XML 处理器
- *xml.dom*：DOM API 定义
- *xml.dom.minidom*：最小的 DOM 实现
- *xml.dom.pulldom*：支持构建部分 DOM 树
- *xml.sax*：SAX2 基类和便利函数
- *xml.parsers.expat*：Expat 解析器绑定

19.6 XML 漏洞

The XML processing modules are not secure against maliciously constructed data. An attacker can abuse vulnerabilities for e.g. denial of service attacks, to access local files, to generate network connections to other machines, or to or circumvent firewalls. The attacks on XML abuse unfamiliar features like inline DTD (document type definition) with entities.

The following table gives an overview of the known attacks and if the various modules are vulnerable to them.

种类	sax	etree	minidom	pulldom	xmlrpc
billion laughs	**易受攻击**	**易受攻击**	**易受攻击**	**易受攻击**	**易受攻击**
quadratic blowup	**易受攻击**	**易受攻击**	**易受攻击**	**易受攻击**	**易受攻击**
external entity expansion	**易受攻击**	安全 (1)	安全 (2)	**易受攻击**	安全 (3)
DTD retrieval	**易受攻击**	安全	安全	**易受攻击**	安全
decompression bomb	安全	安全	安全	安全	**易受攻击**

1. *xml.etree.ElementTree* doesn't expand external entities and raises a ParserError when an entity occurs.
2. *xml.dom.minidom* 不会扩展外部实体，只是简单地返回未扩展的实体。
3. *xmlrpclib* 不扩展外部实体并省略它们。

billion laughs / exponential entity expansion （狂笑/递归实体扩展） The Billion Laughs attack –also known as exponential entity expansion –uses multiple levels of nested entities. Each entity refers to another entity several times, the final entity definition contains a small string. Eventually the small string is expanded to several gigabytes. The exponential expansion consumes lots of CPU time, too.

quadratic blowup entity expansion（二次爆炸实体扩展） A quadratic blowup attack is similar to a Billion Laughs attack; it abuses entity expansion, too. Instead of nested entities it repeats one large entity with a couple of thousand chars over and over again. The attack isn't as efficient as the exponential case but it avoids triggering countermeasures of parsers against heavily nested entities.

external entity expansion Entity declarations can contain more than just text for replacement. They can also point to external resources by public identifiers or system identifiers. System identifiers are standard URIs or can refer to local files. The XML parser retrieves the resource with e.g. HTTP or FTP requests and embeds the content into the XML document.

DTD retrieval Python 的一些 XML 库*xml.dom.pulldom* 从远程或本地位置检索文档类型定义。该功能与外部实体扩展问题具有相似的含义。

decompression bomb The issue of decompression bombs (aka ZIP bomb) apply to all XML libraries that can parse compressed XML stream like gzipped HTTP streams or LZMA-ed files. For an attacker it can reduce the amount of transmitted data by three magnitudes or more.

The documentation of defusedxml on PyPI has further information about all known attack vectors with examples and references.

19.6.1 defused packages

These external packages are recommended for any code that parses untrusted XML data.

defusedxml is a pure Python package with modified subclasses of all stdlib XML parsers that prevent any potentially malicious operation. The package also ships with example exploits and extended documentation on more XML exploits like xpath injection.

defusedexpat provides a modified libexpat and patched replacement `pyexpat` extension module with countermeasures against entity expansion DoS attacks. Defusedexpat still allows a sane and configurable amount of entity expansions. The modifications will be merged into future releases of Python.

The workarounds and modifications are not included in patch releases as they break backward compatibility. After all inline DTD and entity expansion are well-defined XML features.

19.7 xml.etree.ElementTree —ElementTree XML API

2.5 新版功能.

源代码： Lib/xml/etree/ElementTree.py

The *Element* type is a flexible container object, designed to store hierarchical data structures in memory. The type can be described as a cross between a list and a dictionary.

> **警告：** *xml.etree.ElementTree* 模块对于恶意构建的数据是不安全的。如果需要解析不可信或未经身份验证的数据，请参见*XML 漏洞* 。

Each element has a number of properties associated with it:

- a tag which is a string identifying what kind of data this element represents (the element type, in other words).
- a number of attributes, stored in a Python dictionary.
- a text string.
- an optional tail string.
- a number of child elements, stored in a Python sequence

To create an element instance, use the *Element* constructor or the *SubElement()* factory function.

The *ElementTree* class can be used to wrap an element structure, and convert it from and to XML.

A C implementation of this API is available as `xml.etree.cElementTree`.

See http://effbot.org/zone/element-index.htm for tutorials and links to other docs. Fredrik Lundh's page is also the location of the development version of the xml.etree.ElementTree.

在 2.7 版更改: The ElementTree API is updated to 1.3. For more information, see Introducing ElementTree 1.3.

19.7.1 教程

这是一个使用*xml.etree.ElementTree* （简称 `ET` ）的简短教程。目标是演示模块的一些构建块和基本概念。

XML 树和元素

XML 是一种固有的分层数据格式，最自然的表示方法是使用树。为此，`ET` 有两个类*ElementTree* 将整个 XML 文档表示为一个树，*Element* 表示该树中的单个节点。与整个文档的交互（读写文件）通常在*ElementTree* 级别完成。与单个 XML 元素及其子元素的交互是在*Element* 级别完成的。

解析 XML

我们将使用以下 XML 文档作为本节的示例数据：

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

We have a number of ways to import the data. Reading the file from disk:

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

Reading the data from a string:

```
root = ET.fromstring(country_data_as_string)
```

fromstring() 将 XML 从字符串直接解析为*Element*，该元素是已解析树的根元素。其他解析函数可能会创建一个*ElementTree*。确切的信息请检查文档。

作为一个*Element*，root 有一个标记和一个属性字典：

```
>>> root.tag
'data'
>>> root.attrib
{}
```

它还有我们可以迭代的子节点：

```
>>> for child in root:
...     print child.tag, child.attrib
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

子级是可以嵌套的，我们可以通过索引访问特定的子级节点：

```
>>> root[0][1].text
'2008'
```

寻找有趣的元素

Element 有一些很有效的方法，可帮助递归遍历其下的所有子树（包括子级，子级的子级，等等）。例如*Element.iter()*:

```
>>> for neighbor in root.iter('neighbor'):
...     print neighbor.attrib
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

Element.findall() 仅查找当前元素的直接子元素中带有指定标签的元素。*Element.find()* 找带有特定标签的 第一个子级，然后可以用*Element.text* 访问元素的文本内容。*Element.text* 访问元素的属性:

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print name, rank
...
Liechtenstein 1
Singapore 4
Panama 68
```

通过使用*XPath* ，可以更精确地指定要查找的元素。

修改 XML 文件

ElementTree 提供了一种构建 XML 文档并将其写入文件的简单方法。*ElementTree.write()* 方法可达到此目的。

创建后可以直接操作*Element* 对象。例如：使用*Element.text* 修改文本字段，使用*Element.set()* 方法添加和修改属性，以及使用*Element.append()* 添加新的子元素。

假设我们要在每个国家/地区的中添加一个排名，并在 rank 元素中添加一个 `updated` 属性：

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

生成的 XML 现在看起来像这样：

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
```

(下页继续)

(续上页)

```
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank updated="yes">69</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

可以使用*Element.remove()* 删除元素。假设我们要删除排名高于 50 的所有国家/地区:

```
>>> for country in root.findall('country'):
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

生成的 XML 现在看起来像这样:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>
```

构建 XML 文档

SubElement() 函数还提供了一种便捷方法来为给定元素创建新的子元素:

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

使用命名空间解析 XML

If the XML input has namespaces, tags and attributes with prefixes in the form `prefix:sometag` get expanded to `{uri}sometag` where the *prefix* is replaced by the full *URI*. Also, if there is a default namespace, that full URI gets prepended to all of the non-prefixed tags.

Here is an XML example that incorporates two namespaces, one with the prefix "fictional" and the other serving as the default namespace:

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
        xmlns="http://people.example.com">
    <actor>
        <name>John Cleese</name>
        <fictional:character>Lancelot</fictional:character>
        <fictional:character>Archie Leach</fictional:character>
    </actor>
    <actor>
        <name>Eric Idle</name>
        <fictional:character>Sir Robin</fictional:character>
        <fictional:character>Gunther</fictional:character>
        <fictional:character>Commander Clement</fictional:character>
    </actor>
</actors>
```

One way to search and explore this XML example is to manually add the URI to every tag or attribute in the xpath of a *find()* or *findall()*:

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print name.text
    for char in actor.findall('{http://characters.example.com}character'):
        print ' |-->', char.text
```

A better way to search the namespaced XML example is to create a dictionary with your own prefixes and use those in the search functions:

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print name.text
    for char in actor.findall('role:character', ns):
        print ' |-->', char.text
```

These two approaches both output:

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

其他资源

See http://effbot.org/zone/element-index.htm for tutorials and links to other docs.

19.7.2 XPath 支持

This module provides limited support for XPath expressions for locating elements in a tree. The goal is to support a small subset of the abbreviated syntax; a full XPath engine is outside the scope of the module.

示例

Here's an example that demonstrates some of the XPath capabilities of the module. We'll be using the `countrydata` XML document from the *Parsing XML* section:

```
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall(".//year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall(".//*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their parent
root.findall(".//neighbor[2]")
```

支持的 XPath 语法

语法	含义
`tag`	Selects all child elements with the given tag. For example, `spam` selects all child elements named `spam`, and `spam/egg` selects all grandchildren named `egg` in all children named `spam`.
`*`	Selects all child elements. For example, `*/egg` selects all grandchildren named `egg`.
`.`	选择当前节点。这在路径的开头非常有用，用于指示它是相对路径。
`//`	Selects all subelements, on all levels beneath the current element. For example, `.//egg` selects all `egg` elements in the entire tree.
`..`	Selects the parent element.
`[@attrib]`	选择具有给定属性的所有元素。
`[@attrib='value']`	选择给定属性具有给定值的所有元素。该值不能包含引号。
`[tag]`	选择所有包含 `tag` 子元素的元素。只支持直系子元素。
`[tag='text']`	选择所有包含名为 `tag` 的子元素的元素，这些子元素（包括后代）的完整文本内容等于给定的 `text` 。
`[position]`	Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression `last()` (for the last position), or a position relative to the last position (e.g. `last()-1`).

谓词（方括号内的表达式）之前必须带有标签名称，星号或其他谓词。`position` 谓词前必须有标签名称。

19.7.3 参考

函数

`xml.etree.ElementTree.`**`Comment`**(*text=None*)
Comment element factory. This factory function creates a special element that will be serialized as an XML comment by the standard serializer. The comment string can be either a bytestring or a Unicode string. *text* is a string containing the comment string. Returns an element instance representing a comment.

`xml.etree.ElementTree.`**`dump`**(*elem*)
Writes an element tree or element structure to sys.stdout. This function should be used for debugging only.

The exact output format is implementation dependent. In this version, it's written as an ordinary XML file.

elem is an element tree or an individual element.

`xml.etree.ElementTree.`**`fromstring`**(*text*)
Parses an XML section from a string constant. Same as *XML()*. *text* is a string containing XML data. Returns an *Element* instance.

`xml.etree.ElementTree.`**`fromstringlist`**(*sequence*, *parser=None*)
Parses an XML document from a sequence of string fragments. *sequence* is a list or other sequence containing XML data fragments. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns an *Element* instance.

2.7 新版功能.

`xml.etree.ElementTree.`**`iselement`**(*element*)
Checks if an object appears to be a valid element object. *element* is an element instance. Returns a true value if this is an element object.

`xml.etree.ElementTree.`**`iterparse`**(*source*, *events=None*, *parser=None*)
Parses an XML section into an element tree incrementally, and reports what's going on to the user. *source* is a

filename or file object containing XML data. *events* is a list of events to report back. If omitted, only "end" events are reported. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. *parser* is not supported by `cElementTree`. Returns an *iterator* providing `(event, elem)` pairs.

注解: `iterparse()` only guarantees that it has seen the ">" character of a starting tag when it emits a "start" event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.

If you need a fully populated element, look for "end" events instead.

`xml.etree.ElementTree.`**`parse`**(*source*, *parser=None*)
Parses an XML section into an element tree. *source* is a filename or file object containing XML data. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `ElementTree` instance.

`xml.etree.ElementTree.`**`ProcessingInstruction`**(*target*, *text=None*)
PI element factory. This factory function creates a special element that will be serialized as an XML processing instruction. *target* is a string containing the PI target. *text* is a string containing the PI contents, if given. Returns an element instance, representing a processing instruction.

`xml.etree.ElementTree.`**`register_namespace`**(*prefix*, *uri*)
Registers a namespace prefix. The registry is global, and any existing mapping for either the given prefix or the namespace URI will be removed. *prefix* is a namespace prefix. *uri* is a namespace uri. Tags and attributes in this namespace will be serialized with the given prefix, if at all possible.

2.7 新版功能.

`xml.etree.ElementTree.`**`SubElement`**(*parent*, *tag*, *attrib={}*, ***extra*)
Subelement factory. This function creates an element instance, and appends it to an existing element.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *parent* is the parent element. *tag* is the subelement name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments. Returns an element instance.

`xml.etree.ElementTree.`**`tostring`**(*element*, *encoding="us-ascii"*, *method="xml"*)
Generates a string representation of an XML element, including all subelements. *element* is an `Element` instance. *encoding*[1] is the output encoding (default is US-ASCII). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). Returns an encoded string containing the XML data.

`xml.etree.ElementTree.`**`tostringlist`**(*element*, *encoding="us-ascii"*, *method="xml"*)
Generates a string representation of an XML element, including all subelements. *element* is an `Element` instance. *encoding*[1] is the output encoding (default is US-ASCII). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). Returns a list of encoded strings containing the XML data. It does not guarantee any specific sequence, except that `"".join(tostringlist(element)) == tostring(element)`.

2.7 新版功能.

`xml.etree.ElementTree.`**`XML`**(*text*, *parser=None*)
Parses an XML section from a string constant. This function can be used to embed "XML literals" in Python code. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `Element` instance.

`xml.etree.ElementTree.`**`XMLID`**(*text*, *parser=None*)
Parses an XML section from a string constant, and also returns a dictionary which maps from element id:s to

[1] The encoding string included in XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

elements. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns a tuple containing an *Element* instance and a dictionary.

元素对象

class xml.etree.ElementTree.**Element**(*tag*, *attrib={}*, ***extra*)

Element class. This class defines the Element interface, and provides a reference implementation of this interface.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *tag* is the element name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments.

tag

一个标识此元素意味着何种数据的字符串 (换句话说，元素类型)。

text

tail

These attributes can be used to hold additional data associated with the element. Their values are usually strings but may be any application-specific object. If the element is created from an XML file, the *text* attribute holds either the text between the element's start tag and its first child or end tag, or None, and the *tail* attribute holds either the text between the element's end tag and the next tag, or None. For the XML data

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

the *a* element has None for both *text* and *tail* attributes, the *b* element has *text* "1" and *tail* "4", the *c* element has *text* "2" and *tail* None, and the *d* element has *text* None and *tail* "3".

To collect the inner text of an element, see *itertext()*, for example "".join(element.itertext()).

Applications may store arbitrary objects in these attributes.

attrib

A dictionary containing the element's attributes. Note that while the *attrib* value is always a real mutable Python dictionary, an ElementTree implementation may choose to use another internal representation, and create the dictionary only if someone asks for it. To take advantage of such implementations, use the dictionary methods below whenever possible.

The following dictionary-like methods work on the element attributes.

clear()

Resets an element. This function removes all subelements, clears all attributes, and sets the text and tail attributes to None.

get(*key*, *default=None*)

Gets the element attribute named *key*.

Returns the attribute value, or *default* if the attribute was not found.

items()

Returns the element attributes as a sequence of (name, value) pairs. The attributes are returned in an arbitrary order.

keys()

Returns the elements attribute names as a list. The names are returned in an arbitrary order.

set(*key*, *value*)

Set the attribute *key* on the element to *value*.

The following methods work on the element's children (subelements).

append(*subelement*)
Adds the element *subelement* to the end of this elements internal list of subelements.

extend(*subelements*)
Appends *subelements* from a sequence object with zero or more elements. Raises `AssertionError` if a subelement is not a valid object.

2.7 新版功能.

find(*match*)
Finds the first subelement matching *match*. *match* may be a tag name or path. Returns an element instance or `None`.

findall(*match*)
Finds all matching subelements, by tag name or path. Returns a list containing all matching elements in document order.

findtext(*match*, *default=None*)
Finds text for the first subelement matching *match*. *match* may be a tag name or path. Returns the text content of the first matching element, or *default* if no element was found. Note that if the matching element has no text content an empty string is returned.

getchildren()
2.7 版后已移除: Use `list(elem)` or iteration.

getiterator(*tag=None*)
2.7 版后已移除: Use method `Element.iter()` instead.

insert(*index*, *element*)
Inserts a subelement at the given position in this element.

iter(*tag=None*)
Creates a tree *iterator* with the current element as the root. The iterator iterates over this element and all elements below it, in document (depth first) order. If *tag* is not `None` or `'*'`, only elements whose tag equals *tag* are returned from the iterator. If the tree structure is modified during iteration, the result is undefined.

2.7 新版功能.

iterfind(*match*)
Finds all matching subelements, by tag name or path. Returns an iterable yielding all matching elements in document order.

2.7 新版功能.

itertext()
Creates a text iterator. The iterator loops over this element and all subelements, in document order, and returns all inner text.

2.7 新版功能.

makeelement(*tag*, *attrib*)
Creates a new element object of the same type as this element. Do not call this method, use the `SubElement()` factory function instead.

remove(*subelement*)
Removes *subelement* from the element. Unlike the find* methods this method compares elements based on the instance identity, not on tag value or contents.

`Element` objects also support the following sequence type methods for working with subelements: `__delitem__()`, `__getitem__()`, `__setitem__()`, `__len__()`.

Caution: Elements with no subelements will test as `False`. This behavior will change in future versions. Use specific `len(elem)` or `elem is None` test instead.

```
element = root.find('foo')

if not element:  # careful!
    print "element not found, or element has no subelements"

if element is None:
    print "element not found"
```

ElementTree 对象

class xml.etree.ElementTree.**ElementTree**(*element=None*, *file=None*)

ElementTree wrapper class. This class represents an entire element hierarchy, and adds some extra support for serialization to and from standard XML.

element is the root element. The tree is initialized with the contents of the XML *file* if given.

_setroot(*element*)

Replaces the root element for this tree. This discards the current contents of the tree, and replaces it with the given element. Use with care. *element* is an element instance.

find(*match*)

Same as *Element.find()*, starting at the root of the tree.

findall(*match*)

Same as *Element.findall()*, starting at the root of the tree.

findtext(*match*, *default=None*)

Same as *Element.findtext()*, starting at the root of the tree.

getiterator(*tag=None*)

2.7 版后已移除: Use method *ElementTree.iter()* instead.

getroot()

Returns the root element for this tree.

iter(*tag=None*)

Creates and returns a tree iterator for the root element. The iterator loops over all elements in this tree, in section order. *tag* is the tag to look for (default is to return all elements).

iterfind(*match*)

Finds all matching subelements, by tag name or path. Same as getroot().iterfind(match). Returns an iterable yielding all matching elements in document order.

2.7 新版功能.

parse(*source*, *parser=None*)

Loads an external XML section into this element tree. *source* is a file name or file object. *parser* is an optional parser instance. If not given, the standard XMLParser parser is used. Returns the section root element.

write(*file*, *encoding="us-ascii"*, *xml_declaration=None*, *default_namespace=None*, *method="xml"*)

Writes the element tree to a file, as XML. *file* is a file name, or a file object opened for writing. *encoding*[1] is the output encoding (default is US-ASCII). *xml_declaration* controls if an XML declaration should be added to the file. Use `False` for never, `True` for always, `None` for only if not US-ASCII or UTF-8 (default is `None`). *default_namespace* sets the default XML namespace (for "xmlns"). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). Returns an encoded string.

This is the XML file that is going to be manipulated:

```
<html>
    <head>
        <title>Example page</title>
    </head>
    <body>
        <p>Moved to <a href="http://example.org/">example.org</a>
        or <a href="http://example.com/">example.com</a>.</p>
    </body>
</html>
```

Example of changing the attribute "target" of every link in first paragraph:

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")      # Finds first occurrence of tag p in body
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))    # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:              # Iterates through all found links
...     i.attrib["target"] = "blank"
...
>>> tree.write("output.xhtml")
```

QName Objects

class xml.etree.ElementTree.**QName**(*text_or_uri*, *tag=None*)

QName wrapper. This can be used to wrap a QName attribute value, in order to get proper namespace handling on output. *text_or_uri* is a string containing the QName value, in the form {uri}local, or, if the tag argument is given, the URI part of a QName. If *tag* is given, the first argument is interpreted as a URI, and this argument is interpreted as a local name. *QName* instances are opaque.

TreeBuilder Objects

class xml.etree.ElementTree.**TreeBuilder**(*element_factory=None*)

Generic element structure builder. This builder converts a sequence of start, data, and end method calls to a well-formed element structure. You can use this class to build an element structure using a custom XML parser, or a parser for some other XML-like format. The *element_factory* is called to create new *Element* instances when given.

close()

Flushes the builder buffers, and returns the toplevel document element. Returns an *Element* instance.

data(*data*)

Adds text to the current element. *data* is a string. This should be either a bytestring, or a Unicode string.

end(*tag*)

Closes the current element. *tag* is the element name. Returns the closed element.

start(*tag*, *attrs*)

Opens a new element. *tag* is the element name. *attrs* is a dictionary containing element attributes. Returns the opened element.

In addition, a custom *TreeBuilder* object can provide the following method:

doctype(*name*, *pubid*, *system*)
Handles a doctype declaration. *name* is the doctype name. *pubid* is the public identifier. *system* is the system identifier. This method does not exist on the default *TreeBuilder* class.

2.7 新版功能.

XMLParser 对象

class xml.etree.ElementTree.**XMLParser**(*html=0*, *target=None*, *encoding=None*)
Element structure builder for XML source data, based on the expat parser. *html* are predefined HTML entities. This flag is not supported by the current implementation. *target* is the target object. If omitted, the builder uses an instance of the standard TreeBuilder class. *encoding*[1] is optional. If given, the value overrides the encoding specified in the XML file.

close()
Finishes feeding data to the parser. Returns an element structure.

doctype(*name*, *pubid*, *system*)
2.7 版后已移除: Define the *TreeBuilder.doctype()* method on a custom TreeBuilder target.

feed(*data*)
Feeds data to the parser. *data* is encoded data.

XMLParser.feed() calls *target*'s `start()` method for each opening tag, its `end()` method for each closing tag, and data is processed by method `data()`. *XMLParser.close()* calls *target*'s method `close()`. *XMLParser* can be used not only for building a tree structure. This is an example of counting the maximum depth of an XML file:

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                     # The target object of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # Called for each opening tag.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # Called for each closing tag.
...         self.depth -= 1
...     def data(self, data):
...         pass            # We do not need to do anything with data.
...     def close(self):    # Called when all data has been parsed.
...         return self.maxDepth
...
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
...     </c>
...   </b>
... </a>"""
>>> parser.feed(exampleXml)
```

(下页继续)

(续上页)

```
>>> parser.close()
4
```

备注

19.8 xml.dom —The Document Object Model API

2.0 新版功能.

The Document Object Model, or "DOM," is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents. A DOM implementation presents an XML document as a tree structure, or allows client code to build such a structure from scratch. It then gives access to the structure through a set of objects which provided well-known interfaces.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another. If you are looking at a text node, you have no access to a containing element. When you write a SAX application, you need to keep track of your program's position in the document somewhere in your own code. SAX does not do it for you. Also, if you need to look ahead in the XML document, you are just out of luck.

Some applications are simply impossible in an event driven model with no access to a tree. Of course you could build some sort of tree yourself in SAX events, but the DOM allows you to avoid writing that code. The DOM is a standard tree representation for XML data.

The Document Object Model is being defined by the W3C in stages, or "levels" in their terminology. The Python mapping of the API is substantially based on the DOM Level 2 recommendation.

DOM applications typically start by parsing some XML into a DOM. How this is accomplished is not covered at all by DOM Level 1, and Level 2 provides only limited improvements: There is a `DOMImplementation` object class which provides access to `Document` creation methods, but no way to access an XML reader/parser/Document builder in an implementation-independent way. There is also no well-defined way to access these methods without an existing `Document` object. In Python, each DOM implementation will provide a function *`getDOMImplementation()`*. DOM Level 3 adds a Load/Store specification, which defines an interface to the reader, but this is not yet available in the Python standard library.

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification; this portion of the reference manual describes the interpretation of the specification in Python.

The specification provided by the W3C defines the DOM API for Java, ECMAScript, and OMG IDL. The Python mapping defined here is based in large part on the IDL version of the specification, but strict compliance is not required (though implementations are free to support the strict mapping from IDL). See section 一致性 for a detailed discussion of mapping requirements.

参见:

Document Object Model (DOM) Level 2 Specification The W3C recommendation upon which the Python DOM API is based.

Document Object Model (DOM) Level 1 Specification The W3C recommendation for the DOM supported by *`xml.dom.minidom`*.

Python Language Mapping Specification This specifies the mapping from OMG IDL to Python.

19.8.1 模块内容

The *xml.dom* contains the following functions:

xml.dom.**registerDOMImplementation**(*name*, *factory*)
Register the *factory* function with the name *name*. The factory function should return an object which implements the DOMImplementation interface. The factory function can return the same object every time, or a new one for each call, as appropriate for the specific implementation (e.g. if that implementation supports some customization).

xml.dom.**getDOMImplementation**([*name*[, *features*]])
Return a suitable DOM implementation. The *name* is either well-known, the module name of a DOM implementation, or None. If it is not None, imports the corresponding module and returns a DOMImplementation object if the import succeeds. If no name is given, and if the environment variable PYTHON_DOM is set, this variable is used to find the implementation.

If name is not given, this examines the available implementations to find one with the required feature set. If no implementation can be found, raise an *ImportError*. The features list must be a sequence of (feature, version) pairs which are passed to the hasFeature() method on available DOMImplementation objects.

Some convenience constants are also provided:

xml.dom.**EMPTY_NAMESPACE**
The value used to indicate that no namespace is associated with a node in the DOM. This is typically found as the namespaceURI of a node, or used as the *namespaceURI* parameter to a namespaces-specific method.

2.2 新版功能.

xml.dom.**XML_NAMESPACE**
The namespace URI associated with the reserved prefix xml, as defined by Namespaces in XML (section 4).

2.2 新版功能.

xml.dom.**XMLNS_NAMESPACE**
The namespace URI for namespace declarations, as defined by Document Object Model (DOM) Level 2 Core Specification (section 1.1.8).

2.2 新版功能.

xml.dom.**XHTML_NAMESPACE**
The URI of the XHTML namespace as defined by XHTML 1.0: The Extensible HyperText Markup Language (section 3.1.1).

2.2 新版功能.

In addition, *xml.dom* contains a base Node class and the DOM exception classes. The Node class provided by this module does not implement any of the methods or attributes defined by the DOM specification; concrete DOM implementations must provide those. The Node class provided as part of this module does provide the constants used for the nodeType attribute on concrete Node objects; they are located within the class rather than at the module level to conform with the DOM specifications.

19.8.2 Objects in the DOM

DOM 的权威文档是来自 W3C 的 DOM 规范。

请注意，DOM 属性也可以作为节点而不是简单的字符串进行操作。然而，必须这样做的情况相当少见，所以这种用法还没有记录下来。

接口	部件	目的
DOMImplementation	*DOMImplementation Objects*	底层实现的接口。
Node	节点对象	文档中大多数对象的基本接口。
NodeList	节点列表对象	节点序列的接口。
DocumentType	文档类型对象	有关处理文档所需声明的信息。
Document	文档对象	表示整个文档的对象。
Element	元素对象	文档层次结构中的元素节点。
Attr	*Attr* 对象	元素节点上的属性值节点。
Comment	注释对象	源文档中注释的表示形式。
Text	*Text* 和 *CDATASection* 对象	包含文档中文本内容的节点。
ProcessingInstruction	*ProcessingInstruction* 对象	Processing instruction representation.

另一节描述了在 Python 中使用 DOM 定义的异常。

DOMImplementation Objects

The DOMImplementation interface provides a way for applications to determine the availability of particular features in the DOM they are using. DOM Level 2 added the ability to create new Document and DocumentType objects using the DOMImplementation as well.

DOMImplementation.**hasFeature**(*feature*, *version*)
Return true if the feature identified by the pair of strings *feature* and *version* is implemented.

DOMImplementation.**createDocument**(*namespaceUri*, *qualifiedName*, *doctype*)
Return a new Document object (the root of the DOM), with a child Element object having the given *namespaceUri* and *qualifiedName*. The *doctype* must be a DocumentType object created by *createDocumentType()*, or None. In the Python DOM API, the first two arguments can also be None in order to indicate that no Element child is to be created.

DOMImplementation.**createDocumentType**(*qualifiedName*, *publicId*, *systemId*)
Return a new DocumentType object that encapsulates the given *qualifiedName*, *publicId*, and *systemId* strings, representing the information contained in an XML document type declaration.

节点对象

All of the components of an XML document are subclasses of Node.

Node.**nodeType**
An integer representing the node type. Symbolic constants for the types are on the Node object: ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE, CDATA_SECTION_NODE, ENTITY_NODE, PROCESSING_INSTRUCTION_NODE, COMMENT_NODE, DOCUMENT_NODE, DOCUMENT_TYPE_NODE, NOTATION_NODE. This is a read-only attribute.

Node.**parentNode**
The parent of the current node, or None for the document node. The value is always a Node object or None. For Element nodes, this will be the parent element, except for the root element, in which case it will be the Document object. For Attr nodes, this is always None. This is a read-only attribute.

Node.**attributes**
A NamedNodeMap of attribute objects. Only elements have actual values for this; others provide None for this attribute. This is a read-only attribute.

Node.**previousSibling**
The node that immediately precedes this one with the same parent. For instance the element with an end-tag that comes just before the *self* element's start-tag. Of course, XML documents are made up of more than just elements so the previous sibling could be text, a comment, or something else. If this node is the first child of the parent, this attribute will be None. This is a read-only attribute.

Node.**nextSibling**
The node that immediately follows this one with the same parent. See also *previousSibling*. If this is the last child of the parent, this attribute will be None. This is a read-only attribute.

Node.**childNodes**
A list of nodes contained within this node. This is a read-only attribute.

Node.**firstChild**
The first child of the node, if there are any, or None. This is a read-only attribute.

Node.**lastChild**
The last child of the node, if there are any, or None. This is a read-only attribute.

Node.**localName**
The part of the tagName following the colon if there is one, else the entire tagName. The value is a string.

Node.**prefix**
The part of the tagName preceding the colon if there is one, else the empty string. The value is a string, or None.

Node.**namespaceURI**
The namespace associated with the element name. This will be a string or None. This is a read-only attribute.

Node.**nodeName**
This has a different meaning for each node type; see the DOM specification for details. You can always get the information you would get here from another property such as the tagName property for elements or the name property for attributes. For all node types, the value of this attribute will be either a string or None. This is a read-only attribute.

Node.**nodeValue**
This has a different meaning for each node type; see the DOM specification for details. The situation is similar to that with *nodeName*. The value is a string or None.

Node.**hasAttributes**()
Returns true if the node has any attributes.

Node.**hasChildNodes**()
Returns true if the node has any child nodes.

Node.**isSameNode**(*other*)
Returns true if *other* refers to the same node as this node. This is especially useful for DOM implementations which use any sort of proxy architecture (because more than one object can refer to the same node).

注解: This is based on a proposed DOM Level 3 API which is still in the "working draft" stage, but this particular interface appears uncontroversial. Changes from the W3C will not necessarily affect this method in the Python DOM interface (though any new W3C API for this would also be supported).

Node.**appendChild**(*newChild*)
Add a new child node to this node at the end of the list of children, returning *newChild*. If the node was already in the tree, it is removed first.

Node.**insertBefore**(*newChild*, *refChild*)
Insert a new child node before an existing child. It must be the case that *refChild* is a child of this node; if not, *ValueError* is raised. *newChild* is returned. If *refChild* is None, it inserts *newChild* at the end of the children's list.

Node.**removeChild**(*oldChild*)
Remove a child node. *oldChild* must be a child of this node; if not, *ValueError* is raised. *oldChild* is returned on success. If *oldChild* will not be used further, its unlink() method should be called.

Node.**replaceChild**(*newChild*, *oldChild*)
Replace an existing node with a new node. It must be the case that *oldChild* is a child of this node; if not, *ValueError* is raised.

Node.**normalize**()
Join adjacent text nodes so that all stretches of text are stored as single Text instances. This simplifies processing text from a DOM tree for many applications.

2.1 新版功能.

Node.**cloneNode**(*deep*)
Clone this node. Setting *deep* means to clone all child nodes as well. This returns the clone.

节点列表对象

A NodeList represents a sequence of nodes. These objects are used in two ways in the DOM Core recommendation: an Element object provides one as its list of child nodes, and the getElementsByTagName() and getElementsByTagNameNS() methods of Node return objects with this interface to represent query results.

The DOM Level 2 recommendation defines one method and one attribute for these objects:

NodeList.**item**(*i*)
Return the *i*' th item from the sequence, if there is one, or None. The index *i* is not allowed to be less than zero or greater than or equal to the length of the sequence.

NodeList.**length**
The number of nodes in the sequence.

In addition, the Python DOM interface requires that some additional support is provided to allow NodeList objects to be used as Python sequences. All NodeList implementations must include support for __len__() and __getitem__(); this allows iteration over the NodeList in for statements and proper support for the *len()* built-in function.

If a DOM implementation supports modification of the document, the NodeList implementation must also support the __setitem__() and __delitem__() methods.

文档类型对象

Information about the notations and entities declared by a document (including the external subset if the parser uses it and can provide the information) is available from a DocumentType object. The DocumentType for a document is available from the Document object's doctype attribute; if there is no DOCTYPE declaration for the document, the document's doctype attribute will be set to None instead of an instance of this interface.

DocumentType is a specialization of Node, and adds the following attributes:

DocumentType.**publicId**
The public identifier for the external subset of the document type definition. This will be a string or None.

DocumentType.**systemId**
The system identifier for the external subset of the document type definition. This will be a URI as a string, or None.

DocumentType.**internalSubset**
A string giving the complete internal subset from the document. This does not include the brackets which enclose the subset. If the document has no internal subset, this should be None.

DocumentType.**name**
The name of the root element as given in the DOCTYPE declaration, if present.

DocumentType.**entities**
This is a NamedNodeMap giving the definitions of external entities. For entity names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no entities are defined.

DocumentType.**notations**
This is a NamedNodeMap giving the definitions of notations. For notation names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be None if the information is not provided by the parser, or if no notations are defined.

文档对象

A Document represents an entire XML document, including its constituent elements, attributes, processing instructions, comments etc. Remember that it inherits properties from Node.

Document.**documentElement**
The one and only root element of the document.

Document.**createElement**(*tagName*)
Create and return a new element node. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as insertBefore() or appendChild().

Document.**createElementNS**(*namespaceURI*, *tagName*)
Create and return a new element with a namespace. The *tagName* may have a prefix. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as insertBefore() or appendChild().

Document.**createTextNode**(*data*)
Create and return a text node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

Document.**createComment**(*data*)
Create and return a comment node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

Document.**createProcessingInstruction**(*target*, *data*)
Create and return a processing instruction node containing the *target* and *data* passed as parameters. As with the other creation methods, this one does not insert the node into the tree.

Document.**createAttribute**(*name*)
Create and return an attribute node. This method does not associate the attribute node with any particular element. You must use setAttributeNode() on the appropriate Element object to use the newly created attribute instance.

Document.**createAttributeNS**(*namespaceURI*, *qualifiedName*)
Create and return an attribute node with a namespace. The *tagName* may have a prefix. This method does not associate the attribute node with any particular element. You must use setAttributeNode() on the appropriate Element object to use the newly created attribute instance.

Document.**getElementsByTagName**(*tagName*)
Search for all descendants (direct children, children' s children, etc.) with a particular element type name.

Document.**getElementsByTagNameNS**(*namespaceURI*, *localName*)
Search for all descendants (direct children, children' s children, etc.) with a particular namespace URI and localname. The localname is the part of the namespace after the prefix.

元素对象

Element is a subclass of Node, so inherits all the attributes of that class.

Element.**tagName**
The element type name. In a namespace-using document it may have colons in it. The value is a string.

Element.**getElementsByTagName**(*tagName*)
Same as equivalent method in the Document class.

Element.**getElementsByTagNameNS**(*namespaceURI*, *localName*)
Same as equivalent method in the Document class.

Element.**hasAttribute**(*name*)
Returns true if the element has an attribute named by *name*.

Element.**hasAttributeNS**(*namespaceURI*, *localName*)
Returns true if the element has an attribute named by *namespaceURI* and *localName*.

Element.**getAttribute**(*name*)
Return the value of the attribute named by *name* as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

Element.**getAttributeNode**(*attrname*)
Return the Attr node for the attribute named by *attrname*.

Element.**getAttributeNS**(*namespaceURI*, *localName*)
Return the value of the attribute named by *namespaceURI* and *localName* as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

Element.**getAttributeNodeNS**(*namespaceURI*, *localName*)
Return an attribute value as a node, given a *namespaceURI* and *localName*.

Element.**removeAttribute**(*name*)
Remove an attribute by name. If there is no matching attribute, a *NotFoundErr* is raised.

Element.**removeAttributeNode**(*oldAttr*)
Remove and return *oldAttr* from the attribute list, if present. If *oldAttr* is not present, *NotFoundErr* is raised.

Element.**removeAttributeNS**(*namespaceURI*, *localName*)
Remove an attribute by name. Note that it uses a localName, not a qname. No exception is raised if there is no matching attribute.

Element.**setAttribute**(*name*, *value*)
Set an attribute value from a string.

Element.**setAttributeNode**(*newAttr*)
Add a new attribute node to the element, replacing an existing attribute if necessary if the name attribute matches. If a replacement occurs, the old attribute node will be returned. If *newAttr* is already in use, *InuseAttributeErr* will be raised.

Element.**setAttributeNodeNS**(*newAttr*)
Add a new attribute node to the element, replacing an existing attribute if necessary if the namespaceURI and

localName attributes match. If a replacement occurs, the old attribute node will be returned. If *newAttr* is already in use, *InuseAttributeErr* will be raised.

Element.**setAttributeNS**(*namespaceURI*, *qname*, *value*)
Set an attribute value from a string, given a *namespaceURI* and a *qname*. Note that a qname is the whole attribute name. This is different than above.

Attr 对象

Attr inherits from Node, so inherits all its attributes.

Attr.**name**
The attribute name. In a namespace-using document it may include a colon.

Attr.**localName**
The part of the name following the colon if there is one, else the entire name. This is a read-only attribute.

Attr.**prefix**
The part of the name preceding the colon if there is one, else the empty string.

Attr.**value**
The text value of the attribute. This is a synonym for the nodeValue attribute.

NamedNodeMap 对象

NamedNodeMap does *not* inherit from Node.

NamedNodeMap.**length**
The length of the attribute list.

NamedNodeMap.**item**(*index*)
Return an attribute with a particular index. The order you get the attributes in is arbitrary but will be consistent for the life of a DOM. Each item is an attribute node. Get its value with the value attribute.

There are also experimental methods that give this class more mapping behavior. You can use them or you can use the standardized getAttribute*() family of methods on the Element objects.

注释对象

Comment represents a comment in the XML document. It is a subclass of Node, but cannot have child nodes.

Comment.**data**
The content of the comment as a string. The attribute contains all characters between the leading <!-- and trailing -->, but does not include them.

Text 和 CDATASection 对象

The Text interface represents text in the XML document. If the parser and DOM implementation support the DOM's XML extension, portions of the text enclosed in CDATA marked sections are stored in CDATASection objects. These two interfaces are identical, but provide different values for the nodeType attribute.

These interfaces extend the Node interface. They cannot have child nodes.

Text.**data**
The content of the text node as a string.

注解: The use of a `CDATASection` node does not indicate that the node represents a complete CDATA marked section, only that the content of the node was part of a CDATA section. A single CDATA section may be represented by more than one node in the document tree. There is no way to determine whether two adjacent `CDATASection` nodes represent different CDATA marked sections.

ProcessingInstruction 对象

Represents a processing instruction in the XML document; this inherits from the `Node` interface and cannot have child nodes.

`ProcessingInstruction.`**`target`**
The content of the processing instruction up to the first whitespace character. This is a read-only attribute.

`ProcessingInstruction.`**`data`**
The content of the processing instruction following the first whitespace character.

异常

2.1 新版功能.

The DOM Level 2 recommendation defines a single exception, *`DOMException`*, and a number of constants that allow applications to determine what sort of error occurred. *`DOMException`* instances carry a *`code`* attribute that provides the appropriate value for the specific exception.

The Python DOM interface provides the constants, but also expands the set of exceptions so that a specific exception exists for each of the exception codes defined by the DOM. The implementations must raise the appropriate specific exception, each of which carries the appropriate value for the *`code`* attribute.

exception `xml.dom.`**DOMException**
Base exception class used for all specific DOM exceptions. This exception class cannot be directly instantiated.

exception `xml.dom.`**DomstringSizeErr**
Raised when a specified range of text does not fit into a string. This is not known to be used in the Python DOM implementations, but may be received from DOM implementations not written in Python.

exception `xml.dom.`**HierarchyRequestErr**
Raised when an attempt is made to insert a node where the node type is not allowed.

exception `xml.dom.`**IndexSizeErr**
Raised when an index or size parameter to a method is negative or exceeds the allowed values.

exception `xml.dom.`**InuseAttributeErr**
Raised when an attempt is made to insert an `Attr` node that is already present elsewhere in the document.

exception `xml.dom.`**InvalidAccessErr**
Raised if a parameter or an operation is not supported on the underlying object.

exception `xml.dom.`**InvalidCharacterErr**
This exception is raised when a string parameter contains a character that is not permitted in the context it's being used in by the XML 1.0 recommendation. For example, attempting to create an `Element` node with a space in the element type name will cause this error to be raised.

exception `xml.dom.`**InvalidModificationErr**
Raised when an attempt is made to modify the type of a node.

exception `xml.dom.`**InvalidStateErr**
Raised when an attempt is made to use an object that is not defined or is no longer usable.

exception xml.dom.**NamespaceErr**
If an attempt is made to change any object in a way that is not permitted with regard to the Namespaces in XML recommendation, this exception is raised.

exception xml.dom.**NotFoundErr**
Exception when a node does not exist in the referenced context. For example, `NamedNodeMap.removeNamedItem()` will raise this if the node passed in does not exist in the map.

exception xml.dom.**NotSupportedErr**
Raised when the implementation does not support the requested type of object or operation.

exception xml.dom.**NoDataAllowedErr**
This is raised if data is specified for a node which does not support data.

exception xml.dom.**NoModificationAllowedErr**
Raised on attempts to modify an object where modifications are not allowed (such as for read-only nodes).

exception xml.dom.**SyntaxErr**
Raised when an invalid or illegal string is specified.

exception xml.dom.**WrongDocumentErr**
Raised when a node is inserted in a different document than it currently belongs to, and the implementation does not support migrating the node from one document to the other.

The exception codes defined in the DOM recommendation map to the exceptions described above according to this table:

常数	异常
DOMSTRING_SIZE_ERR	*DomstringSizeErr*
HIERARCHY_REQUEST_ERR	*HierarchyRequestErr*
INDEX_SIZE_ERR	*IndexSizeErr*
INUSE_ATTRIBUTE_ERR	*InuseAttributeErr*
INVALID_ACCESS_ERR	*InvalidAccessErr*
INVALID_CHARACTER_ERR	*InvalidCharacterErr*
INVALID_MODIFICATION_ERR	*InvalidModificationErr*
INVALID_STATE_ERR	*InvalidStateErr*
NAMESPACE_ERR	*NamespaceErr*
NOT_FOUND_ERR	*NotFoundErr*
NOT_SUPPORTED_ERR	*NotSupportedErr*
NO_DATA_ALLOWED_ERR	*NoDataAllowedErr*
NO_MODIFICATION_ALLOWED_ERR	*NoModificationAllowedErr*
SYNTAX_ERR	*SyntaxErr*
WRONG_DOCUMENT_ERR	*WrongDocumentErr*

19.8.3 一致性

This section describes the conformance requirements and relationships between the Python DOM API, the W3C DOM recommendations, and the OMG IDL mapping for Python.

类型映射

The primitive IDL types used in the DOM specification are mapped to Python types according to the following table.

IDL 类型	Python 类型
`boolean`	`IntegerType` (with a value of `0` or `1`)
`int`	`IntegerType`
`long int`	`IntegerType`
`unsigned int`	`IntegerType`

Additionally, the `DOMString` defined in the recommendation is mapped to a Python string or Unicode string. Applications should be able to handle Unicode whenever a string is returned from the DOM.

The IDL `null` value is mapped to `None`, which may be accepted or provided by the implementation whenever `null` is allowed by the API.

Accessor Methods

The mapping from OMG IDL to Python defines accessor functions for IDL `attribute` declarations in much the way the Java mapping does. Mapping the IDL declarations

```
readonly attribute string someValue;
         attribute string anotherValue;
```

yields three accessor functions: a "get" method for `someValue` (`_get_someValue()`), and "get" and "set" methods for `anotherValue` (`_get_anotherValue()` and `_set_anotherValue()`). The mapping, in particular, does not require that the IDL attributes are accessible as normal Python attributes: `object.someValue` is *not* required to work, and may raise an *AttributeError*.

The Python DOM API, however, *does* require that normal attribute access work. This means that the typical surrogates generated by Python IDL compilers are not likely to work, and wrapper objects may be needed on the client if the DOM objects are accessed via CORBA. While this does require some additional consideration for CORBA DOM clients, the implementers with experience using DOM over CORBA from Python do not consider this a problem. Attributes that are declared `readonly` may not restrict write access in all DOM implementations.

In the Python DOM API, accessor functions are not required. If provided, they should take the form defined by the Python IDL mapping, but these methods are considered unnecessary since the attributes are accessible directly from Python. "Set" accessors should never be provided for `readonly` attributes.

The IDL definitions do not fully embody the requirements of the W3C DOM API, such as the notion of certain objects, such as the return value of `getElementsByTagName()`, being "live". The Python DOM API does not require implementations to enforce such requirements.

19.9 `xml.dom.minidom` —Minimal DOM implementation

2.0 新版功能.

Source code: Lib/xml/dom/minidom.py

---

xml.dom.minidom is a minimal implementation of the Document Object Model interface, with an API similar to that in other languages. It is intended to be simpler than the full DOM and also significantly smaller. Users who are

not already proficient with the DOM should consider using the *xml.etree.ElementTree* module for their XML processing instead.

> **警告:** The *xml.dom.minidom* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML 漏洞*.

DOM applications typically start by parsing some XML into a DOM. With *xml.dom.minidom*, this is done through the parse functions:

```
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml')  # parse an XML file by name

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource)  # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

The *parse()* function can take either a filename or an open file object.

xml.dom.minidom.**parse**(*filename_or_file*[, *parser*[, *bufsize*]])
Return a Document from the given input. *filename_or_file* may be either a file name, or a file-like object. *parser*, if given, must be a SAX2 parser object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the *parseString()* function instead:

xml.dom.minidom.**parseString**(*string*[, *parser*])
Return a Document that represents the *string*. This method creates a *StringIO* object for the string and passes that on to *parse()*.

Both functions return a Document object representing the content of the document.

What the *parse()* and *parseString()* functions do is connect an XML parser with a "DOM builder" that can accept parse events from any SAX parser and convert them into a DOM tree. The name of the functions are perhaps misleading, but are easy to grasp when learning the interfaces. The parsing of the document will be completed before these functions return; it's simply that these functions do not provide a parser implementation themselves.

You can also create a Document by calling a method on a "DOM Implementation" object. You can get this object either by calling the getDOMImplementation() function in the *xml.dom* package or the *xml.dom.minidom* module. Using the implementation from the *xml.dom.minidom* module will always return a Document instance from the minidom implementation, while the version from *xml.dom* may provide an alternate implementation (this is likely if you have the PyXML package installed). Once you have a Document, you can add child nodes to it to populate the DOM:

```
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification. The main property of the document object is the

documentElement property. It gives you the main element in the XML document: the one that holds all others. Here is an example program:

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

When you are finished with a DOM tree, you may optionally call the unlink() method to encourage early cleanup of the now-unneeded objects. unlink() is an *xml.dom.minidom*-specific extension to the DOM API that renders the node and its descendants are essentially useless. Otherwise, Python's garbage collector will eventually take care of the objects in the tree.

参见:

Document Object Model (DOM) Level 1 Specification The W3C recommendation for the DOM supported by *xml.dom.minidom*.

19.9.1 DOM Objects

The definition of the DOM API for Python is given as part of the *xml.dom* module documentation. This section lists the differences between the API and *xml.dom.minidom*.

Node.**unlink**()
Break internal references within the DOM so that it will be garbage collected on versions of Python without cyclic GC. Even when cyclic GC is available, using this can make large amounts of memory available sooner, so calling this on DOM objects as soon as they are no longer needed is good practice. This only needs to be called on the Document object, but may be called on child nodes to discard children of that node.

Node.**writexml**(*writer*, *indent=""*, *addindent=""*, *newl=""*)
Write XML to the writer object. The writer should have a write() method which matches that of the file object interface. The *indent* parameter is the indentation of the current node. The *addindent* parameter is the incremental indentation to use for subnodes of the current one. The *newl* parameter specifies the string to use to terminate newlines.

For the Document node, an additional keyword argument *encoding* can be used to specify the encoding field of the XML header.

在 2.1 版更改: The optional keyword parameters *indent*, *addindent*, and *newl* were added to support pretty output.

在 2.3 版更改: For the Document node, an additional keyword argument *encoding* can be used to specify the encoding field of the XML header.

Node.**toxml**([*encoding*])
Return the XML that the DOM represents as a string.

With no argument, the XML header does not specify an encoding, and the result is Unicode string if the default encoding cannot represent all characters in the document. Encoding this string in an encoding other than UTF-8 is likely incorrect, since UTF-8 is the default encoding of XML.

With an explicit *encoding*[1] argument, the result is a byte string in the specified encoding. It is recommended that this argument is always specified. To avoid *UnicodeError* exceptions in case of unrepresentable text data, the encoding argument should be specified as "utf-8".

在 2.3 版更改: the *encoding* argument was introduced; see *writexml()*.

[1] The encoding string included in XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

Node.**toprettyxml**(*indent="\t", newl="\n", encoding=None*)
Return a pretty-printed version of the document. *indent* specifies the indentation string and defaults to a tabulator; *newl* specifies the string emitted at the end of each line and defaults to `\n`.

2.1 新版功能.

在 2.3 版更改: the encoding argument was introduced; see *writexml()*.

The following standard DOM methods have special considerations with *xml.dom.minidom*:

Node.**cloneNode**(*deep*)
Although this method was present in the version of *xml.dom.minidom* packaged with Python 2.0, it was seriously broken. This has been corrected for subsequent releases.

19.9.2 DOM Example

This example program is a fairly realistic example of a simple program. In this particular case, we do not take much advantage of the flexibility of the DOM.

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print "<html>"
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print "</html>"

def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)
```

(下页继续)

(续上页)

```
def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print "<title>%s</title>" % getText(title.childNodes)

def handleSlideTitle(title):
    print "<h2>%s</h2>" % getText(title.childNodes)

def handlePoints(points):
    print "<ul>"
    for point in points:
        handlePoint(point)
    print "</ul>"

def handlePoint(point):
    print "<li>%s</li>" % getText(point.childNodes)

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print "<p>%s</p>" % getText(title.childNodes)

handleSlideshow(dom)
```

19.9.3 minidom and the DOM standard

The *xml.dom.minidom* module is essentially a DOM 1.0-compatible DOM with some DOM 2 features (primarily namespace features).

Usage of the DOM interface in Python is straight-forward. The following mapping rules apply:

- Interfaces are accessed through instance objects. Applications should not instantiate the classes themselves; they should use the creator functions available on the `Document` object. Derived interfaces support all operations (and attributes) from the base interfaces, plus any new operations.
- Operations are used as methods. Since the DOM uses only `in` parameters, the arguments are passed in normal order (from left to right). There are no optional arguments. `void` operations return `None`.
- IDL attributes map to instance attributes. For compatibility with the OMG IDL language mapping for Python, an attribute `foo` can also be accessed through accessor methods `_get_foo()` and `_set_foo()`. `readonly` attributes must not be changed; this is not enforced at runtime.
- The types `short int`, `unsigned int`, `unsigned long long`, and `boolean` all map to Python integer objects.
- The type `DOMString` maps to Python strings. *xml.dom.minidom* supports either byte or Unicode strings, but will normally produce Unicode strings. Values of type `DOMString` may also be `None` where allowed to have the IDL `null` value by the DOM specification from the W3C.
- `const` declarations map to variables in their respective scope (e.g. `xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE`); they must not be changed.
- `DOMException` is currently not supported in *xml.dom.minidom*. Instead, *xml.dom.minidom* uses standard Python exceptions such as *TypeError* and *AttributeError*.

- NodeList objects are implemented using Python's built-in list type. Starting with Python 2.2, these objects provide the interface defined in the DOM specification, but with earlier versions of Python they do not support the official API. They are, however, much more "Pythonic" than the interface defined in the W3C recommendations.

The following interfaces have no implementation in *xml.dom.minidom*:

- DOMTimeStamp
- DocumentType (added in Python 2.1)
- DOMImplementation (added in Python 2.1)
- CharacterData
- CDATASection
- Notation
- Entity
- EntityReference
- DocumentFragment

Most of these reflect information in the XML document that is not of general utility to most DOM users.

备注

19.10 xml.dom.pulldom —Support for building partial DOM trees

2.0 新版功能.

Source code: Lib/xml/dom/pulldom.py

---

xml.dom.pulldom allows building only selected portions of a Document Object Model representation of a document from SAX events.

> **警告:** The *xml.dom.pulldom* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML 漏洞*.

class xml.dom.pulldom.**PullDOM**([*documentFactory*])
xml.sax.handler.ContentHandler implementation that ···

class xml.dom.pulldom.**DOMEventStream**(*stream*, *parser*, *bufsize*)
···

class xml.dom.pulldom.**SAX2DOM**([*documentFactory*])
xml.sax.handler.ContentHandler implementation that ···

xml.dom.pulldom.**parse**(*stream_or_string*[, *parser*[, *bufsize*]])
···

xml.dom.pulldom.**parseString**(*string*[, *parser*])
···

xml.dom.pulldom.**default_bufsize**
Default value for the *bufsize* parameter to *parse()*.

在 2.1 版更改: The value of this variable can be changed before calling *parse()* and the new value will take effect.

19.10.1 DOMEventStream Objects

DOMEventStream.**getEvent**()
…

DOMEventStream.**expandNode**(*node*)
…

DOMEventStream.**reset**()
…

19.11 xml.sax —Support for SAX2 parsers

2.0 新版功能.

The *xml.sax* package provides a number of modules which implement the Simple API for XML (SAX) interface for Python. The package itself provides the SAX exceptions and the convenience functions which will be most used by users of the SAX API.

警告: The *xml.sax* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML 漏洞*.

The convenience functions are:

xml.sax.**make_parser**([*parser_list*])
Create and return a SAX *XMLReader* object. The first parser found will be used. If *parser_list* is provided, it must be a list of strings which name modules that have a function named create_parser(). Modules listed in *parser_list* will be used before modules in the default list of parsers.

xml.sax.**parse**(*filename_or_stream*, *handler*[, *error_handler*])
Create a SAX parser and use it to parse a document. The document, passed in as *filename_or_stream*, can be a filename or a file object. The *handler* parameter needs to be a SAX *ContentHandler* instance. If *error_handler* is given, it must be a SAX *ErrorHandler* instance; if omitted, *SAXParseException* will be raised on all errors. There is no return value; all work must be done by the *handler* passed in.

xml.sax.**parseString**(*string*, *handler*[, *error_handler*])
Similar to *parse()*, but parses from a buffer *string* received as a parameter.

A typical SAX application uses three kinds of objects: readers, handlers and input sources. "Reader" in this context is another term for parser, i.e. some piece of code that reads the bytes or characters from the input source, and produces a sequence of events. The events then get distributed to the handler objects, i.e. the reader invokes a method on the handler. A SAX application must therefore obtain a reader object, create or open the input sources, create the handlers, and connect these objects all together. As the final step of preparation, the reader is called to parse the input. During parsing, methods on the handler objects are called based on structural and syntactic events from the input data.

For these objects, only the interfaces are relevant; they are normally not instantiated by the application itself. Since Python does not have an explicit notion of interface, they are formally introduced as classes, but applications may use implementations which do not inherit from the provided classes. The *InputSource*, *Locator*, Attributes, AttributesNS, and *XMLReader* interfaces are defined in the module *xml.sax.xmlreader*. The handler interfaces are defined in *xml.sax.handler*. For convenience, *InputSource* (which is often instantiated directly) and the handler classes are also available from *xml.sax*. These interfaces are described below.

In addition to these classes, *xml.sax* provides the following exception classes.

exception xml.sax.**SAXException**(*msg*[, *exception*])
Encapsulate an XML error or warning. This class can contain basic error or warning information from either the XML parser or the application: it can be subclassed to provide additional functionality or to add localization. Note that although the handlers defined in the *ErrorHandler* interface receive instances of this exception, it is not required to actually raise the exception —it is also useful as a container for information.

When instantiated, *msg* should be a human-readable description of the error. The optional *exception* parameter, if given, should be None or an exception that was caught by the parsing code and is being passed along as information.

This is the base class for the other SAX exception classes.

exception xml.sax.**SAXParseException**(*msg*, *exception*, *locator*)
Subclass of *SAXException* raised on parse errors. Instances of this class are passed to the methods of the SAX *ErrorHandler* interface to provide information about the parse error. This class supports the SAX *Locator* interface as well as the *SAXException* interface.

exception xml.sax.**SAXNotRecognizedException**(*msg*[, *exception*])
Subclass of *SAXException* raised when a SAX *XMLReader* is confronted with an unrecognized feature or property. SAX applications and extensions may use this class for similar purposes.

exception xml.sax.**SAXNotSupportedException**(*msg*[, *exception*])
Subclass of *SAXException* raised when a SAX *XMLReader* is asked to enable a feature that is not supported, or to set a property to a value that the implementation does not support. SAX applications and extensions may use this class for similar purposes.

参见:

SAX: The Simple API for XML This site is the focal point for the definition of the SAX API. It provides a Java implementation and online documentation. Links to implementations and historical information are also available.

Module ***xml.sax.handler*** Definitions of the interfaces for application-provided objects.

Module ***xml.sax.saxutils*** Convenience functions for use in SAX applications.

Module ***xml.sax.xmlreader*** Definitions of the interfaces for parser-provided objects.

19.11.1 SAXException Objects

The *SAXException* exception class supports the following methods:

SAXException.**getMessage**()
Return a human-readable message describing the error condition.

SAXException.**getException**()
Return an encapsulated exception object, or None.

19.12 xml.sax.handler —Base classes for SAX handlers

2.0 新版功能.

The SAX API defines four kinds of handlers: content handlers, DTD handlers, error handlers, and entity resolvers. Applications normally only need to implement those interfaces whose events they are interested in; they can implement the interfaces in a single object or in multiple objects. Handler implementations should inherit from the base classes provided in the module *xml.sax.handler*, so that all methods get default implementations.

class xml.sax.handler.**ContentHandler**
This is the main callback interface in SAX, and the one most important to applications. The order of events in this interface mirrors the order of the information in the document.

class xml.sax.handler.**DTDHandler**
Handle DTD events.

This interface specifies only those DTD events required for basic parsing (unparsed entities and attributes).

class xml.sax.handler.**EntityResolver**
Basic interface for resolving entities. If you create an object implementing this interface, then register the object with your Parser, the parser will call the method in your object to resolve all external entities.

class xml.sax.handler.**ErrorHandler**
Interface used by the parser to present error and warning messages to the application. The methods of this object control whether errors are immediately converted to exceptions or are handled in some other way.

In addition to these classes, *xml.sax.handler* provides symbolic constants for the feature and property names.

xml.sax.handler.**feature_namespaces**

value: "http://xml.org/sax/features/namespaces"
true: Perform Namespace processing.
false: Optionally do not perform Namespace processing (implies namespace-prefixes; default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_namespace_prefixes**

value: "http://xml.org/sax/features/namespace-prefixes"
true: Report the original prefixed names and attributes used for Namespace declarations.
false: Do not report attributes used for Namespace declarations, and optionally do not report original prefixed names (default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_string_interning**

value: "http://xml.org/sax/features/string-interning"
true: All element names, prefixes, attribute names, Namespace URIs, and local names are interned using the built-in intern function.
false: Names are not necessarily interned, although they may be (default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_validation**

value: "http://xml.org/sax/features/validation"
true: Report all validation errors (implies external-general-entities and external-parameter-entities).
false: Do not report validation errors.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_external_ges**

value: "http://xml.org/sax/features/external-general-entities"
true: Include all external general (text) entities.
false: Do not include external general entities.

access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_external_pes**

value: "http://xml.org/sax/features/external-parameter-entities"
true: Include all external parameter entities, including the external DTD subset.
false: Do not include any external parameter entities, even the external DTD subset.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**all_features**
List of all features.

xml.sax.handler.**property_lexical_handler**

value: "http://xml.org/sax/properties/lexical-handler"
data type: xml.sax.sax2lib.LexicalHandler (not supported in Python 2)
description: An optional extension handler for lexical events like comments.
access: read/write

xml.sax.handler.**property_declaration_handler**

value: "http://xml.org/sax/properties/declaration-handler"
data type: xml.sax.sax2lib.DeclHandler (not supported in Python 2)
description: An optional extension handler for DTD-related events other than notations and unparsed entities.
access: read/write

xml.sax.handler.**property_dom_node**

value: "http://xml.org/sax/properties/dom-node"
data type: org.w3c.dom.Node (not supported in Python 2)
description: When parsing, the current DOM node being visited if this is a DOM iterator; when not parsing, the root DOM node for iteration.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**property_xml_string**

value: "http://xml.org/sax/properties/xml-string"
data type: String
description: The literal string of characters that was the source for the current event.
access: read-only

xml.sax.handler.**all_properties**
List of all known property names.

19.12.1 ContentHandler 对象

Users are expected to subclass *ContentHandler* to support their application. The following methods are called by the parser on the appropriate events in the input document:

ContentHandler.**setDocumentLocator**(*locator*)
Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application's business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

ContentHandler.**startDocument**()
Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler (except for *setDocumentLocator()*).

ContentHandler.**endDocument**()
Receive notification of the end of a document.

The SAX parser will invoke this method only once, and it will be the last method invoked during the parse. The parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or reached the end of input.

ContentHandler.**startPrefixMapping**(*prefix*, *uri*)
Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the `feature_namespaces` feature is enabled (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the *startPrefixMapping()* and *endPrefixMapping()* events supply the information to the application to expand prefixes in those contexts itself, if necessary.

Note that *startPrefixMapping()* and *endPrefixMapping()* events are not guaranteed to be properly nested relative to each-other: all *startPrefixMapping()* events will occur before the corresponding *startElement()* event, and all *endPrefixMapping()* events will occur after the corresponding *endElement()* event, but their order is not guaranteed.

ContentHandler.**endPrefixMapping**(*prefix*)
End the scope of a prefix-URI mapping.

See *startPrefixMapping()* for details. This event will always occur after the corresponding *endElement()* event, but the order of *endPrefixMapping()* events is not otherwise guaranteed.

ContentHandler.**startElement**(*name*, *attrs*)
Signals the start of an element in non-namespace mode.

The *name* parameter contains the raw XML 1.0 name of the element type as a string and the *attrs* parameter holds an object of the `Attributes` interface (see *The Attributes Interface*) containing the attributes of the element. The object passed as *attrs* may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the *copy()* method of the *attrs* object.

ContentHandler.**endElement**(*name*)
Signals the end of an element in non-namespace mode.

The *name* parameter contains the name of the element type, just as with the *startElement()* event.

ContentHandler.**startElementNS**(*name*, *qname*, *attrs*)
Signals the start of an element in namespace mode.

The *name* parameter contains the name of the element type as a `(uri, localname)` tuple, the *qname* parameter contains the raw XML 1.0 name used in the source document, and the *attrs* parameter holds an instance of the `AttributesNS` interface (see *The AttributesNS Interface*) containing the attributes of the element. If no namespace is associated with the element, the *uri* component of *name* will be `None`. The object passed as *attrs* may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the *copy()* method of the *attrs* object.

Parsers may set the *qname* parameter to `None`, unless the `feature_namespace_prefixes` feature is activated.

ContentHandler.**endElementNS**(*name*, *qname*)
Signals the end of an element in namespace mode.

The *name* parameter contains the name of the element type, just as with the *startElementNS()* method, likewise the *qname* parameter.

ContentHandler.**characters**(*content*)
Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information.

content may be a Unicode string or a byte string; the `expat` reader module produces always Unicode strings.

注解: The earlier SAX 1 interface provided by the Python XML Special Interest Group used a more Java-like interface for this method. Since most parsers used from Python did not take advantage of the older interface, the simpler signature was chosen to replace it. To convert old code to the new interface, use *content* instead of slicing content with the old *offset* and *length* parameters.

ContentHandler.**ignorableWhitespace**(*whitespace*)
Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

ContentHandler.**processingInstruction**(*target*, *data*)
Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

ContentHandler.**skippedEntity**(*name*)
Receive notification of a skipped entity.

The Parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the `feature_external_ges` and the `feature_external_pes` properties.

19.12.2 DTDHandler 对象

DTDHandler instances provide the following methods:

`DTDHandler.`**`notationDecl`**(*name*, *publicId*, *systemId*)
Handle a notation declaration event.

`DTDHandler.`**`unparsedEntityDecl`**(*name*, *publicId*, *systemId*, *ndata*)
Handle an unparsed entity declaration event.

19.12.3 EntityResolver 对象

`EntityResolver.`**`resolveEntity`**(*publicId*, *systemId*)
Resolve the system identifier of an entity and return either the system identifier to read from as a string, or an InputSource to read from. The default implementation returns *systemId*.

19.12.4 ErrorHandler 对象

Objects with this interface are used to receive error and warning information from the *XMLReader*. If you create an object that implements this interface, then register the object with your *XMLReader*, the parser will call the methods in your object to report all warnings and errors. There are three levels of errors available: warnings, (possibly) recoverable errors, and unrecoverable errors. All methods take a `SAXParseException` as the only parameter. Errors and warnings may be converted to an exception by raising the passed-in exception object.

`ErrorHandler.`**`error`**(*exception*)
Called when the parser encounters a recoverable error. If this method does not raise an exception, parsing may continue, but further document information should not be expected by the application. Allowing the parser to continue may allow additional errors to be discovered in the input document.

`ErrorHandler.`**`fatalError`**(*exception*)
Called when the parser encounters an error it cannot recover from; parsing is expected to terminate when this method returns.

`ErrorHandler.`**`warning`**(*exception*)
Called when the parser presents minor warning information to the application. Parsing is expected to continue when this method returns, and document information will continue to be passed to the application. Raising an exception in this method will cause parsing to end.

19.13 xml.sax.saxutils —SAX 工具集

2.0 新版功能.

xml.sax.saxutils 模块包含一些在创建 SAX 应用程序时十分有用的类和函数，它们可以被直接使用，或者是作为基类使用。

xml.sax.saxutils.**escape**(*data*[, *entities*])
对数据字符串中对 '&', '<' 和 '>' 进行转义。

你可以通过传入一个字典作为可选的 *entities* 形参来对其他字符串数据进行转义。字典的键和值必须为字符串；每个键将被替换为其所对应的值。字符 '&', '<' 和 '>' 总是会被转义，即使提供了 *entities*。

xml.sax.saxutils.**unescape**(*data*[, *entities*])
对字符串数据中的 '&', '<' 和 '>' 进行反转义。

你可以通过传入一个字典作为可选的 *entities* 形参来对其他数据字符串进行转义。字典的键和值必须都为字符串；每个键将被替换为所对应的值。'&amp', '<' 和 '>' 将总是保持不被转义，即使提供了 *entities*。

2.3 新版功能.

xml.sax.saxutils.**quoteattr**(*data*[, *entities*])
类似于 *escape()*，但还会对 *data* 进行处理以将其用作属性值。返回值是 *data* 加上任何额外要求的替换的带引号版本。*quoteattr()* 将基于 *data* 的内容选择一个引号字符，以尽量避免在字符串中编码任何引号字符。如果单双引号字符在 *data* 中都存在，则双引号字符将被编码并且 *data* 将使用双引号来标记。结果字符串可被直接用作属性值:

```
>>> print "<element attr=%s>" % quoteattr("ab ' cd \" ef")
<element attr="ab ' cd &quot; ef">
```

此函数适用于为 HTML 或任何使用引用实体语法的 SGML 生成属性值。

2.2 新版功能.

class xml.sax.saxutils.**XMLGenerator**([*out*[, *encoding*]])
This class implements the *ContentHandler* interface by writing SAX events back into an XML document. In other words, using an *XMLGenerator* as the content handler will reproduce the original document being parsed. *out* should be a file-like object which will default to *sys.stdout*. *encoding* is the encoding of the output stream which defaults to 'iso-8859-1'.

class xml.sax.saxutils.**XMLFilterBase**(*base*)
这个类被设计用来分隔 *XMLReader* 和客户端应用的事件处理程序。在默认情况下，它除了将请求传送给读取器并将事件传送给处理程序之外什么都不做，但其子类可以重载特定的方法以在传送它们的时候修改事件流或配置请求。

xml.sax.saxutils.**prepare_input_source**(*source*[, *base*])
此函数接受一个输入源和一个可选的基准 URL 并返回一个经过完整解析可供读取的 *InputSource*。输入源的给出形式可以为字符串、文件类对象或 *InputSource* 对象；解析器将使用此函数来针对它们的 parse() 方法实现多态 *source* 参数。

19.14 xml.sax.xmlreader —Interface for XML parsers

2.0 新版功能.

SAX parsers implement the *XMLReader* interface. They are implemented in a Python module, which must provide a function `create_parser()`. This function is invoked by *xml.sax.make_parser()* with no arguments to create a new parser object.

class xml.sax.xmlreader.**XMLReader**
Base class which can be inherited by SAX parsers.

class xml.sax.xmlreader.**IncrementalParser**
In some cases, it is desirable not to parse an input source at once, but to feed chunks of the document as they get available. Note that the reader will normally not read the entire file, but read it in chunks as well; still `parse()` won't return until the entire document is processed. So these interfaces should be used if the blocking behaviour of `parse()` is not desirable.

When the parser is instantiated it is ready to begin accepting data from the feed method immediately. After parsing has been finished with a call to close the reset method must be called to make the parser ready to accept new data, either from feed or using the parse method.

Note that these methods must *not* be called during parsing, that is, after parse has been called and before it returns.

By default, the class also implements the parse method of the XMLReader interface using the feed, close and reset methods of the IncrementalParser interface as a convenience to SAX 2.0 driver writers.

class xml.sax.xmlreader.**Locator**
Interface for associating a SAX event with a document location. A locator object will return valid results only during calls to DocumentHandler methods; at any other time, the results are unpredictable. If information is not available, methods may return `None`.

class xml.sax.xmlreader.**InputSource**([*systemId*])
Encapsulation of the information needed by the *XMLReader* to read entities.

This class may include information about the public identifier, system identifier, byte stream (possibly with character encoding information) and/or the character stream of an entity.

Applications will create objects of this class for use in the *XMLReader.parse()* method and for returning from EntityResolver.resolveEntity.

An *InputSource* belongs to the application, the *XMLReader* is not allowed to modify *InputSource* objects passed to it from the application, although it may make copies and modify those.

class xml.sax.xmlreader.**AttributesImpl**(*attrs*)
This is an implementation of the `Attributes` interface (see section *The Attributes Interface*). This is a dictionary-like object which represents the element attributes in a `startElement()` call. In addition to the most useful dictionary operations, it supports a number of other methods as described by the interface. Objects of this class should be instantiated by readers; *attrs* must be a dictionary-like object containing a mapping from attribute names to attribute values.

class xml.sax.xmlreader.**AttributesNSImpl**(*attrs*, *qnames*)
Namespace-aware variant of *AttributesImpl*, which will be passed to `startElementNS()`. It is derived from *AttributesImpl*, but understands attribute names as two-tuples of *namespaceURI* and *localname*. In addition, it provides a number of methods expecting qualified names as they appear in the original document. This class implements the `AttributesNS` interface (see section *The AttributesNS Interface*).

19.14.1 XMLReader 对象

The *XMLReader* interface supports the following methods:

XMLReader.**parse**(*source*)
Process an input source, producing SAX events. The *source* object can be a system identifier (a string identifying the input source –typically a file name or a URL), a file-like object, or an *InputSource* object. When *parse()* returns, the input is completely processed, and the parser object can be discarded or reset. As a limitation, the current implementation only accepts byte streams; processing of character streams is for further study.

XMLReader.**getContentHandler**()
Return the current *ContentHandler*.

XMLReader.**setContentHandler**(*handler*)
Set the current *ContentHandler*. If no *ContentHandler* is set, content events will be discarded.

XMLReader.**getDTDHandler**()
Return the current *DTDHandler*.

XMLReader.**setDTDHandler**(*handler*)
Set the current *DTDHandler*. If no *DTDHandler* is set, DTD events will be discarded.

XMLReader.**getEntityResolver**()
Return the current *EntityResolver*.

XMLReader.**setEntityResolver**(*handler*)
Set the current *EntityResolver*. If no *EntityResolver* is set, attempts to resolve an external entity will result in opening the system identifier for the entity, and fail if it is not available.

XMLReader.**getErrorHandler**()
Return the current *ErrorHandler*.

XMLReader.**setErrorHandler**(*handler*)
Set the current error handler. If no *ErrorHandler* is set, errors will be raised as exceptions, and warnings will be printed.

XMLReader.**setLocale**(*locale*)
Allow an application to set the locale for errors and warnings.

SAX parsers are not required to provide localization for errors and warnings; if they cannot support the requested locale, however, they must raise a SAX exception. Applications may request a locale change in the middle of a parse.

XMLReader.**getFeature**(*featurename*)
Return the current setting for feature *featurename*. If the feature is not recognized, `SAXNotRecognizedException` is raised. The well-known featurenames are listed in the module *xml.sax.handler*.

XMLReader.**setFeature**(*featurename*, *value*)
Set the *featurename* to *value*. If the feature is not recognized, `SAXNotRecognizedException` is raised. If the feature or its setting is not supported by the parser, *SAXNotSupportedException* is raised.

XMLReader.**getProperty**(*propertyname*)
Return the current setting for property *propertyname*. If the property is not recognized, a `SAXNotRecognizedException` is raised. The well-known propertynames are listed in the module *xml.sax.handler*.

XMLReader.**setProperty**(*propertyname*, *value*)
Set the *propertyname* to *value*. If the property is not recognized, `SAXNotRecognizedException` is raised. If the property or its setting is not supported by the parser, *SAXNotSupportedException* is raised.

19.14.2 IncrementalParser 对象

Instances of *IncrementalParser* offer the following additional methods:

IncrementalParser.**feed**(*data*)
Process a chunk of *data*.

IncrementalParser.**close**()
Assume the end of the document. That will check well-formedness conditions that can be checked only at the end, invoke handlers, and may clean up resources allocated during parsing.

IncrementalParser.**reset**()
This method is called after close has been called to reset the parser so that it is ready to parse new documents. The results of calling parse or feed after close without calling reset are undefined.

19.14.3 Locator 对象

Instances of *Locator* provide these methods:

Locator.**getColumnNumber**()
Return the column number where the current event begins.

Locator.**getLineNumber**()
Return the line number where the current event begins.

Locator.**getPublicId**()
Return the public identifier for the current event.

Locator.**getSystemId**()
Return the system identifier for the current event.

19.14.4 InputSource 对象

InputSource.**setPublicId**(*id*)
Sets the public identifier of this *InputSource*.

InputSource.**getPublicId**()
Returns the public identifier of this *InputSource*.

InputSource.**setSystemId**(*id*)
Sets the system identifier of this *InputSource*.

InputSource.**getSystemId**()
Returns the system identifier of this *InputSource*.

InputSource.**setEncoding**(*encoding*)
Sets the character encoding of this *InputSource*.

The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML recommendation).

The encoding attribute of the *InputSource* is ignored if the *InputSource* also contains a character stream.

InputSource.**getEncoding**()
Get the character encoding of this InputSource.

InputSource.**setByteStream**(*bytefile*)
Set the byte stream (a Python file-like object which does not perform byte-to-character conversion) for this input source.

The SAX parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself.

If the application knows the character encoding of the byte stream, it should set it with the setEncoding method.

`InputSource.`**`getByteStream`**`()`
Get the byte stream for this input source.

The getEncoding method will return the character encoding for this byte stream, or `None` if unknown.

`InputSource.`**`setCharacterStream`**`(`*charfile*`)`
Set the character stream for this input source. (The stream must be a Python 1.6 Unicode-wrapped file-like that performs conversion to Unicode strings.)

If there is a character stream specified, the SAX parser will ignore any byte stream and will not attempt to open a URI connection to the system identifier.

`InputSource.`**`getCharacterStream`**`()`
Get the character stream for this input source.

19.14.5 The `Attributes` Interface

`Attributes` objects implement a portion of the mapping protocol, including the methods `copy()`, `get()`, `has_key()`, `items()`, `keys()`, and `values()`. The following methods are also provided:

`Attributes.`**`getLength`**`()`
Return the number of attributes.

`Attributes.`**`getNames`**`()`
Return the names of the attributes.

`Attributes.`**`getType`**`(`*name*`)`
Returns the type of the attribute *name*, which is normally `'CDATA'`.

`Attributes.`**`getValue`**`(`*name*`)`
Return the value of attribute *name*.

19.14.6 The `AttributesNS` Interface

This interface is a subtype of the `Attributes` interface (see section *The Attributes Interface*). All methods supported by that interface are also available on `AttributesNS` objects.

The following methods are also available:

`AttributesNS.`**`getValueByQName`**`(`*name*`)`
Return the value for a qualified name.

`AttributesNS.`**`getNameByQName`**`(`*name*`)`
Return the `(namespace, localname)` pair for a qualified *name*.

`AttributesNS.`**`getQNameByName`**`(`*name*`)`
Return the qualified name for a `(namespace, localname)` pair.

`AttributesNS.`**`getQNames`**`()`
Return the qualified names of all attributes.

19.15 xml.parsers.expat —Fast XML parsing using Expat

警告： The pyexpat module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML 漏洞*.

2.0 新版功能.

The *xml.parsers.expat* module is a Python interface to the Expat non-validating XML parser. The module provides a single extension type, xmlparser, that represents the current state of an XML parser. After an xmlparser object has been created, various attributes of the object can be set to handler functions. When an XML document is then fed to the parser, the handler functions are called for the character data and markup in the XML document.

This module uses the pyexpat module to provide access to the Expat parser. Direct use of the pyexpat module is deprecated.

This module provides one exception and one type object:

exception xml.parsers.expat.**ExpatError**
The exception raised when Expat reports an error. See section *ExpatError Exceptions* for more information on interpreting Expat errors.

exception xml.parsers.expat.**error**
Alias for *ExpatError*.

xml.parsers.expat.**XMLParserType**
The type of the return values from the *ParserCreate()* function.

The *xml.parsers.expat* module contains two functions:

xml.parsers.expat.**ErrorString**(*errno*)
Returns an explanatory string for a given error number *errno*.

xml.parsers.expat.**ParserCreate**([*encoding*[, *namespace_separator*]])
Creates and returns a new xmlparser object. *encoding*, if specified, must be a string naming the encoding used by the XML data. Expat doesn't support as many encodings as Python does, and its repertoire of encodings can't be extended; it supports UTF-8, UTF-16, ISO-8859-1 (Latin1), and ASCII. If *encoding*[1] is given it will override the implicit or explicit encoding of the document.

Expat can optionally do XML namespace processing for you, enabled by providing a value for *namespace_separator*. The value must be a one-character string; a *ValueError* will be raised if the string has an illegal length (None is considered the same as omission). When namespace processing is enabled, element type names and attribute names that belong to a namespace will be expanded. The element name passed to the element handlers StartElementHandler and EndElementHandler will be the concatenation of the namespace URI, the namespace separator character, and the local part of the name. If the namespace separator is a zero byte (chr(0)) then the namespace URI and the local part will be concatenated without any separator.

For example, if *namespace_separator* is set to a space character (' ') and the following document is parsed:

```
<?xml version="1.0"?>
<root xmlns    = "http://default-namespace.org/"
      xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

[1] The encoding string included in XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

`StartElementHandler` will receive the following strings for each element:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

Due to limitations in the `Expat` library used by `pyexpat`, the `xmlparser` instance returned can only be used to parse a single XML document. Call `ParserCreate` for each document to provide unique parser instances.

参见:

The Expat XML Parser Home page of the Expat project.

19.15.1 XMLParser 对象

`xmlparser` objects have the following methods:

`xmlparser.`**`Parse`**(*data*[, *isfinal*])
Parses the contents of the string *data*, calling the appropriate handler functions to process the parsed data. *isfinal* must be true on the final call to this method; it allows the parsing of a single file in fragments, not the submission of multiple files. *data* can be the empty string at any time.

`xmlparser.`**`ParseFile`**(*file*)
Parse XML data reading from the object *file*. *file* only needs to provide the `read(nbytes)` method, returning the empty string when there's no more data.

`xmlparser.`**`SetBase`**(*base*)
Sets the base to be used for resolving relative URIs in system identifiers in declarations. Resolving relative identifiers is left to the application: this value will be passed through as the *base* argument to the *`ExternalEntityRefHandler()`*, *`NotationDeclHandler()`*, and *`UnparsedEntityDeclHandler()`* functions.

`xmlparser.`**`GetBase`**()
Returns a string containing the base set by a previous call to *`SetBase()`*, or `None` if *`SetBase()`* hasn't been called.

`xmlparser.`**`GetInputContext`**()
Returns the input data that generated the current event as a string. The data is in the encoding of the entity which contains the text. When called while an event handler is not active, the return value is `None`.

2.1 新版功能.

`xmlparser.`**`ExternalEntityParserCreate`**(*context*[, *encoding*])
Create a "child" parser which can be used to parse an external parsed entity referred to by content parsed by the parent parser. The *context* parameter should be the string passed to the *`ExternalEntityRefHandler()`* handler function, described below. The child parser is created with the *`ordered_attributes`*, *`returns_unicode`* and *`specified_attributes`* set to the values of this parser.

`xmlparser.`**`SetParamEntityParsing`**(*flag*)
Control parsing of parameter entities (including the external DTD subset). Possible *flag* values are `XML_PARAM_ENTITY_PARSING_NEVER`, `XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE` and `XML_PARAM_ENTITY_PARSING_ALWAYS`. Return true if setting the flag was successful.

`xmlparser.`**`UseForeignDTD`**([*flag*])
Calling this with a true value for *flag* (the default) will cause Expat to call the *`ExternalEntityRefHandler`* with *`None`* for all arguments to allow an alternate DTD to be loaded. If the document does not contain a document type declaration, the *`ExternalEntityRefHandler`* will still be called, but the *`StartDoctypeDeclHandler`* and *`EndDoctypeDeclHandler`* will not be called.

Passing a false value for *flag* will cancel a previous call that passed a true value, but otherwise has no effect.

This method can only be called before the *Parse()* or *ParseFile()* methods are called; calling it after either of those have been called causes *ExpatError* to be raised with the *code* attribute set to `errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING`.

2.3 新版功能.

`xmlparser` objects have the following attributes:

`xmlparser.`**buffer_size**
The size of the buffer used when *buffer_text* is true. A new buffer size can be set by assigning a new integer value to this attribute. When the size is changed, the buffer will be flushed.

2.3 新版功能.

在 2.6 版更改: The buffer size can now be changed.

`xmlparser.`**buffer_text**
Setting this to true causes the `xmlparser` object to buffer textual content returned by Expat to avoid multiple calls to the *CharacterDataHandler()* callback whenever possible. This can improve performance substantially since Expat normally breaks character data into chunks at every line ending. This attribute is false by default, and may be changed at any time.

2.3 新版功能.

`xmlparser.`**buffer_used**
If *buffer_text* is enabled, the number of bytes stored in the buffer. These bytes represent UTF-8 encoded text. This attribute has no meaningful interpretation when *buffer_text* is false.

2.3 新版功能.

`xmlparser.`**ordered_attributes**
Setting this attribute to a non-zero integer causes the attributes to be reported as a list rather than a dictionary. The attributes are presented in the order found in the document text. For each attribute, two list entries are presented: the attribute name and the attribute value. (Older versions of this module also used this format.) By default, this attribute is false; it may be changed at any time.

2.1 新版功能.

`xmlparser.`**returns_unicode**
If this attribute is set to a non-zero integer, the handler functions will be passed Unicode strings. If *returns_unicode* is *False*, 8-bit strings containing UTF-8 encoded data will be passed to the handlers. This is *True* by default when Python is built with Unicode support.

在 1.6 版更改: Can be changed at any time to affect the result type.

`xmlparser.`**specified_attributes**
If set to a non-zero integer, the parser will report only those attributes which were specified in the document instance and not those which were derived from attribute declarations. Applications which set this need to be especially careful to use what additional information is available from the declarations as needed to comply with the standards for the behavior of XML processors. By default, this attribute is false; it may be changed at any time.

2.1 新版功能.

The following attributes contain values relating to the most recent error encountered by an `xmlparser` object, and will only have correct values once a call to `Parse()` or `ParseFile()` has raised an *xml.parsers.expat.ExpatError* exception.

`xmlparser.`**ErrorByteIndex**
Byte index at which an error occurred.

xmlparser.**ErrorCode**
Numeric code specifying the problem. This value can be passed to the *ErrorString()* function, or compared to one of the constants defined in the errors object.

xmlparser.**ErrorColumnNumber**
Column number at which an error occurred.

xmlparser.**ErrorLineNumber**
Line number at which an error occurred.

The following attributes contain values relating to the current parse location in an xmlparser object. During a callback reporting a parse event they indicate the location of the first of the sequence of characters that generated the event. When called outside of a callback, the position indicated will be just past the last parse event (regardless of whether there was an associated callback).

2.4 新版功能.

xmlparser.**CurrentByteIndex**
Current byte index in the parser input.

xmlparser.**CurrentColumnNumber**
Current column number in the parser input.

xmlparser.**CurrentLineNumber**
Current line number in the parser input.

Here is the list of handlers that can be set. To set a handler on an xmlparser object *o*, use o.handlername = func. *handlername* must be taken from the following list, and *func* must be a callable object accepting the correct number of arguments. The arguments are all strings, unless otherwise stated.

xmlparser.**XmlDeclHandler**(*version*, *encoding*, *standalone*)
Called when the XML declaration is parsed. The XML declaration is the (optional) declaration of the applicable version of the XML recommendation, the encoding of the document text, and an optional "standalone" declaration. *version* and *encoding* will be strings of the type dictated by the *returns_unicode* attribute, and *standalone* will be 1 if the document is declared standalone, 0 if it is declared not to be standalone, or -1 if the standalone clause was omitted. This is only available with Expat version 1.95.0 or newer.

2.1 新版功能.

xmlparser.**StartDoctypeDeclHandler**(*doctypeName*, *systemId*, *publicId*, *has_internal_subset*)
Called when Expat begins parsing the document type declaration (<!DOCTYPE ...). The *doctypeName* is provided exactly as presented. The *systemId* and *publicId* parameters give the system and public identifiers if specified, or None if omitted. *has_internal_subset* will be true if the document contains and internal document declaration subset. This requires Expat version 1.2 or newer.

xmlparser.**EndDoctypeDeclHandler**()
Called when Expat is done parsing the document type declaration. This requires Expat version 1.2 or newer.

xmlparser.**ElementDeclHandler**(*name*, *model*)
Called once for each element type declaration. *name* is the name of the element type, and *model* is a representation of the content model.

xmlparser.**AttlistDeclHandler**(*elname*, *attname*, *type*, *default*, *required*)
Called for each declared attribute for an element type. If an attribute list declaration declares three attributes, this handler is called three times, once for each attribute. *elname* is the name of the element to which the declaration applies and *attname* is the name of the attribute declared. The attribute type is a string passed as *type*; the possible values are 'CDATA', 'ID', 'IDREF', ... *default* gives the default value for the attribute used when the attribute is not specified by the document instance, or None if there is no default value (#IMPLIED values). If the attribute is required to be given in the document instance, *required* will be true. This requires Expat version 1.95.0 or newer.

xmlparser.**StartElementHandler**(*name*, *attributes*)
Called for the start of every element. *name* is a string containing the element name, and *attributes* is a dictionary mapping attribute names to their values.

xmlparser.**EndElementHandler**(*name*)
Called for the end of every element.

xmlparser.**ProcessingInstructionHandler**(*target*, *data*)
Called for every processing instruction.

xmlparser.**CharacterDataHandler**(*data*)
Called for character data. This will be called for normal character data, CDATA marked content, and ignorable whitespace. Applications which must distinguish these cases can use the *StartCdataSectionHandler*, *EndCdataSectionHandler*, and *ElementDeclHandler* callbacks to collect the required information.

xmlparser.**UnparsedEntityDeclHandler**(*entityName*, *base*, *systemId*, *publicId*, *notationName*)
Called for unparsed (NDATA) entity declarations. This is only present for version 1.2 of the Expat library; for more recent versions, use *EntityDeclHandler* instead. (The underlying function in the Expat library has been declared obsolete.)

xmlparser.**EntityDeclHandler**(*entityName*, *is_parameter_entity*, *value*, *base*, *systemId*, *publicId*, *notationName*)
Called for all entity declarations. For parameter and internal entities, *value* will be a string giving the declared contents of the entity; this will be `None` for external entities. The *notationName* parameter will be `None` for parsed entities, and the name of the notation for unparsed entities. *is_parameter_entity* will be true if the entity is a parameter entity or false for general entities (most applications only need to be concerned with general entities). This is only available starting with version 1.95.0 of the Expat library.

2.1 新版功能.

xmlparser.**NotationDeclHandler**(*notationName*, *base*, *systemId*, *publicId*)
Called for notation declarations. *notationName*, *base*, and *systemId*, and *publicId* are strings if given. If the public identifier is omitted, *publicId* will be `None`.

xmlparser.**StartNamespaceDeclHandler**(*prefix*, *uri*)
Called when an element contains a namespace declaration. Namespace declarations are processed before the *StartElementHandler* is called for the element on which declarations are placed.

xmlparser.**EndNamespaceDeclHandler**(*prefix*)
Called when the closing tag is reached for an element that contained a namespace declaration. This is called once for each namespace declaration on the element in the reverse of the order for which the *StartNamespaceDeclHandler* was called to indicate the start of each namespace declaration's scope. Calls to this handler are made after the corresponding *EndElementHandler* for the end of the element.

xmlparser.**CommentHandler**(*data*)
Called for comments. *data* is the text of the comment, excluding the leading `'<!--'` and trailing `'-->'`.

xmlparser.**StartCdataSectionHandler**()
Called at the start of a CDATA section. This and *EndCdataSectionHandler* are needed to be able to identify the syntactical start and end for CDATA sections.

xmlparser.**EndCdataSectionHandler**()
Called at the end of a CDATA section.

xmlparser.**DefaultHandler**(*data*)
Called for any characters in the XML document for which no applicable handler has been specified. This means characters that are part of a construct which could be reported, but for which no handler has been supplied.

xmlparser.**DefaultHandlerExpand**(*data*)
This is the same as the *DefaultHandler()*, but doesn't inhibit expansion of internal entities. The entity reference will not be passed to the default handler.

xmlparser.**NotStandaloneHandler**()
Called if the XML document hasn't been declared as being a standalone document. This happens when there is an external subset or a reference to a parameter entity, but the XML declaration does not set standalone to `yes` in an XML declaration. If this handler returns `0`, then the parser will raise an `XML_ERROR_NOT_STANDALONE` error. If this handler is not set, no exception is raised by the parser for this condition.

xmlparser.**ExternalEntityRefHandler**(*context*, *base*, *systemId*, *publicId*)
Called for references to external entities. *base* is the current base, as set by a previous call to *SetBase()*. The public and system identifiers, *systemId* and *publicId*, are strings if given; if the public identifier is not given, *publicId* will be `None`. The *context* value is opaque and should only be used as described below.

For external entities to be parsed, this handler must be implemented. It is responsible for creating the sub-parser using `ExternalEntityParserCreate(context)`, initializing it with the appropriate callbacks, and parsing the entity. This handler should return an integer; if it returns `0`, the parser will raise an `XML_ERROR_EXTERNAL_ENTITY_HANDLING` error, otherwise parsing will continue.

If this handler is not provided, external entities are reported by the *DefaultHandler* callback, if provided.

19.15.2 ExpatError Exceptions

ExpatError exceptions have a number of interesting attributes:

ExpatError.**code**
Expat's internal error number for the specific error. This will match one of the constants defined in the `errors` object from this module.

2.1 新版功能.

ExpatError.**lineno**
Line number on which the error was detected. The first line is numbered `1`.

2.1 新版功能.

ExpatError.**offset**
Character offset into the line where the error occurred. The first column is numbered `0`.

2.1 新版功能.

19.15.3 示例

The following program defines three handlers that just print out their arguments.

```
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print 'Start element:', name, attrs
def end_element(name):
    print 'End element:', name
def char_data(data):
    print 'Character data:', repr(data)

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data
```

(下页继续)

(续上页)

```
p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

The output from this program is:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

19.15.4 Content Model Descriptions

Content models are described using nested tuples. Each tuple contains four values: the type, the quantifier, the name, and a tuple of children. Children are simply additional content model descriptions.

The values of the first two fields are constants defined in the `model` object of the *`xml.parsers.expat`* module. These constants can be collected in two groups: the model type group and the quantifier group.

The constants in the model type group are:

`xml.parsers.expat.`**`XML_CTYPE_ANY`**
: The element named by the model name was declared to have a content model of `ANY`.

`xml.parsers.expat.`**`XML_CTYPE_CHOICE`**
: The named element allows a choice from a number of options; this is used for content models such as `(A | B | C)`.

`xml.parsers.expat.`**`XML_CTYPE_EMPTY`**
: Elements which are declared to be `EMPTY` have this model type.

`xml.parsers.expat.`**`XML_CTYPE_MIXED`**

`xml.parsers.expat.`**`XML_CTYPE_NAME`**

`xml.parsers.expat.`**`XML_CTYPE_SEQ`**
: Models which represent a series of models which follow one after the other are indicated with this model type. This is used for models such as `(A, B, C)`.

The constants in the quantifier group are:

`xml.parsers.expat.`**`XML_CQUANT_NONE`**
: No modifier is given, so it can appear exactly once, as for `A`.

`xml.parsers.expat.`**`XML_CQUANT_OPT`**
: The model is optional: it can appear once or not at all, as for `A?`.

`xml.parsers.expat.`**`XML_CQUANT_PLUS`**
: The model must occur one or more times (like `A+`).

xml.parsers.expat.**XML_CQUANT_REP**
The model must occur zero or more times, as for `A*`.

19.15.5 Expat error constants

The following constants are provided in the `errors` object of the *xml.parsers.expat* module. These constants are useful in interpreting some of the attributes of the *ExpatError* exception objects raised when an error has occurred.

The `errors` object has the following attributes:

xml.parsers.expat.**XML_ERROR_ASYNC_ENTITY**

xml.parsers.expat.**XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF**
An entity reference in an attribute value referred to an external entity instead of an internal entity.

xml.parsers.expat.**XML_ERROR_BAD_CHAR_REF**
A character reference referred to a character which is illegal in XML (for example, character `0`, or '`&#0;`').

xml.parsers.expat.**XML_ERROR_BINARY_ENTITY_REF**
An entity reference referred to an entity which was declared with a notation, so cannot be parsed.

xml.parsers.expat.**XML_ERROR_DUPLICATE_ATTRIBUTE**
An attribute was used more than once in a start tag.

xml.parsers.expat.**XML_ERROR_INCORRECT_ENCODING**

xml.parsers.expat.**XML_ERROR_INVALID_TOKEN**
Raised when an input byte could not properly be assigned to a character; for example, a NUL byte (value `0`) in a UTF-8 input stream.

xml.parsers.expat.**XML_ERROR_JUNK_AFTER_DOC_ELEMENT**
Something other than whitespace occurred after the document element.

xml.parsers.expat.**XML_ERROR_MISPLACED_XML_PI**
An XML declaration was found somewhere other than the start of the input data.

xml.parsers.expat.**XML_ERROR_NO_ELEMENTS**
The document contains no elements (XML requires all documents to contain exactly one top-level element)..

xml.parsers.expat.**XML_ERROR_NO_MEMORY**
Expat was not able to allocate memory internally.

xml.parsers.expat.**XML_ERROR_PARAM_ENTITY_REF**
A parameter entity reference was found where it was not allowed.

xml.parsers.expat.**XML_ERROR_PARTIAL_CHAR**
An incomplete character was found in the input.

xml.parsers.expat.**XML_ERROR_RECURSIVE_ENTITY_REF**
An entity reference contained another reference to the same entity; possibly via a different name, and possibly indirectly.

xml.parsers.expat.**XML_ERROR_SYNTAX**
Some unspecified syntax error was encountered.

xml.parsers.expat.**XML_ERROR_TAG_MISMATCH**
An end tag did not match the innermost open start tag.

xml.parsers.expat.**XML_ERROR_UNCLOSED_TOKEN**
Some token (such as a start tag) was not closed before the end of the stream or the next token was encountered.

xml.parsers.expat.**XML_ERROR_UNDEFINED_ENTITY**
A reference was made to an entity which was not defined.

xml.parsers.expat.**XML_ERROR_UNKNOWN_ENCODING**
The document encoding is not supported by Expat.

xml.parsers.expat.**XML_ERROR_UNCLOSED_CDATA_SECTION**
A CDATA marked section was not closed.

xml.parsers.expat.**XML_ERROR_EXTERNAL_ENTITY_HANDLING**

xml.parsers.expat.**XML_ERROR_NOT_STANDALONE**
The parser determined that the document was not "standalone" though it declared itself to be in the XML declaration, and the `NotStandaloneHandler` was set and returned `0`.

xml.parsers.expat.**XML_ERROR_UNEXPECTED_STATE**

xml.parsers.expat.**XML_ERROR_ENTITY_DECLARED_IN_PE**

xml.parsers.expat.**XML_ERROR_FEATURE_REQUIRES_XML_DTD**
An operation was requested that requires DTD support to be compiled in, but Expat was configured without DTD support. This should never be reported by a standard build of the *xml.parsers.expat* module.

xml.parsers.expat.**XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING**
A behavioral change was requested after parsing started that can only be changed before parsing has started. This is (currently) only raised by `UseForeignDTD()`.

xml.parsers.expat.**XML_ERROR_UNBOUND_PREFIX**
An undeclared prefix was found when namespace processing was enabled.

xml.parsers.expat.**XML_ERROR_UNDECLARING_PREFIX**
The document attempted to remove the namespace declaration associated with a prefix.

xml.parsers.expat.**XML_ERROR_INCOMPLETE_PE**
A parameter entity contained incomplete markup.

xml.parsers.expat.**XML_ERROR_XML_DECL**
The document contained no document element at all.

xml.parsers.expat.**XML_ERROR_TEXT_DECL**
There was an error parsing a text declaration in an external entity.

xml.parsers.expat.**XML_ERROR_PUBLICID**
Characters were found in the public id that are not allowed.

xml.parsers.expat.**XML_ERROR_SUSPENDED**
The requested operation was made on a suspended parser, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.**XML_ERROR_NOT_SUSPENDED**
An attempt to resume the parser was made when the parser had not been suspended.

xml.parsers.expat.**XML_ERROR_ABORTED**
This should not be reported to Python applications.

xml.parsers.expat.**XML_ERROR_FINISHED**
The requested operation was made on a parser which was finished parsing input, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.**XML_ERROR_SUSPEND_PE**

备注

CHAPTER 20

互联网协议和支持

本章介绍的模块实现了互联网协议并支持相关技术。它们都是用 Python 实现的。这些模块中的大多数都需要存在依赖于系统的模块*socket*，目前大多数流行平台都支持它。这是一个概述：

20.1 webbrowser —方便的 Web 浏览器控制器

源码： Lib/webbrowser.py

webbrowser 模块提供了一个高级接口，允许向用户显示基于 Web 的文档。在大多数情况下，只需从该模块调用*open()* 函数就可以了。

在 Unix 下，图形浏览器在 X11 下是首选，但如果图形浏览器不可用或 X11 显示不可用，则将使用文本模式浏览器。如果使用文本模式浏览器，则调用进程将阻塞，直到用户退出浏览器。

If the environment variable BROWSER exists, it is interpreted to override the platform default list of browsers, as an *os.pathsep*-separated list of browsers to try in order. When the value of a list part contains the string %s, then it is interpreted as a literal browser command line to be used with the argument URL substituted for %s; if the part does not contain %s, it is simply interpreted as the name of the browser to launch.[1]

对于非 Unix 平台，或者当 Unix 上有远程浏览器时，控制过程不会等待用户完成浏览器，而是允许远程浏览器在显示界面上维护自己的窗口。如果 Unix 上没有远程浏览器，控制进程将启动一个新的浏览器并等待。

脚本 **webbrowser** 可以用作模块的命令行界面。它接受一个 URL 作为参数。还接受以下可选参数：-n 如果可能，在新的浏览器窗口中打开 URL；-t 在新的浏览器页面（“标签”）中打开 URL。这些选择当然是相互排斥的。用法示例：

```
python -m webbrowser -t "http://www.python.org"
```

定义了以下异常：

exception webbrowser.**Error**
发生浏览器控件错误时引发异常。

[1] 这里命名的不带完整路径的可执行文件将在 PATH 环境变量给出的目录中搜索。

定义了以下函数：

webbrowser.**open**(*url*, *new=0*, *autoraise=True*)
使用默认浏览器显示 *url*。如果 *new* 为 0，则尽可能在同一浏览器窗口中打开 *url*。如果 *new* 为 1，则尽可能打开新的浏览器窗口。如果 *new* 为 2，则尽可能打开新的浏览器页面（“标签”）。如果 *autoraise* 为 “True”，则会尽可能置前窗口（请注意，在许多窗口管理器下，无论此变量的设置如何，都会置前窗口）。

请注意，在某些平台上，尝试使用此函数打开文件名，可能会起作用并启动操作系统的关联程序。但是，这种方式不被支持也不可移植。

在 2.5 版更改: *new* can now be 2.

webbrowser.**open_new**(*url*)
如果可能，在默认浏览器的新窗口中打开 *url*，否则，在唯一的浏览器窗口中打开 *url*。

webbrowser.**open_new_tab**(*url*)
如果可能，在默认浏览器的新页面（“标签”）中打开 *url*，否则等效于*open_new()*。

2.5 新版功能.

webbrowser.**get**([*name*])
Return a controller object for the browser type *name*. If *name* is empty, return a controller for a default browser appropriate to the caller’s environment.

webbrowser.**register**(*name*, *constructor*[, *instance*])
注册 *name* 浏览器类型。注册浏览器类型后，*get()* 函数可以返回该浏览器类型的控制器。如果没有提供 *instance*，或者为 None，*constructor* 将在没有参数的情况下被调用，以在需要时创建实例。如果提供了 *instance*，则永远不会调用 *constructor*，并且可能是 None。

This entry point is only useful if you plan to either set the BROWSER variable or call *get()* with a nonempty argument matching the name of a handler you declare.

预定义了许多浏览器类型。此表给出了可以传递给*get()* 函数的类型名称以及控制器类的相应实例化，这些都在此模块中定义。

类型名	类名	注释
'mozilla'	Mozilla('mozilla')	
'firefox'	Mozilla('mozilla')	
'netscape'	Mozilla('netscape')	
'galeon'	Galeon('galeon')	
'epiphany'	Galeon('epiphany')	
'skipstone'	BackgroundBrowser('skipstone')	
'kfmclient'	Konqueror()	(1)
'konqueror'	Konqueror()	(1)
'kfm'	Konqueror()	(1)
'mosaic'	BackgroundBrowser('mosaic')	
'opera'	Opera()	
'grail'	Grail()	
'links'	GenericBrowser('links')	
'elinks'	Elinks('elinks')	
'lynx'	GenericBrowser('lynx')	
'w3m'	GenericBrowser('w3m')	
'windows-default'	WindowsDefault	(2)
'macosx'	MacOSX('default')	(3)
'safari'	MacOSX('safari')	(3)
'google-chrome'	Chrome('google-chrome')	(4)
'chrome'	Chrome('chrome')	(4)
'chromium'	Chromium('chromium')	(4)
'chromium-browser'	Chromium('chromium-browser')	(4)

注释:

(1) “Konqueror” 是 Unix 的 KDE 桌面环境的文件管理器，只有在 KDE 运行时才有意义。一些可靠地检测 KDE 的方法会很好；仅检查 KDEDIR 变量是不够的。另请注意，KDE 2 的 **konqueror** 命令，会使用名称 “kfm” —此实现选择运行的 Konqueror 的最佳策略。

(2) 仅限 Windows 平台。

(3) 仅限 Mac OS X 平台。

(4) Support for Chrome/Chromium has been added in version 2.7.5.

以下是一些简单的例子:

```
url = 'http://www.python.org/'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url + 'doc/')

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

20.1.1 浏览器控制器对象

浏览器控制器提供三个与模块级便捷函数相同的方法：

`controller.`**`open`**(*url*, *new=0*, *autoraise=True*)
使用此控制器处理的浏览器显示 *url*。如果 *new* 为 1，则尽可能打开新的浏览器窗口。如果 *new* 为 2，则尽可能打开新的浏览器页面（“标签”）。

`controller.`**`open_new`**(*url*)
如果可能，在此控制器处理的浏览器的新窗口中打开 *url* ，否则，在唯一的浏览器窗口中打开 *url* 。别名*open_new()*。

`controller.`**`open_new_tab`**(*url*)
如果可能，在此控制器处理的浏览器的新页面（“标签”）中打开 *url*，否则等效于*open_new()*。

2.5 新版功能.

备注

20.2 `cgi` —Common Gateway Interface support

源代码： Lib/cgi.py

Support module for Common Gateway Interface (CGI) scripts.

This module defines a number of utilities for use by CGI scripts written in Python.

20.2.1 概述

A CGI script is invoked by an HTTP server, usually to process user input submitted through an HTML `<FORM>` or `<ISINDEX>` element.

Most often, CGI scripts live in the server’s special `cgi-bin` directory. The HTTP server places all sorts of information about the request (such as the client’s hostname, the requested URL, the query string, and lots of other goodies) in the script’s shell environment, executes the script, and sends the script’s output back to the client.

The script’s input is connected to the client too, and sometimes the form data is read this way; at other times the form data is passed via the “query string” part of the URL. This module is intended to take care of the different cases and provide a simpler interface to the Python script. It also provides a number of utilities that help in debugging scripts, and the latest addition is support for file uploads from a form (if your browser supports it).

The output of a CGI script should consist of two sections, separated by a blank line. The first section contains a number of headers, telling the client what kind of data is following. Python code to generate a minimal header section looks like this:

```
print "Content-Type: text/html"     # HTML is following
print                               # blank line, end of headers
```

The second section is usually HTML, which allows the client software to display nicely formatted text with header, in-line images, etc. Here’s Python code that prints a simple piece of HTML:

```
print "<TITLE>CGI script output</TITLE>"
print "<H1>This is my first CGI script</H1>"
print "Hello, world!"
```

20.2.2 使用 cgi 模块。

Begin by writing `import cgi`. Do not use `from cgi import *` —the module defines all sorts of names for its own use or for backward compatibility that you don't want in your namespace.

当你在写一个新脚本时，考虑加上这些语句:

```
import cgitb
cgitb.enable()
```

This activates a special exception handler that will display detailed reports in the Web browser if any errors occur. If you'd rather not show the guts of your program to users of your script, you can have the reports saved to files instead, with code like this:

```
import cgitb
cgitb.enable(display=0, logdir="/path/to/logdir")
```

It's very helpful to use this feature during script development. The reports produced by *cgitb* provide information that can save you a lot of time in tracking down bugs. You can always remove the `cgitb` line later when you have tested your script and are confident that it works correctly.

To get at submitted form data, it's best to use the `FieldStorage` class. The other classes defined in this module are provided mostly for backward compatibility. Instantiate it exactly once, without arguments. This reads the form contents from standard input or the environment (depending on the value of various environment variables set according to the CGI standard). Since it may consume standard input, it should be instantiated only once.

The `FieldStorage` instance can be indexed like a Python dictionary. It allows membership testing with the `in` operator, and also supports the standard dictionary method *keys()* and the built-in function *len()*. Form fields containing empty strings are ignored and do not appear in the dictionary; to keep such values, provide a true value for the optional *keep_blank_values* keyword parameter when creating the `FieldStorage` instance.

For instance, the following code (which assumes that the *Content-Type* header and blank line have already been printed) checks that the fields `name` and `addr` are both set to a non-empty string:

```
form = cgi.FieldStorage()
if "name" not in form or "addr" not in form:
    print "<H1>Error</H1>"
    print "Please fill in the name and addr fields."
    return
print "<p>name:", form["name"].value
print "<p>addr:", form["addr"].value
...further form processing here...
```

Here the fields, accessed through `form[key]`, are themselves instances of `FieldStorage` (or `MiniFieldStorage`, depending on the form encoding). The `value` attribute of the instance yields the string value of the field. The `getvalue()` method returns this string value directly; it also accepts an optional second argument as a default to return if the requested key is not present.

If the submitted form data contains more than one field with the same name, the object retrieved by `form[key]` is not a `FieldStorage` or `MiniFieldStorage` instance but a list of such instances. Similarly, in this situation, `form.getvalue(key)` would return a list of strings. If you expect this possibility (when your HTML form contains multiple fields with the same name), use the *getlist()* method, which always returns a list of values (so that you do not need to special-case the single item case). For example, this code concatenates any number of username fields, separated by commas:

```
value = form.getlist("username")
usernames = ",".join(value)
```

If a field represents an uploaded file, accessing the value via the `value` attribute or the `getvalue()` method reads the entire file in memory as a string. This may not be what you want. You can test for an uploaded file by testing either the `filename` attribute or the `file` attribute. You can then read the data at leisure from the `file` attribute:

```
fileitem = form["userfile"]
if fileitem.file:
    # It's an uploaded file; count lines
    linecount = 0
    while 1:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```

If an error is encountered when obtaining the contents of an uploaded file (for example, when the user interrupts the form submission by clicking on a Back or Cancel button) the `done` attribute of the object for the field will be set to the value -1.

The file upload draft standard entertains the possibility of uploading multiple files from one field (using a recursive *multipart/** encoding). When this occurs, the item will be a dictionary-like `FieldStorage` item. This can be determined by testing its `type` attribute, which should be *multipart/form-data* (or perhaps another MIME type matching *multipart/**). In this case, it can be iterated over recursively just like the top-level form object.

When a form is submitted in the "old" format (as the query string or as a single data part of type *application/x-www-form-urlencoded*), the items will actually be instances of the class `MiniFieldStorage`. In this case, the `list`, `file`, and `filename` attributes are always `None`.

A form submitted via POST that also has a query string will contain both `FieldStorage` and `MiniFieldStorage` items.

20.2.3 Higher Level Interface

2.2 新版功能.

The previous section explains how to read CGI form data using the `FieldStorage` class. This section describes a higher level interface which was added to this class to allow one to do it in a more readable and intuitive way. The interface doesn't make the techniques described in previous sections obsolete —they are still useful to process file uploads efficiently, for example.

The interface consists of two simple methods. Using the methods you can process form data in a generic way, without the need to worry whether only one or more values were posted under one name.

In the previous section, you learned to write following code anytime you expected a user to post more than one value under one name:

```
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

This situation is common for example when a form contains a group of multiple checkboxes with the same name:

```
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

In most situations, however, there's only one form control with a particular name in a form and then you expect and need only one value associated with this name. So you write a script containing for example this code:

```
user = form.getvalue("user").upper()
```

The problem with the code is that you should never expect that a client will provide valid input to your scripts. For example, if a curious user appends another `user=foo` pair to the query string, then the script would crash, because in this situation the `getvalue("user")` method call returns a list instead of a string. Calling the *upper()* method on a list is not valid (since lists do not have a method of this name) and results in an *AttributeError* exception.

Therefore, the appropriate way to read form data values was to always use the code which checks whether the obtained value is a single value or a list of values. That's annoying and leads to less readable scripts.

A more convenient approach is to use the methods *getfirst()* and *getlist()* provided by this higher level interface.

`FieldStorage.`**`getfirst`**`(`*name*`[`, *default*`])`

This method always returns only one value associated with form field *name*. The method returns only the first value in case that more values were posted under such name. Please note that the order in which the values are received may vary from browser to browser and should not be counted on.[1] If no such form field or value exists then the method returns the value specified by the optional parameter *default*. This parameter defaults to `None` if not specified.

`FieldStorage.`**`getlist`**`(`*name*`)`

This method always returns a list of values associated with form field *name*. The method returns an empty list if no such form field or value exists for *name*. It returns a list consisting of one item if only one such value exists.

Using these methods you can write nice compact code:

```
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").upper()    # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```

20.2.4 Old classes

2.6 版后已移除: These classes, present in earlier versions of the *cgi* module, are still supported for backward compatibility. New applications should use the `FieldStorage` class.

`SvFormContentDict` stores single value form content as dictionary; it assumes each field name occurs in the form only once.

`FormContentDict` stores multiple value form content as a dictionary (the form items are lists of values). Useful if your form contains multiple fields with the same name.

Other classes (`FormContent`, `InterpFormContentDict`) are present for backwards compatibility with really old applications only.

[1] Note that some recent versions of the HTML specification do state what order the field values should be supplied in, but knowing whether a request was received from a conforming browser, or even from a browser at all, is tedious and error-prone.

20.2.5 函数

These are useful if you want more control, or if you want to employ some of the algorithms implemented in this module in other circumstances.

cgi.**parse**(*fp*[, *environ*[, *keep_blank_values*[, *strict_parsing*]]])
Parse a query in the environment or from a file (the file defaults to `sys.stdin` and environment defaults to `os.environ`). The *keep_blank_values* and *strict_parsing* parameters are passed to `urlparse.parse_qs()` unchanged.

cgi.**parse_qs**(*qs*[, *keep_blank_values*[, *strict_parsing*[, *max_num_fields*]]])
This function is deprecated in this module. Use `urlparse.parse_qs()` instead. It is maintained here only for backward compatibility.

cgi.**parse_qsl**(*qs*[, *keep_blank_values*[, *strict_parsing*[, *max_num_fields*]]])
This function is deprecated in this module. Use `urlparse.parse_qsl()` instead. It is maintained here only for backward compatibility.

cgi.**parse_multipart**(*fp*, *pdict*)
Parse input of type `multipart/form-data` (for file uploads). Arguments are *fp* for the input file and *pdict* for a dictionary containing other parameters in the `Content-Type` header.

Returns a dictionary just like `urlparse.parse_qs()` keys are the field names, each value is a list of values for that field. This is easy to use but not much good if you are expecting megabytes to be uploaded —in that case, use the `FieldStorage` class instead which is much more flexible.

Note that this does not parse nested multipart parts —use `FieldStorage` for that.

cgi.**parse_header**(*string*)
Parse a MIME header (such as `Content-Type`) into a main value and a dictionary of parameters.

cgi.**test**()
Robust test CGI script, usable as main program. Writes minimal HTTP headers and formats all information provided to the script in HTML form.

cgi.**print_environ**()
Format the shell environment in HTML.

cgi.**print_form**(*form*)
Format a form in HTML.

cgi.**print_directory**()
Format the current directory in HTML.

cgi.**print_environ_usage**()
Print a list of useful (used by CGI) environment variables in HTML.

cgi.**escape**(*s*[, *quote*])
Convert the characters `'&'`, `'<'` and `'>'` in string *s* to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. If the optional flag *quote* is true, the quotation mark character (`"`) is also translated; this helps for inclusion in an HTML attribute value delimited by double quotes, as in `<a href="...">`. Note that single quotes are never translated.

If the value to be quoted might include single- or double-quote characters, or both, consider using the `quoteattr()` function in the `xml.sax.saxutils` module instead.

20.2.6 Caring about security

There's one important rule: if you invoke an external program (via the *os.system()* or *os.popen()* functions. or others with similar functionality), make very sure you don't pass arbitrary strings received from the client to the shell. This is a well-known security hole whereby clever hackers anywhere on the Web can exploit a gullible CGI script to invoke arbitrary shell commands. Even parts of the URL or field names cannot be trusted, since the request doesn't have to come from your form!

To be on the safe side, if you must pass a string gotten from a form to a shell command, you should make sure the string contains only alphanumeric characters, dashes, underscores, and periods.

20.2.7 Installing your CGI script on a Unix system

Read the documentation for your HTTP server and check with your local system administrator to find the directory where CGI scripts should be installed; usually this is in a directory `cgi-bin` in the server tree.

Make sure that your script is readable and executable by "others"; the Unix file mode should be `0755` octal (use `chmod 0755 filename`). Make sure that the first line of the script contains `#!` starting in column 1 followed by the pathname of the Python interpreter, for instance:

```
#!/usr/local/bin/python
```

Make sure the Python interpreter exists and is executable by "others".

Make sure that any files your script needs to read or write are readable or writable, respectively, by "others"—their mode should be `0644` for readable and `0666` for writable. This is because, for security reasons, the HTTP server executes your script as user "nobody", without any special privileges. It can only read (write, execute) files that everybody can read (write, execute). The current directory at execution time is also different (it is usually the server's cgi-bin directory) and the set of environment variables is also different from what you get when you log in. In particular, don't count on the shell's search path for executables (`PATH`) or the Python module search path (`PYTHONPATH`) to be set to anything interesting.

If you need to load modules from a directory which is not on Python's default module search path, you can change the path in your script, before importing other modules. For example:

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

(This way, the directory inserted last will be searched first!)

Instructions for non-Unix systems will vary; check your HTTP server's documentation (it will usually have a section on CGI scripts).

20.2.8 Testing your CGI script

Unfortunately, a CGI script will generally not run when you try it from the command line, and a script that works perfectly from the command line may fail mysteriously when run from the server. There's one reason why you should still test your script from the command line: if it contains a syntax error, the Python interpreter won't execute it at all, and the HTTP server will most likely send a cryptic error to the client.

Assuming your script has no syntax errors, yet it does not work, you have no choice but to read the next section.

20.2.9 Debugging CGI scripts

First of all, check for trivial installation errors —reading the section above on installing your CGI script carefully can save you a lot of time. If you wonder whether you have understood the installation procedure correctly, try installing a copy of this module file (`cgi.py`) as a CGI script. When invoked as a script, the file will dump its environment and the contents of the form in HTML form. Give it the right mode etc, and send it a request. If it's installed in the standard `cgi-bin` directory, it should be possible to send it a request by entering a URL into your browser of the form:

```
http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home
```

If this gives an error of type 404, the server cannot find the script –perhaps you need to install it in a different directory. If it gives another error, there's an installation problem that you should fix before trying to go any further. If you get a nicely formatted listing of the environment and form content (in this example, the fields should be listed as "addr" with value "At Home" and "name" with value "Joe Blow"), the `cgi.py` script has been installed correctly. If you follow the same procedure for your own script, you should now be able to debug it.

The next step could be to call the *cgi* module's *test()* function from your script: replace its main code with the single statement

```
cgi.test()
```

This should produce the same results as those gotten from installing the `cgi.py` file itself.

When an ordinary Python script raises an unhandled exception (for whatever reason: of a typo in a module name, a file that can't be opened, etc.), the Python interpreter prints a nice traceback and exits. While the Python interpreter will still do this when your CGI script raises an exception, most likely the traceback will end up in one of the HTTP server's log files, or be discarded altogether.

Fortunately, once you have managed to get your script to execute *some* code, you can easily send tracebacks to the Web browser using the *cgitb* module. If you haven't done so already, just add the lines:

```
import cgitb
cgitb.enable()
```

to the top of your script. Then try running it again; when a problem occurs, you should see a detailed report that will likely make apparent the cause of the crash.

If you suspect that there may be a problem in importing the *cgitb* module, you can use an even more robust approach (which only uses built-in modules):

```
import sys
sys.stderr = sys.stdout
print "Content-Type: text/plain"
print
...your code here...
```

This relies on the Python interpreter to print the traceback. The content type of the output is set to plain text, which disables all HTML processing. If your script works, the raw HTML will be displayed by your client. If it raises an exception, most likely after the first two lines have been printed, a traceback will be displayed. Because no HTML interpretation is going on, the traceback will be readable.

20.2.10 Common problems and solutions

- Most HTTP servers buffer the output from CGI scripts until the script is completed. This means that it is not possible to display a progress report on the client's display while the script is running.
- Check the installation instructions above.
- Check the HTTP server's log files. (`tail -f logfile` in a separate window may be useful!)
- Always check a script for syntax errors first, by doing something like `python script.py`.
- If your script does not have any syntax errors, try adding `import cgitb; cgitb.enable()` to the top of the script.
- When invoking external programs, make sure they can be found. Usually, this means using absolute path names —`PATH` is usually not set to a very useful value in a CGI script.
- When reading or writing external files, make sure they can be read or written by the userid under which your CGI script will be running: this is typically the userid under which the web server is running, or some explicitly specified userid for a web server's `suexec` feature.
- Don't try to give a CGI script a set-uid mode. This doesn't work on most systems, and is a security liability as well.

备注

20.3 cgitb —用于 CGI 脚本的回溯管理器

2.2 新版功能.

cgitb 模块提供了用于 Python 脚本的特殊异常处理程序。(这个名称有一点误导性。它最初是设计用来显示 HTML 格式的 CGI 脚本详细回溯信息。但后来被一般化为也可显示纯文本格式的回溯信息。) 激活这个模块之后，如果发生了未被捕获的异常，将会显示详细的已格式化的报告。报告显示内容包括每个层级的源代码摘录，还有当前正在运行的函数的参数和局部变量值，以帮助你调试问题。你也可以选择将此信息保存至文件而不是将其发送至浏览器。

要启用此特性，只需简单地将此代码添加到你的 CGI 脚本的最顶端:

```
import cgitb
cgitb.enable()
```

enable() 函数的选项可以控制是将报告显示在浏览器中，还是将报告记录到文件以供随后进行分析。

cgitb.**enable**([*display*[, *logdir*[, *context*[, *format*]]]])
此函数可通过设置*sys.excepthook* 的值以使*cgitb* 模块接管解释器默认的异常处理机制。

可选参数 *display* 默认为 `1` 并可被设为 `0` 来停止将回溯发送至浏览器。如果给出了参数 *logdir*，则回溯会被写入文件。*logdir* 的值应当是一个用于存放所写入文件的目录。可选参数 *context* 是要在回溯中的当前源代码行前后显示的上下文行数；默认为 `5`。如果可选参数 *format* 为 `"html"`，输出将为 HTML 格式。任何其它值都会强制启用纯文本输出。默认取值为 `"html"`。

cgitb.**handler**([*info*])
此函数使用默认设置处理异常（即在浏览器中显示报告，但不记录到文件）。当你捕获了一个异常并希望使用*cgitb* 来报告它时可以使用此函数。可选的 *info* 参数应为一个包含异常类型，异常值和回溯对象的 3 元组，与*sys.exc_info()* 所返回的元组完全一致。如果未提供 *info* 参数，则会从*sys.exc_info()* 获取当前异常。

20.4 wsgiref —WSGI Utilities and Reference Implementation

2.5 新版功能.

The Web Server Gateway Interface (WSGI) is a standard interface between web server software and web applications written in Python. Having a standard interface makes it easy to use an application that supports WSGI with a number of different web servers.

Only authors of web servers and programming frameworks need to know every detail and corner case of the WSGI design. You don't need to understand every detail of WSGI just to install a WSGI application or to write a web application using an existing framework.

wsgiref is a reference implementation of the WSGI specification that can be used to add WSGI support to a web server or framework. It provides utilities for manipulating WSGI environment variables and response headers, base classes for implementing WSGI servers, a demo HTTP server that serves WSGI applications, and a validation tool that checks WSGI servers and applications for conformance to the WSGI specification (**PEP 333**).

See https://wsgi.readthedocs.org/ for more information about WSGI, and links to tutorials and other resources.

20.4.1 wsgiref.util –WSGI environment utilities

This module provides a variety of utility functions for working with WSGI environments. A WSGI environment is a dictionary containing HTTP request variables as described in **PEP 333**. All of the functions taking an *environ* parameter expect a WSGI-compliant dictionary to be supplied; please see **PEP 333** for a detailed specification.

wsgiref.util.**guess_scheme**(*environ*)
Return a guess for whether `wsgi.url_scheme` should be "http" or "https", by checking for a `HTTPS` environment variable in the *environ* dictionary. The return value is a string.

This function is useful when creating a gateway that wraps CGI or a CGI-like protocol such as FastCGI. Typically, servers providing such protocols will include a `HTTPS` variable with a value of "1" "yes", or "on" when a request is received via SSL. So, this function returns "https" if such a value is found, and "http" otherwise.

wsgiref.util.**request_uri**(*environ*, *include_query=1*)
Return the full request URI, optionally including the query string, using the algorithm found in the "URL Reconstruction" section of **PEP 333**. If *include_query* is false, the query string is not included in the resulting URI.

wsgiref.util.**application_uri**(*environ*)
Similar to *request_uri()*, except that the `PATH_INFO` and `QUERY_STRING` variables are ignored. The result is the base URI of the application object addressed by the request.

wsgiref.util.**shift_path_info**(*environ*)
Shift a single name from `PATH_INFO` to `SCRIPT_NAME` and return the name. The *environ* dictionary is *modified* in-place; use a copy if you need to keep the original `PATH_INFO` or `SCRIPT_NAME` intact.

If there are no remaining path segments in `PATH_INFO`, `None` is returned.

Typically, this routine is used to process each portion of a request URI path, for example to treat the path as a series of dictionary keys. This routine modifies the passed-in environment to make it suitable for invoking another WSGI application that is located at the target URI. For example, if there is a WSGI application at `/foo`, and the request URI path is `/foo/bar/baz`, and the WSGI application at `/foo` calls *shift_path_info()*, it will receive the string "bar", and the environment will be updated to be suitable for passing to a WSGI application at `/foo/bar`. That is, `SCRIPT_NAME` will change from `/foo` to `/foo/bar`, and `PATH_INFO` will change from `/bar/baz` to `/baz`.

When `PATH_INFO` is just a "/", this routine returns an empty string and appends a trailing slash to `SCRIPT_NAME`, even though empty path segments are normally ignored, and `SCRIPT_NAME` doesn't nor-

mally end in a slash. This is intentional behavior, to ensure that an application can tell the difference between URIs ending in `/x` from ones ending in `/x/` when using this routine to do object traversal.

`wsgiref.util.`**`setup_testing_defaults`**(*environ*)

Update *environ* with trivial defaults for testing purposes.

This routine adds various parameters required for WSGI, including `HTTP_HOST`, `SERVER_NAME`, `SERVER_PORT`, `REQUEST_METHOD`, `SCRIPT_NAME`, `PATH_INFO`, and all of the **PEP 333**-defined `wsgi.*` variables. It only supplies default values, and does not replace any existing settings for these variables.

This routine is intended to make it easier for unit tests of WSGI servers and applications to set up dummy environments. It should NOT be used by actual WSGI servers or applications, since the data is fake!

用法示例：

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It's going to print out the
# environment dictionary after being updated by setup_testing_defaults
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain')]

    start_response(status, headers)

    ret = ["%s: %s\n" % (key, value)
           for key, value in environ.iteritems()]
    return ret

httpd = make_server('', 8000, simple_app)
print "Serving on port 8000..."
httpd.serve_forever()
```

In addition to the environment functions above, the *wsgiref.util* module also provides these miscellaneous utilities:

`wsgiref.util.`**`is_hop_by_hop`**(*header_name*)

Return true if 'header_name' is an HTTP/1.1 "Hop-by-Hop" header, as defined by **RFC 2616**.

class `wsgiref.util.`**`FileWrapper`**(*filelike*, *blksize=8192*)

A wrapper to convert a file-like object to an *iterator*. The resulting objects support both `__getitem__()` and `__iter__()` iteration styles, for compatibility with Python 2.1 and Jython. As the object is iterated over, the optional *blksize* parameter will be repeatedly passed to the *filelike* object's `read()` method to obtain strings to yield. When `read()` returns an empty string, iteration is ended and is not resumable.

If *filelike* has a `close()` method, the returned object will also have a `close()` method, and it will invoke the *filelike* object's `close()` method when called.

用法示例：

```
from StringIO import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)
```

(下页继续)

(续上页)

```
for chunk in wrapper:
    print chunk
```

20.4.2 wsgiref.headers –WSGI response header tools

This module provides a single class, *Headers*, for convenient manipulation of WSGI response headers using a mapping-like interface.

class wsgiref.headers.**Headers**(*headers*)

Create a mapping-like object wrapping *headers*, which must be a list of header name/value tuples as described in **PEP 333**. Any changes made to the new *Headers* object will directly update the *headers* list it was created with.

Headers objects support typical mapping operations including `__getitem__()`, `get()`, `__setitem__()`, `setdefault()`, `__delitem__()`, `__contains__()` and `has_key()`. For each of these methods, the key is the header name (treated case-insensitively), and the value is the first value associated with that header name. Setting a header deletes any existing values for that header, then adds a new value at the end of the wrapped header list. Headers' existing order is generally maintained, with new headers added to the end of the wrapped list.

Unlike a dictionary, *Headers* objects do not raise an error when you try to get or delete a key that isn't in the wrapped header list. Getting a nonexistent header just returns `None`, and deleting a nonexistent header does nothing.

Headers objects also support `keys()`, `values()`, and `items()` methods. The lists returned by `keys()` and `items()` can include the same key more than once if there is a multi-valued header. The `len()` of a *Headers* object is the same as the length of its `items()`, which is the same as the length of the wrapped header list. In fact, the `items()` method just returns a copy of the wrapped header list.

Calling `str()` on a *Headers* object returns a formatted string suitable for transmission as HTTP response headers. Each header is placed on a line with its value, separated by a colon and a space. Each line is terminated by a carriage return and line feed, and the string is terminated with a blank line.

In addition to their mapping interface and formatting features, *Headers* objects also have the following methods for querying and adding multi-valued headers, and for adding headers with MIME parameters:

get_all(*name*)

Return a list of all the values for the named header.

The returned list will be sorted in the order they appeared in the original header list or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list. If no fields exist with the given name, returns an empty list.

add_header(*name*, *value*, ***_params*)

Add a (possibly multi-valued) header, with optional MIME parameters specified via keyword arguments.

name is the header field to add. Keyword arguments can be used to set MIME parameters for the header field. Each parameter must be a string or `None`. Underscores in parameter names are converted to dashes, since dashes are illegal in Python identifiers, but many MIME parameter names include dashes. If the parameter value is a string, it is added to the header value parameters in the form `name="value"`. If it is `None`, only the parameter name is added. (This is used for MIME parameters without a value.) Example usage:

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

The above will add a header that looks like this:

```
Content-Disposition: attachment; filename="bud.gif"
```

20.4.3 wsgiref.simple_server –a simple WSGI HTTP server

This module implements a simple HTTP server (based on *BaseHTTPServer*) that serves WSGI applications. Each server instance serves a single WSGI application on a given host and port. If you want to serve multiple applications on a single host and port, you should create a WSGI application that parses PATH_INFO to select which application to invoke for each request. (E.g., using the shift_path_info() function from *wsgiref.util*.)

wsgiref.simple_server.**make_server**(*host*, *port*, *app*, *server_class=WSGIServer*, *handler_class=WSGIRequestHandler*)

Create a new WSGI server listening on *host* and *port*, accepting connections for *app*. The return value is an instance of the supplied *server_class*, and will process requests using the specified *handler_class*. *app* must be a WSGI application object, as defined by **PEP 333**.

用法示例:

```
from wsgiref.simple_server import make_server, demo_app

httpd = make_server('', 8000, demo_app)
print "Serving HTTP on port 8000..."

# Respond to requests until process is killed
httpd.serve_forever()

# Alternative: serve one request, then exit
httpd.handle_request()
```

wsgiref.simple_server.**demo_app**(*environ*, *start_response*)

This function is a small but complete WSGI application that returns a text page containing the message "Hello world!" and a list of the key/value pairs provided in the *environ* parameter. It's useful for verifying that a WSGI server (such as *wsgiref.simple_server*) is able to run a simple WSGI application correctly.

class wsgiref.simple_server.**WSGIServer**(*server_address*, *RequestHandlerClass*)

Create a *WSGIServer* instance. *server_address* should be a (host,port) tuple, and *RequestHandlerClass* should be the subclass of *BaseHTTPServer.BaseHTTPRequestHandler* that will be used to process requests.

You do not normally need to call this constructor, as the *make_server()* function can handle all the details for you.

WSGIServer is a subclass of *BaseHTTPServer.HTTPServer*, so all of its methods (such as serve_forever() and handle_request()) are available. *WSGIServer* also provides these WSGI-specific methods:

set_app(*application*)

Sets the callable *application* as the WSGI application that will receive requests.

get_app()

Returns the currently-set application callable.

Normally, however, you do not need to use these additional methods, as *set_app()* is normally called by *make_server()*, and the *get_app()* exists mainly for the benefit of request handler instances.

class wsgiref.simple_server.**WSGIRequestHandler**(*request*, *client_address*, *server*)

Create an HTTP handler for the given *request* (i.e. a socket), *client_address* (a (host,port) tuple), and *server* (*WSGIServer* instance).

You do not need to create instances of this class directly; they are automatically created as needed by *WSGIServer* objects. You can, however, subclass this class and supply it as a *handler_class* to the *make_server()* function. Some possibly relevant methods for overriding in subclasses:

get_environ()
Returns a dictionary containing the WSGI environment for a request. The default implementation copies the contents of the *WSGIServer* object's `base_environ` dictionary attribute and then adds various headers derived from the HTTP request. Each call to this method should return a new dictionary containing all of the relevant CGI environment variables as specified in **PEP 333**.

get_stderr()
Return the object that should be used as the `wsgi.errors` stream. The default implementation just returns `sys.stderr`.

handle()
Process the HTTP request. The default implementation creates a handler instance using a *wsgiref.handlers* class to implement the actual WSGI application interface.

20.4.4 `wsgiref.validate` —WSGI conformance checker

When creating new WSGI application objects, frameworks, servers, or middleware, it can be useful to validate the new code's conformance using *wsgiref.validate*. This module provides a function that creates WSGI application objects that validate communications between a WSGI server or gateway and a WSGI application object, to check both sides for protocol conformance.

Note that this utility does not guarantee complete **PEP 333** compliance; an absence of errors from this module does not necessarily mean that errors do not exist. However, if this module does produce an error, then it is virtually certain that either the server or application is not 100% compliant.

This module is based on the `paste.lint` module from Ian Bicking's "Python Paste" library.

`wsgiref.validate.`**validator**(*application*)
Wrap *application* and return a new WSGI application object. The returned application will forward all requests to the original *application*, and will check that both the *application* and the server invoking it are conforming to the WSGI specification and to RFC 2616.

Any detected nonconformance results in an *AssertionError* being raised; note, however, that how these errors are handled is server-dependent. For example, *wsgiref.simple_server* and other servers based on *wsgiref.handlers* (that don't override the error handling methods to do something else) will simply output a message that an error has occurred, and dump the traceback to `sys.stderr` or some other error stream.

This wrapper may also generate output using the *warnings* module to indicate behaviors that are questionable but which may not actually be prohibited by **PEP 333**. Unless they are suppressed using Python command-line options or the *warnings* API, any such warnings will be written to `sys.stderr` (*not* `wsgi.errors`, unless they happen to be the same object).

用法示例:

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant to the
# standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    # This is going to break because we need to return a list, and
    # the validator is going to inform us
    return "Hello World"
```

(下页继续)

(续上页)

```
# This is the application wrapped in a validator
validator_app = validator(simple_app)

httpd = make_server('', 8000, validator_app)
print "Listening on port 8000...."
httpd.serve_forever()
```

20.4.5 wsgiref.handlers –server/gateway base classes

This module provides base handler classes for implementing WSGI servers and gateways. These base classes handle most of the work of communicating with a WSGI application, as long as they are given a CGI-like environment, along with input, output, and error streams.

class wsgiref.handlers.**CGIHandler**

CGI-based invocation via sys.stdin, sys.stdout, sys.stderr and os.environ. This is useful when you have a WSGI application and want to run it as a CGI script. Simply invoke CGIHandler().run(app), where app is the WSGI application object you wish to invoke.

This class is a subclass of *BaseCGIHandler* that sets wsgi.run_once to true, wsgi.multithread to false, and wsgi.multiprocess to true, and always uses *sys* and *os* to obtain the necessary CGI streams and environment.

class wsgiref.handlers.**BaseCGIHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Similar to *CGIHandler*, but instead of using the *sys* and *os* modules, the CGI environment and I/O streams are specified explicitly. The *multithread* and *multiprocess* values are used to set the wsgi.multithread and wsgi.multiprocess flags for any applications run by the handler instance.

This class is a subclass of *SimpleHandler* intended for use with software other than HTTP "origin servers". If you are writing a gateway protocol implementation (such as CGI, FastCGI, SCGI, etc.) that uses a Status: header to send an HTTP status, you probably want to subclass this instead of *SimpleHandler*.

class wsgiref.handlers.**SimpleHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Similar to *BaseCGIHandler*, but designed for use with HTTP origin servers. If you are writing an HTTP server implementation, you will probably want to subclass this instead of *BaseCGIHandler*.

This class is a subclass of *BaseHandler*. It overrides the __init__(), get_stdin(), get_stderr(), add_cgi_vars(), _write(), and _flush() methods to support explicitly setting the environment and streams via the constructor. The supplied environment and streams are stored in the stdin, stdout, stderr, and environ attributes.

class wsgiref.handlers.**BaseHandler**

This is an abstract base class for running WSGI applications. Each instance will handle a single HTTP request, although in principle you could create a subclass that was reusable for multiple requests.

BaseHandler instances have only one method intended for external use:

run(*app*)

Run the specified WSGI application, *app*.

All of the other *BaseHandler* methods are invoked by this method in the process of running the application, and thus exist primarily to allow customizing the process.

The following methods MUST be overridden in a subclass:

_write(*data*)
Buffer the string *data* for transmission to the client. It's okay if this method actually transmits the data; *BaseHandler* just separates write and flush operations for greater efficiency when the underlying system actually has such a distinction.

_flush()
Force buffered data to be transmitted to the client. It's okay if this method is a no-op (i.e., if *_write()* actually sends the data).

get_stdin()
Return an input stream object suitable for use as the `wsgi.input` of the request currently being processed.

get_stderr()
Return an output stream object suitable for use as the `wsgi.errors` of the request currently being processed.

add_cgi_vars()
Insert CGI variables for the current request into the `environ` attribute.

Here are some other methods and attributes you may wish to override. This list is only a summary, however, and does not include every method that can be overridden. You should consult the docstrings and source code for additional information before attempting to create a customized *BaseHandler* subclass.

Attributes and methods for customizing the WSGI environment:

wsgi_multithread
The value to be used for the `wsgi.multithread` environment variable. It defaults to true in *BaseHandler*, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_multiprocess
The value to be used for the `wsgi.multiprocess` environment variable. It defaults to true in *BaseHandler*, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_run_once
The value to be used for the `wsgi.run_once` environment variable. It defaults to false in *BaseHandler*, but *CGIHandler* sets it to true by default.

os_environ
The default environment variables to be included in every request's WSGI environment. By default, this is a copy of `os.environ` at the time that *wsgiref.handlers* was imported, but subclasses can either create their own at the class or instance level. Note that the dictionary should be considered read-only, since the default value is shared between multiple classes and instances.

server_software
If the *origin_server* attribute is set, this attribute's value is used to set the default `SERVER_SOFTWARE` WSGI environment variable, and also to set a default `Server:` header in HTTP responses. It is ignored for handlers (such as *BaseCGIHandler* and *CGIHandler*) that are not HTTP origin servers.

get_scheme()
Return the URL scheme being used for the current request. The default implementation uses the `guess_scheme()` function from *wsgiref.util* to guess whether the scheme should be "http" or "https", based on the current request's `environ` variables.

setup_environ()
Set the `environ` attribute to a fully-populated WSGI environment. The default implementation uses all of the above methods and attributes, plus the *get_stdin()*, *get_stderr()*, and *add_cgi_vars()* methods and the *wsgi_file_wrapper* attribute. It also inserts a `SERVER_SOFTWARE` key if not present, as long as the *origin_server* attribute is a true value and the *server_software* attribute is set.

Methods and attributes for customizing exception handling:

log_exception(*exc_info*)
Log the *exc_info* tuple in the server log. *exc_info* is a `(type, value, traceback)` tuple. The default implementation simply writes the traceback to the request's `wsgi.errors` stream and flushes it. Subclasses can override this method to change the format or retarget the output, mail the traceback to an administrator, or whatever other action may be deemed suitable.

traceback_limit
The maximum number of frames to include in tracebacks output by the default *log_exception()* method. If `None`, all frames are included.

error_output(*environ*, *start_response*)
This method is a WSGI application to generate an error page for the user. It is only invoked if an error occurs before headers are sent to the client.

This method can access the current error information using `sys.exc_info()`, and should pass that information to *start_response* when calling it (as described in the "Error Handling" section of **PEP 333**).

The default implementation just uses the *error_status*, *error_headers*, and *error_body* attributes to generate an output page. Subclasses can override this to produce more dynamic error output.

Note, however, that it's not recommended from a security perspective to spit out diagnostics to any old user; ideally, you should have to do something special to enable diagnostic output, which is why the default implementation doesn't include any.

error_status
The HTTP status used for error responses. This should be a status string as defined in **PEP 333**; it defaults to a 500 code and message.

error_headers
The HTTP headers used for error responses. This should be a list of WSGI response headers (`(name, value)` tuples), as described in **PEP 333**. The default list just sets the content type to `text/plain`.

error_body
The error response body. This should be an HTTP response body string. It defaults to the plain text, "A server error occurred. Please contact the administrator."

Methods and attributes for **PEP 333**'s "Optional Platform-Specific File Handling" feature:

wsgi_file_wrapper
A `wsgi.file_wrapper` factory, or `None`. The default value of this attribute is the `FileWrapper` class from *wsgiref.util*.

sendfile()
Override to implement platform-specific file transmission. This method is called only if the application's return value is an instance of the class specified by the *wsgi_file_wrapper* attribute. It should return a true value if it was able to successfully transmit the file, so that the default transmission code will not be executed. The default implementation of this method just returns a false value.

Miscellaneous methods and attributes:

origin_server
This attribute should be set to a true value if the handler's *_write()* and *_flush()* are being used to communicate directly to the client, rather than via a CGI-like gateway protocol that wants the HTTP status in a special `Status:` header.

This attribute's default value is true in *BaseHandler*, but false in *BaseCGIHandler* and *CGIHandler*.

http_version
If *origin_server* is true, this string attribute is used to set the HTTP version of the response set to the client. It defaults to `"1.0"`.

20.4.6 例子

This is a working "Hello World" WSGI application:

```
from wsgiref.simple_server import make_server

# Every WSGI application must have an application object - a callable
# object that accepts two arguments. For that purpose, we're going to
# use a function (note that you're not limited to a function, you can
# use a class for example). The first argument passed to the function
# is a dictionary containing CGI-style environment variables and the
# second variable is the callable object (see PEP 333).
def hello_world_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return ["Hello World"]

httpd = make_server('', 8000, hello_world_app)
print "Serving on port 8000..."

# Serve until process is killed
httpd.serve_forever()
```

20.5 `urllib` —Open arbitrary resources by URL

注解: The *urllib* module has been split into parts and renamed in Python 3 to `urllib.request`, `urllib.parse`, and `urllib.error`. The *2to3* tool will automatically adapt imports when converting your sources to Python 3. Also note that the `urllib.request.urlopen()` function in Python 3 is equivalent to *urllib2.urlopen()* and that *urllib.urlopen()* has been removed.

This module provides a high-level interface for fetching data across the World Wide Web. In particular, the *urlopen()* function is similar to the built-in function *open()*, but accepts Universal Resource Locators (URLs) instead of filenames. Some restrictions apply —it can only open URLs for reading, and no seek operations are available.

参见:

The Requests package is recommended for a higher-level HTTP client interface.

在 2.7.9 版更改: For HTTPS URIs, *urllib* performs all the neccessary certificate and hostname checks by default.

警告: For Python versions earlier than 2.7.9, urllib does not attempt to validate the server certificates of HTTPS URIs. Use at your own risk!

20.5.1 High-level interface

urllib.**urlopen**(*url*[, *data*[, *proxies*[, *context*]]])

Open a network object denoted by a URL for reading. If the URL does not have a scheme identifier, or if it has file: as its scheme identifier, this opens a local file (without *universal newlines*); otherwise it opens a socket to a server somewhere on the network. If the connection cannot be made the *IOError* exception is raised. If all went well, a file-like object is returned. This supports the following methods: read(), *readline()*, readlines(), fileno(), close(), info(), getcode() and geturl(). It also has proper support for the *iterator* protocol. One caveat: the read() method, if the size argument is omitted or negative, may not read until the end of the data stream; there is no good way to determine that the entire stream from a socket has been read in the general case.

Except for the info(), getcode() and geturl() methods, these methods have the same interface as for file objects —see section *File Objects* in this manual. (It is not a built-in file object, however, so it can't be used at those few places where a true built-in file object is required.)

The info() method returns an instance of the class *mimetools.Message* containing meta-information associated with the URL. When the method is HTTP, these headers are those returned by the server at the head of the retrieved HTML page (including Content-Length and Content-Type). When the method is FTP, a Content-Length header will be present if (as is now usual) the server passed back a file length in response to the FTP retrieval request. A Content-Type header will be present if the MIME type can be guessed. When the method is local-file, returned headers will include a Date representing the file's last-modified time, a Content-Length giving file size, and a Content-Type containing a guess at the file's type. See also the description of the *mimetools* module.

The geturl() method returns the real URL of the page. In some cases, the HTTP server redirects a client to another URL. The *urlopen()* function handles this transparently, but in some cases the caller needs to know which URL the client was redirected to. The geturl() method can be used to get at this redirected URL.

The getcode() method returns the HTTP status code that was sent with the response, or None if the URL is no HTTP URL.

If the *url* uses the http: scheme identifier, the optional *data* argument may be given to specify a POST request (normally the request type is GET). The *data* argument must be in standard *application/x-www-form-urlencoded* format; see the *urlencode()* function below.

The *urlopen()* function works transparently with proxies which do not require authentication. In a Unix or Windows environment, set the http_proxy, or ftp_proxy environment variables to a URL that identifies the proxy server before starting the Python interpreter. For example (the '%' is the command prompt):

```
% http_proxy="http://www.someproxy.com:3128"
% export http_proxy
% python
...
```

The no_proxy environment variable can be used to specify hosts which shouldn't be reached via proxy; if set, it should be a comma-separated list of hostname suffixes, optionally with :port appended, for example cern.ch,ncsa.uiuc.edu,some.host:8080.

In a Windows environment, if no proxy environment variables are set, proxy settings are obtained from the registry's Internet Settings section.

In a Mac OS X environment, *urlopen()* will retrieve proxy information from the OS X System Configuration Framework, which can be managed with Network System Preferences panel.

Alternatively, the optional *proxies* argument may be used to explicitly specify proxies. It must be a dictionary mapping scheme names to proxy URLs, where an empty dictionary causes no proxies to be used, and None (the default value) causes environmental proxy settings to be used as discussed above. For example:

```
# Use http://www.someproxy.com:3128 for HTTP proxying
proxies = {'http': 'http://www.someproxy.com:3128'}
filehandle = urllib.urlopen(some_url, proxies=proxies)
# Don't use any proxies
filehandle = urllib.urlopen(some_url, proxies={})
# Use proxies from environment - both versions are equivalent
filehandle = urllib.urlopen(some_url, proxies=None)
filehandle = urllib.urlopen(some_url)
```

Proxies which require authentication for use are not currently supported; this is considered an implementation limitation.

The *context* parameter may be set to a *ssl.SSLContext* instance to configure the SSL settings that are used if *urlopen()* makes a HTTPS connection.

在 2.3 版更改: Added the *proxies* support.

在 2.6 版更改: Added `getcode()` to returned object and support for the `no_proxy` environment variable.

在 2.7.9 版更改: The *context* parameter was added. All the neccessary certificate and hostname checks are done by default.

2.6 版后已移除: The *urlopen()* function has been removed in Python 3 in favor of *urllib2.urlopen()*.

`urllib.`**`urlretrieve`**(*url*[, *filename*[, *reporthook*[, *data*[, *context*]]]])

Copy a network object denoted by a URL to a local file, if necessary. If the URL points to a local file, or a valid cached copy of the object exists, the object is not copied. Return a tuple `(filename, headers)` where *filename* is the local file name under which the object can be found, and *headers* is whatever the `info()` method of the object returned by *urlopen()* returned (for a remote object, possibly cached). Exceptions are the same as for *urlopen()*.

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a callable that will be called once on establishment of the network connection and once after each block read thereafter. The callable will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a `POST` request (normally the request type is `GET`). The *data* argument must in standard *application/x-www-form-urlencoded* format; see the *urlencode()* function below.

The *context* parameter may be set to a *ssl.SSLContext* instance to configure the SSL settings that are used if *urlretrieve()* makes a HTTPS connection.

在 2.5 版更改: *urlretrieve()* will raise *ContentTooShortError* when it detects that the amount of data available was less than the expected amount (which is the size reported by a *Content-Length* header). This can occur, for example, when the download is interrupted.

The *Content-Length* is treated as a lower bound: if there's more data to read, *urlretrieve()* reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the `content` attribute of the exception instance.

If no *Content-Length* header was supplied, *urlretrieve()* can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

在 2.7.9 版更改: The *context* parameter was added. All the neccessary certificate and hostname checks are done by default.

urllib.**_urlopener**
The public functions *urlopen()* and *urlretrieve()* create an instance of the *FancyURLopener* class and use it to perform their requested actions. To override this functionality, programmers can create a subclass of *URLopener* or *FancyURLopener*, then assign an instance of that class to the `urllib._urlopener` variable before calling the desired function. For example, applications may want to specify a different *User-Agent* header than *URLopener* defines. This can be accomplished with the following code:

```
import urllib

class AppURLopener(urllib.FancyURLopener):
    version = "App/1.7"

urllib._urlopener = AppURLopener()
```

urllib.**urlcleanup**()
Clear the cache that may have been built up by previous calls to *urlretrieve()*.

20.5.2 Utility functions

urllib.**quote**(*string*[, *safe*])
Replace special characters in *string* using the `%xx` escape. Letters, digits, and the characters `'_.-'` are never quoted. By default, this function is intended for quoting the path section of the URL. The optional *safe* parameter specifies additional characters that should not be quoted —its default value is `'/'`.

Example: `quote('/~connolly/')` yields `'/%7econnolly/'`.

urllib.**quote_plus**(*string*[, *safe*])
Like *quote()*, but also replaces spaces by plus signs, as required for quoting HTML form values when building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in *safe*. It also does not have *safe* default to `'/'`.

urllib.**unquote**(*string*)
Replace `%xx` escapes by their single-character equivalent.

Example: `unquote('/%7Econnolly/')` yields `'/~connolly/'`.

urllib.**unquote_plus**(*string*)
Like *unquote()*, but also replaces plus signs by spaces, as required for unquoting HTML form values.

urllib.**urlencode**(*query*[, *doseq*])
Convert a mapping object or a sequence of two-element tuples to a "percent-encoded" string, suitable to pass to *urlopen()* above as the optional *data* argument. This is useful to pass a dictionary of form fields to a `POST` request. The resulting string is a series of `key=value` pairs separated by `'&'` characters, where both *key* and *value* are quoted using *quote_plus()* above. When a sequence of two-element tuples is used as the *query* argument, the first element of each tuple is a key and the second is a value. The value element in itself can be a sequence and in that case, if the optional parameter *doseq* is evaluates to `True`, individual `key=value` pairs separated by `'&'` are generated for each element of the value sequence for the key. The order of parameters in the encoded string will match the order of parameter tuples in the sequence. The *urlparse* module provides the functions `parse_qs()` and `parse_qsl()` which are used to parse query strings into Python data structures.

urllib.**pathname2url**(*path*)
Convert the pathname *path* from the local syntax for a path to the form used in the path component of a URL. This does not produce a complete URL. The return value will already be quoted using the *quote()* function.

urllib.**url2pathname**(*path*)
Convert the path component *path* from a percent-encoded URL to the local syntax for a path. This does not accept a complete URL. This function uses *unquote()* to decode *path*.

urllib.**getproxies**()
This helper function returns a dictionary of scheme to proxy server URL mappings. It scans the environment for variables named <scheme>_proxy, in case insensitive way, for all operating systems first, and when it cannot find it, looks for proxy information from Mac OSX System Configuration for Mac OS X and Windows Systems Registry for Windows. If both lowercase and uppercase environment variables exist (and disagree), lowercase is preferred.

注解: If the environment variable REQUEST_METHOD is set, which usually indicates your script is running in a CGI environment, the environment variable HTTP_PROXY (uppercase _PROXY) will be ignored. This is because that variable can be injected by a client using the "Proxy:" HTTP header. If you need to use an HTTP proxy in a CGI environment, either use ProxyHandler explicitly, or make sure the variable name is in lowercase (or at least the _proxy suffix).

注解: urllib also exposes certain utility functions like splittype, splithost and others parsing URL into various components. But it is recommended to use *urlparse* for parsing URLs rather than using these functions directly. Python 3 does not expose these helper functions from urllib.parse module.

20.5.3 URL Opener objects

class urllib.**URLopener**([*proxies*[, *context*[, ***x509*]]])
Base class for opening and reading URLs. Unless you need to support opening objects using schemes other than http:, ftp:, or file:, you probably want to use *FancyURLopener*.

By default, the *URLopener* class sends a *User-Agent* header of urllib/VVV, where *VVV* is the *urllib* version number. Applications can define their own *User-Agent* header by subclassing *URLopener* or *FancyURLopener* and setting the class attribute *version* to an appropriate string value in the subclass definition.

The optional *proxies* parameter should be a dictionary mapping scheme names to proxy URLs, where an empty dictionary turns proxies off completely. Its default value is None, in which case environmental proxy settings will be used if present, as discussed in the definition of *urlopen()*, above.

The *context* parameter may be a *ssl.SSLContext* instance. If given, it defines the SSL settings the opener uses to make HTTPS connections.

Additional keyword parameters, collected in *x509*, may be used for authentication of the client when using the https: scheme. The keywords *key_file* and *cert_file* are supported to provide an SSL key and certificate; both are needed to support client authentication.

URLopener objects will raise an *IOError* exception if the server returns an error code.

在 2.7.9 版更改: The *context* parameter was added. All the neccessary certificate and hostname checks are done by default.

open(*fullurl*[, *data*])
Open *fullurl* using the appropriate protocol. This method sets up cache and proxy information, then calls the appropriate open method with its input arguments. If the scheme is not recognized, *open_unknown()* is called. The *data* argument has the same meaning as the *data* argument of *urlopen()*.

open_unknown(*fullurl*[, *data*])
Overridable interface to open unknown URL types.

retrieve(*url*[, *filename*[, *reporthook*[, *data*]]])
Retrieves the contents of *url* and places it in *filename*. The return value is a tuple consisting of a local filename

and either a *mimetools.Message* object containing the response headers (for remote URLs) or None (for local URLs). The caller must then open and read the contents of *filename*. If *filename* is not given and the URL refers to a local file, the input filename is returned. If the URL is non-local and *filename* is not given, the filename is the output of *tempfile.mktemp()* with a suffix that matches the suffix of the last path component of the input URL. If *reporthook* is given, it must be a function accepting three numeric parameters. It will be called after each chunk of data is read from the network. *reporthook* is ignored for local URLs.

If the *url* uses the http: scheme identifier, the optional *data* argument may be given to specify a POST request (normally the request type is GET). The *data* argument must in standard *application/x-www-form-urlencoded* format; see the *urlencode()* function below.

version
Variable that specifies the user agent of the opener object. To get *urllib* to tell servers that it is a particular user agent, set this in a subclass as a class variable or in the constructor before calling the base constructor.

class urllib.**FancyURLopener**(...)
FancyURLopener subclasses *URLopener* providing default handling for the following HTTP response codes: 301, 302, 303, 307 and 401. For the 30x response codes listed above, the *Location* header is used to fetch the actual URL. For 401 response codes (authentication required), basic HTTP authentication is performed. For the 30x response codes, recursion is bounded by the value of the *maxtries* attribute, which defaults to 10.

For all other response codes, the method http_error_default() is called which you can override in subclasses to handle the error appropriately.

注解: According to the letter of **RFC 2616**, 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and *urllib* reproduces this behaviour.

The parameters to the constructor are the same as those for *URLopener*.

注解: When performing basic authentication, a *FancyURLopener* instance calls its *prompt_user_passwd()* method. The default implementation asks the users for the required information on the controlling terminal. A subclass may override this method to support more appropriate behavior if needed.

The *FancyURLopener* class offers one additional method that should be overloaded to provide the appropriate behavior:

prompt_user_passwd(*host*, *realm*)
Return information needed to authenticate the user at the given host in the specified security realm. The return value should be a tuple, (user, password), which can be used for basic authentication.

The implementation prompts for this information on the terminal; an application should override this method to use an appropriate interaction model in the local environment.

exception urllib.**ContentTooShortError**(*msg*[, *content*])
This exception is raised when the *urlretrieve()* function detects that the amount of the downloaded data is less than the expected amount (given by the *Content-Length* header). The content attribute stores the downloaded (and supposedly truncated) data.

2.5 新版功能.

20.5.4 urllib Restrictions

- Currently, only the following protocols are supported: HTTP, (versions 0.9 and 1.0), FTP, and local files.
- The caching feature of *urlretrieve()* has been disabled until I find the time to hack proper processing of Expiration time headers.
- There should be a function to query whether a particular URL is in the cache.
- For backward compatibility, if a URL appears to point to a local file but the file can't be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.
- The *urlopen()* and *urlretrieve()* functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive Web client using these functions without using threads.
- The data returned by *urlopen()* or *urlretrieve()* is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the *Content-Type* header. If the returned data is HTML, you can use the module *htmllib* to parse it.
- The code handling the FTP protocol cannot differentiate between a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL ends in a `/`, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing `/` has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible; the FTP code will try to read it, fail with a 550 error, and then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the *ftplib* module, subclassing *FancyURLopener*, or changing *_urlopener* to meet your needs.
- This module does not support the use of proxies which require authentication. This may be implemented in the future.
- Although the *urllib* module contains (undocumented) routines to parse and unparse URL strings, the recommended interface for URL manipulation is in module *urlparse*.

20.5.5 Examples

Here is an example session that uses the `GET` method to retrieve a URL containing parameters:

```
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.musi-cal.com/cgi-bin/query?%s" % params)
>>> print f.read()
```

The following example uses the `POST` method instead:

```
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.musi-cal.com/cgi-bin/query", params)
>>> print f.read()
```

The following example uses an explicitly specified HTTP proxy, overriding environment settings:

```
>>> import urllib
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.FancyURLopener(proxies)
>>> f = opener.open("http://www.python.org")
>>> f.read()
```

The following example uses no proxies at all, overriding environment settings:

```
>>> import urllib
>>> opener = urllib.FancyURLopener({})
>>> f = opener.open("http://www.python.org/")
>>> f.read()
```

20.6 urllib2 —extensible library for opening URLs

注解: The *urllib2* module has been split across several modules in Python 3 named `urllib.request` and `urllib.error`. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

The *urllib2* module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.

参见:

The Requests package is recommended for a higher-level HTTP client interface.

The *urllib2* module defines the following functions:

urllib2.**urlopen**(*url[, data[, timeout[, cafile[, capath[, cadefault[, context]]]]]]*)

Open the URL *url*, which can be either a string or a *Request* object.

data may be a string specifying additional data to send to the server, or `None` if no such data is needed. Currently HTTP requests are the only ones that use *data*; the HTTP request will be a POST instead of a GET when the *data* parameter is provided. *data* should be a buffer in the standard *application/x-www-form-urlencoded* format. The *urllib.urlencode()* function takes a mapping or sequence of 2-tuples and returns a string in this format. urllib2 module sends HTTP/1.1 requests with `Connection:close` header included.

The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). This actually only works for HTTP, HTTPS and FTP connections.

If *context* is specified, it must be a *ssl.SSLContext* instance describing the various SSL options. See *HTTPSConnection* for more details.

The optional *cafile* and *capath* parameters specify a set of trusted CA certificates for HTTPS requests. *cafile* should point to a single file containing a bundle of CA certificates, whereas *capath* should point to a directory of hashed certificate files. More information can be found in *ssl.SSLContext.load_verify_locations()*.

The *cadefault* parameter is ignored.

This function returns a file-like object with three additional methods:

- `geturl()` —return the URL of the resource retrieved, commonly used to determine if a redirect was followed
- `info()` —return the meta-information of the page, such as headers, in the form of an *mimetools.Message* instance (see Quick Reference to HTTP Headers)

- `getcode()` —return the HTTP status code of the response.

Raises *URLError* on errors.

Note that `None` may be returned if no handler handles the request (though the default installed global *OpenerDirector* uses *UnknownHandler* to ensure this never happens).

In addition, if proxy settings are detected (for example, when a `*_proxy` environment variable like `http_proxy` is set), *ProxyHandler* is default installed and makes sure the requests are handled through the proxy.

在 2.6 版更改: *timeout* was added.

在 2.7.9 版更改: *cafile*, *capath*, *cadefault*, and *context* were added.

`urllib2.`**`install_opener`**(*opener*)

Install an *OpenerDirector* instance as the default global opener. Installing an opener is only necessary if you want urlopen to use that opener; otherwise, simply call *OpenerDirector.open()* instead of *urlopen()*. The code does not check for a real *OpenerDirector*, and any class with the appropriate interface will work.

`urllib2.`**`build_opener`**(*[handler, ...]*)

Return an *OpenerDirector* instance, which chains the handlers in the order given. *handler*s can be either instances of *BaseHandler*, or subclasses of *BaseHandler* (in which case it must be possible to call the constructor without any parameters). Instances of the following classes will be in front of the *handler*s, unless the *handler*s contain them, instances of them or subclasses of them: *ProxyHandler* (if proxy settings are detected), *UnknownHandler*, *HTTPHandler*, *HTTPDefaultErrorHandler*, *HTTPRedirectHandler*, *FTPHandler*, *FileHandler*, *HTTPErrorProcessor*.

If the Python installation has SSL support (i.e., if the *ssl* module can be imported), *HTTPSHandler* will also be added.

Beginning in Python 2.3, a *BaseHandler* subclass may also change its `handler_order` attribute to modify its position in the handlers list.

The following exceptions are raised as appropriate:

exception `urllib2.`**`URLError`**

The handlers raise this exception (or derived exceptions) when they run into a problem. It is a subclass of *IOError*.

`reason`

The reason for this error. It can be a message string or another exception instance (*socket.error* for remote URLs, *OSError* for local URLs).

exception `urllib2.`**`HTTPError`**

Though being an exception (a subclass of *URLError*), an *HTTPError* can also function as a non-exceptional file-like return value (the same thing that *urlopen()* returns). This is useful when handling exotic HTTP errors, such as requests for authentication.

`code`

An HTTP status code as defined in RFC 2616. This numeric value corresponds to a value found in the dictionary of codes as found in *BaseHTTPServer.BaseHTTPRequestHandler.responses*.

`reason`

The reason for this error. It can be a message string or another exception instance.

The following classes are provided:

class `urllib2.`**`Request`**(*url[, data][, headers][, origin_req_host][, unverifiable]*)

This class is an abstraction of a URL request.

url should be a string containing a valid URL.

data may be a string specifying additional data to send to the server, or `None` if no such data is needed. Currently HTTP requests are the only ones that use *data*; the HTTP request will be a POST instead of a GET when the *data* parameter is provided. *data* should be a buffer in the standard *application/x-www-form-urlencoded* format. The *urllib.urlencode()* function takes a mapping or sequence of 2-tuples and returns a string in this format.

headers should be a dictionary, and will be treated as if *add_header()* was called with each key and value as arguments. This is often used to "spoof" the `User-Agent` header value, which is used by a browser to identify itself –some HTTP servers only allow requests coming from common browsers as opposed to scripts. For example, Mozilla Firefox may identify itself as `"Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11"`, while *urllib2*'s default user agent string is `"Python-urllib/2.6"` (on Python 2.6).

The final two arguments are only of interest for correct handling of third-party HTTP cookies:

origin_req_host should be the request-host of the origin transaction, as defined by **RFC 2965**. It defaults to `cookielib.request_host(self)`. This is the host name or IP address of the original request that was initiated by the user. For example, if the request is for an image in an HTML document, this should be the request-host of the request for the page containing the image.

unverifiable should indicate whether the request is unverifiable, as defined by RFC 2965. It defaults to `False`. An unverifiable request is one whose URL the user did not have the option to approve. For example, if the request is for an image in an HTML document, and the user had no option to approve the automatic fetching of the image, this should be true.

class urllib2.**OpenerDirector**
The *OpenerDirector* class opens URLs via *BaseHandler*s chained together. It manages the chaining of handlers, and recovery from errors.

class urllib2.**BaseHandler**
This is the base class for all registered handlers —and handles only the simple mechanics of registration.

class urllib2.**HTTPDefaultErrorHandler**
A class which defines a default handler for HTTP error responses; all responses are turned into *HTTPError* exceptions.

class urllib2.**HTTPRedirectHandler**
A class to handle redirections.

class urllib2.**HTTPCookieProcessor**([*cookiejar*])
A class to handle HTTP Cookies.

class urllib2.**ProxyHandler**([*proxies*])
Cause requests to go through a proxy. If *proxies* is given, it must be a dictionary mapping protocol names to URLs of proxies. The default is to read the list of proxies from the environment variables `<protocol>_proxy`. If no proxy environment variables are set, then in a Windows environment proxy settings are obtained from the registry's Internet Settings section, and in a Mac OS X environment proxy information is retrieved from the OS X System Configuration Framework.

To disable autodetected proxy pass an empty dictionary.

> **注解：** `HTTP_PROXY` will be ignored if a variable `REQUEST_METHOD` is set; see the documentation on *getproxies()*.

class urllib2.**HTTPPasswordMgr**
Keep a database of `(realm, uri) -> (user, password)` mappings.

class urllib2.**HTTPPasswordMgrWithDefaultRealm**
Keep a database of `(realm, uri) -> (user, password)` mappings. A realm of `None` is considered a catch-all realm, which is searched if no other realm fits.

class urllib2.**AbstractBasicAuthHandler**([*password_mgr*])
This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with `HTTPPasswordMgr`; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib2.**HTTPBasicAuthHandler**([*password_mgr*])
Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with `HTTPPasswordMgr`; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib2.**ProxyBasicAuthHandler**([*password_mgr*])
Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with `HTTPPasswordMgr`; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib2.**AbstractDigestAuthHandler**([*password_mgr*])
This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with `HTTPPasswordMgr`; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib2.**HTTPDigestAuthHandler**([*password_mgr*])
Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with `HTTPPasswordMgr`; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib2.**ProxyDigestAuthHandler**([*password_mgr*])
Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with `HTTPPasswordMgr`; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib2.**HTTPHandler**
A class to handle opening of HTTP URLs.

class urllib2.**HTTPSHandler**([*debuglevel*[, *context*]])
A class to handle opening of HTTPS URLs. *context* has the same meaning as for `httplib.HTTPSConnection`.

在 2.7.9 版更改: *context* added.

class urllib2.**FileHandler**
Open local files.

class urllib2.**FTPHandler**
Open FTP URLs.

class urllib2.**CacheFTPHandler**
Open FTP URLs, keeping a cache of open FTP connections to minimize delays.

class urllib2.**UnknownHandler**
A catch-all class to handle unknown URLs.

class urllib2.**HTTPErrorProcessor**
Process HTTP error responses.

20.6.1 Request Objects

The following methods describe all of `Request`'s public interface, and so all must be overridden in subclasses.

`Request.`**`add_data`**(*data*)
Set the `Request` data to *data*. This is ignored by all handlers except HTTP handlers —and there it should be a byte string, and will change the request to be `POST` rather than `GET`.

`Request.`**`get_method`**()
Return a string indicating the HTTP request method. This is only meaningful for HTTP requests, and currently always returns `'GET'` or `'POST'`.

`Request.`**`has_data`**()
Return whether the instance has a non-`None` data.

`Request.`**`get_data`**()
Return the instance's data.

`Request.`**`add_header`**(*key*, *val*)
Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the *key* collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header.

`Request.`**`add_unredirected_header`**(*key*, *header*)
Add a header that will not be added to a redirected request.

2.4 新版功能.

`Request.`**`has_header`**(*header*)
Return whether the instance has the named header (checks both regular and unredirected).

2.4 新版功能.

`Request.`**`get_full_url`**()
Return the URL given in the constructor.

`Request.`**`get_type`**()
Return the type of the URL —also known as the scheme.

`Request.`**`get_host`**()
Return the host to which a connection will be made.

`Request.`**`get_selector`**()
Return the selector —the part of the URL that is sent to the server.

`Request.`**`get_header`**(*header_name*, *default=None*)
Return the value of the given header. If the header is not present, return the default value.

`Request.`**`header_items`**()
Return a list of tuples (header_name, header_value) of the Request headers.

`Request.`**`set_proxy`**(*host*, *type*)
Prepare the request by connecting to a proxy server. The *host* and *type* will replace those of the instance, and the instance's selector will be the original URL given in the constructor.

`Request.`**`get_origin_req_host`**()
Return the request-host of the origin transaction, as defined by **RFC 2965**. See the documentation for the `Request` constructor.

Request.**is_unverifiable**()
Return whether the request is unverifiable, as defined by RFC 2965. See the documentation for the *Request* constructor.

20.6.2 OpenerDirector Objects

OpenerDirector instances have the following methods:

OpenerDirector.**add_handler**(*handler*)
handler should be an instance of *BaseHandler*. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case).

- *protocol*_open —signal that the handler knows how to open *protocol* URLs.
- http_error_*type* —signal that the handler knows how to handle HTTP errors with HTTP error code *type*.
- *protocol*_error —signal that the handler knows how to handle errors from (non-http) *protocol*.
- *protocol*_request —signal that the handler knows how to pre-process *protocol* requests.
- *protocol*_response —signal that the handler knows how to post-process *protocol* responses.

OpenerDirector.**open**(*url[, data][, timeout]*)
Open the given *url* (which can be a request object or a string), optionally passing the given *data*. Arguments, return values and exceptions raised are the same as those of *urlopen()* (which simply calls the *open()* method on the currently installed global *OpenerDirector*). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). The timeout feature actually works only for HTTP, HTTPS and FTP connections).

在 2.6 版更改: *timeout* was added.

OpenerDirector.**error**(*proto*[, *arg*[, ...]])
Handle an error of the given protocol. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). The HTTP protocol is a special case which uses the HTTP response code to determine the specific error handler; refer to the http_error_*() methods of the handler classes.

Return values and exceptions raised are the same as those of *urlopen()*.

OpenerDirector objects open URLs in three stages:

The order in which these methods are called within each stage is determined by sorting the handler instances.

1. Every handler with a method named like *protocol*_request has that method called to pre-process the request.
2. Handlers with a method named like *protocol*_open are called to handle the request. This stage ends when a handler either returns a non-*None* value (ie. a response), or raises an exception (usually *URLError*). Exceptions are allowed to propagate.

   In fact, the above algorithm is first tried for methods named default_open(). If all such methods return *None*, the algorithm is repeated for methods named like *protocol*_open. If all such methods return *None*, the algorithm is repeated for methods named unknown_open().

   Note that the implementation of these methods may involve calls of the parent *OpenerDirector* instance's *open()* and *error()* methods.
3. Every handler with a method named like *protocol*_response has that method called to post-process the response.

20.6.3 BaseHandler Objects

BaseHandler objects provide a couple of methods that are directly useful, and others that are meant to be used by derived classes. These are intended for direct use:

BaseHandler.**add_parent**(*director*)
Add a director as parent.

BaseHandler.**close**()
Remove any parents.

The following attributes and methods should only be used by classes derived from *BaseHandler*.

注 解: The convention has been adopted that subclasses defining `protocol_request()` or `protocol_response()` methods are named `*Processor`; all others are named `*Handler`.

BaseHandler.**parent**
A valid *OpenerDirector*, which can be used to open using a different protocol, or handle errors.

BaseHandler.**default_open**(*req*)
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to catch all URLs.

This method, if implemented, will be called by the parent *OpenerDirector*. It should return a file-like object as described in the return value of the *open()* of *OpenerDirector*, or `None`. It should raise *URLError*, unless a truly exceptional thing happens (for example, *MemoryError* should not be mapped to *URLError*).

This method will be called before any protocol-specific open method.

BaseHandler.**protocol_open**(*req*)
("protocol" is to be replaced by the protocol name.)

This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to handle URLs with the given *protocol*.

This method, if defined, will be called by the parent *OpenerDirector*. Return values should be the same as for *default_open()*.

BaseHandler.**unknown_open**(*req*)
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to catch all URLs with no specific registered handler to open it.

This method, if implemented, will be called by the *parent OpenerDirector*. Return values should be the same as for *default_open()*.

BaseHandler.**http_error_default**(*req*, *fp*, *code*, *msg*, *hdrs*)
This method is *not* defined in *BaseHandler*, but subclasses should override it if they intend to provide a catch-all for otherwise unhandled HTTP errors. It will be called automatically by the *OpenerDirector* getting the error, and should not normally be called in other circumstances.

req will be a *Request* object, *fp* will be a file-like object with the HTTP error body, *code* will be the three-digit code of the error, *msg* will be the user-visible explanation of the code and *hdrs* will be a mapping object with the headers of the error.

Return values and exceptions raised should be the same as those of *urlopen()*.

BaseHandler.**http_error_nnn**(*req*, *fp*, *code*, *msg*, *hdrs*)
nnn should be a three-digit HTTP error code. This method is also not defined in *BaseHandler*, but will be called, if it exists, on an instance of a subclass, when an HTTP error with code *nnn* occurs.

Subclasses should override this method to handle specific HTTP errors.

Arguments, return values and exceptions raised should be the same as for *http_error_default()*.

BaseHandler.**protocol_request**(*req*)
("protocol" is to be replaced by the protocol name.)

This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to pre-process requests of the given *protocol*.

This method, if defined, will be called by the parent *OpenerDirector*. *req* will be a *Request* object. The return value should be a *Request* object.

BaseHandler.**protocol_response**(*req*, *response*)
("protocol" is to be replaced by the protocol name.)

This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to post-process responses of the given *protocol*.

This method, if defined, will be called by the parent *OpenerDirector*. *req* will be a *Request* object. *response* will be an object implementing the same interface as the return value of *urlopen()*. The return value should implement the same interface as the return value of *urlopen()*.

20.6.4 HTTPRedirectHandler Objects

注解: Some HTTP redirections require action from this module's client code. If this is the case, *HTTPError* is raised. See **RFC 2616** for details of the precise meanings of the various redirection codes.

HTTPRedirectHandler.**redirect_request**(*req*, *fp*, *code*, *msg*, *hdrs*, *newurl*)
Return a *Request* or `None` in response to a redirect. This is called by the default implementations of the `http_error_30*()` methods when a redirection is received from the server. If a redirection should take place, return a new *Request* to allow `http_error_30*()` to perform the redirect to *newurl*. Otherwise, raise *HTTPError* if no other handler should try to handle this URL, or return `None` if you can't but another handler might.

注解: The default implementation of this method does not strictly follow **RFC 2616**, which says that 301 and 302 responses to `POST` requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a `GET`, and the default implementation reproduces this behavior.

HTTPRedirectHandler.**http_error_301**(*req*, *fp*, *code*, *msg*, *hdrs*)
Redirect to the `Location:` or `URI:` URL. This method is called by the parent *OpenerDirector* when getting an HTTP 'moved permanently' response.

HTTPRedirectHandler.**http_error_302**(*req*, *fp*, *code*, *msg*, *hdrs*)
The same as *http_error_301()*, but called for the 'found' response.

HTTPRedirectHandler.**http_error_303**(*req*, *fp*, *code*, *msg*, *hdrs*)
The same as *http_error_301()*, but called for the 'see other' response.

HTTPRedirectHandler.**http_error_307**(*req*, *fp*, *code*, *msg*, *hdrs*)
The same as *http_error_301()*, but called for the 'temporary redirect' response.

20.6.5 HTTPCookieProcessor Objects

2.4 新版功能.

HTTPCookieProcessor instances have one attribute:

HTTPCookieProcessor.**cookiejar**
The *cookielib.CookieJar* in which cookies are stored.

20.6.6 ProxyHandler Objects

ProxyHandler.**protocol_open**(*request*)
(“protocol” is to be replaced by the protocol name.)

The *ProxyHandler* will have a method *protocol*_open for every *protocol* which has a proxy in the *proxies* dictionary given in the constructor. The method will modify requests to go through the proxy, by calling `request.set_proxy()`, and call the next handler in the chain to actually execute the protocol.

20.6.7 HTTPPasswordMgr Objects

These methods are available on *HTTPPasswordMgr* and *HTTPPasswordMgrWithDefaultRealm* objects.

HTTPPasswordMgr.**add_password**(*realm*, *uri*, *user*, *passwd*)
uri can be either a single URI, or a sequence of URIs. *realm*, *user* and *passwd* must be strings. This causes `(user, passwd)` to be used as authentication tokens when authentication for *realm* and a super-URI of any of the given URIs is given.

HTTPPasswordMgr.**find_user_password**(*realm*, *authuri*)
Get user/password for given realm and URI, if any. This method will return `(None, None)` if there is no matching user/password.

For *HTTPPasswordMgrWithDefaultRealm* objects, the realm `None` will be searched if the given *realm* has no matching user/password.

20.6.8 AbstractBasicAuthHandler Objects

AbstractBasicAuthHandler.**http_error_auth_reqed**(*authreq*, *host*, *req*, *headers*)
Handle an authentication request by getting a user/password pair, and re-trying the request. *authreq* should be the name of the header where the information about the realm is included in the request, *host* specifies the URL and path to authenticate for, *req* should be the (failed) *Request* object, and *headers* should be the error headers.

host is either an authority (e.g. `"python.org"`) or a URL containing an authority component (e.g. `"http://python.org/"`). In either case, the authority must not contain a userinfo component (so, `"python.org"` and `"python.org:80"` are fine, `"joe:password@python.org"` is not).

20.6.9 HTTPBasicAuthHandler Objects

`HTTPBasicAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

20.6.10 ProxyBasicAuthHandler Objects

`ProxyBasicAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

20.6.11 AbstractDigestAuthHandler Objects

`AbstractDigestAuthHandler.`**`http_error_auth_reqed`**(*authreq*, *host*, *req*, *headers*)
authreq should be the name of the header where the information about the realm is included in the request, *host* should be the host to authenticate to, *req* should be the (failed) `Request` object, and *headers* should be the error headers.

20.6.12 HTTPDigestAuthHandler Objects

`HTTPDigestAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

20.6.13 ProxyDigestAuthHandler Objects

`ProxyDigestAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

20.6.14 HTTPHandler Objects

`HTTPHandler.`**`http_open`**(*req*)
Send an HTTP request, which can be either GET or POST, depending on `req.has_data()`.

20.6.15 HTTPSHandler Objects

`HTTPSHandler.`**`https_open`**(*req*)
Send an HTTPS request, which can be either GET or POST, depending on `req.has_data()`.

20.6.16 FileHandler Objects

`FileHandler.`**`file_open`**(*req*)
Open the file locally, if there is no host name, or the host name is `'localhost'`. Change the protocol to `ftp` otherwise, and retry opening it using `parent`.

20.6.17 FTPHandler Objects

`FTPHandler.`**`ftp_open`**`(req)`
Open the FTP file indicated by *req*. The login is always done with empty username and password.

20.6.18 CacheFTPHandler Objects

`CacheFTPHandler` objects are *`FTPHandler`* objects with the following additional methods:

`CacheFTPHandler.`**`setTimeout`**`(t)`
Set timeout of connections to *t* seconds.

`CacheFTPHandler.`**`setMaxConns`**`(m)`
Set maximum number of cached connections to *m*.

20.6.19 UnknownHandler Objects

`UnknownHandler.`**`unknown_open`**`()`
Raise a *`URLError`* exception.

20.6.20 HTTPErrorProcessor Objects

2.4 新版功能.

`HTTPErrorProcessor.`**`http_response`**`()`
Process HTTP error responses.

For 200 error codes, the response object is returned immediately.

For non-200 error codes, this simply passes the job on to the *`protocol`*`_error_code` handler methods, via *`OpenerDirector.error()`*. Eventually, *`urllib2.HTTPDefaultErrorHandler`* will raise an *`HTTPError`* if no other handler handles the error.

`HTTPErrorProcessor.`**`https_response`**`()`
Process HTTPS error responses.

The behavior is same as *`http_response()`*.

20.6.21 Examples

In addition to the examples below, more examples are given in urllib-howto.

This example gets the python.org main page and displays the first 100 bytes of it:

```
>>> import urllib2
>>> f = urllib2.urlopen('http://www.python.org/')
>>> print f.read(100)
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<?xml-stylesheet href="./css/ht2html
```

Here we are sending a data-stream to the stdin of a CGI and reading the data it returns to us. Note that this example will only work when the Python installation supports SSL.

```
>>> import urllib2
>>> req = urllib2.Request(url='https://localhost/cgi-bin/test.cgi',
...                       data='This data is passed to stdin of the CGI')
>>> f = urllib2.urlopen(req)
>>> print f.read()
Got Data: "This data is passed to stdin of the CGI"
```

The code for the sample CGI used in the above example is:

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print 'Content-type: text-plain\n\nGot Data: "%s"' % data
```

Use of Basic HTTP Authentication:

```
import urllib2
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib2.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib2.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib2.install_opener(opener)
urllib2.urlopen('http://www.example.com/login.html')
```

build_opener() provides many handlers by default, including a *ProxyHandler*. By default, *ProxyHandler* uses the environment variables named <scheme>_proxy, where <scheme> is the URL scheme involved. For example, the http_proxy environment variable is read to obtain the HTTP proxy's URL.

This example replaces the default *ProxyHandler* with one that uses programmatically-supplied proxy URLs, and adds proxy authorization support with *ProxyBasicAuthHandler*.

```
proxy_handler = urllib2.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib2.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib2.build_opener(proxy_handler, proxy_auth_handler)
# This time, rather than install the OpenerDirector, we use it directly:
opener.open('http://www.example.com/login.html')
```

Adding HTTP headers:

Use the *headers* argument to the *Request* constructor, or:

```
import urllib2
req = urllib2.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
# Customize the default User-Agent header value:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . .)')
r = urllib2.urlopen(req)
```

OpenerDirector automatically adds a *User-Agent* header to every *Request*. To change this:

```
import urllib2
opener = urllib2.build_opener()
```

(下页继续)

(续上页)

```
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

Also, remember that a few standard headers (*Content-Length*, *Content-Type* and *Host*) are added when the *Request* is passed to *urlopen()* (or *OpenerDirector.open()*).

20.7 httplib —HTTP protocol client

注解: The *httplib* module has been renamed to `http.client` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/httplib.py

This module defines classes which implement the client side of the HTTP and HTTPS protocols. It is normally not used directly —the module *urllib* uses it to handle URLs that use HTTP and HTTPS.

参见:

The Requests package is recommended for a higher-level HTTP client interface.

注解: HTTPS support is only available if the *socket* module was compiled with SSL support.

注解: The public interface for this module changed substantially in Python 2.0. The `HTTP` class is retained only for backward compatibility with 1.5.2. It should not be used in new code. Refer to the online docstrings for usage.

The module provides the following classes:

class `httplib.`**HTTPConnection**(*host*[, *port*[, *strict*[, *timeout*[, *source_address*]]]])

An *HTTPConnection* instance represents one transaction with an HTTP server. It should be instantiated passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form `host:port`, else the default HTTP port (80) is used. When true, the optional parameter *strict* (which defaults to a false value) causes `BadStatusLine` to be raised if the status line can't be parsed as a valid HTTP/1.0 or 1.1 status line. If the optional *timeout* parameter is given, blocking operations (like connection attempts) will timeout after that many seconds (if it is not given, the global default timeout setting is used). The optional *source_address* parameter may be a tuple of a (host, port) to use as the source address the HTTP connection is made from.

For example, the following calls all create instances that connect to the server at the same host and port:

```
>>> h1 = httplib.HTTPConnection('www.cwi.nl')
>>> h2 = httplib.HTTPConnection('www.cwi.nl:80')
>>> h3 = httplib.HTTPConnection('www.cwi.nl', 80)
>>> h3 = httplib.HTTPConnection('www.cwi.nl', 80, timeout=10)
```

2.0 新版功能.

在 2.6 版更改: *timeout* was added.

在 2.7 版更改: *source_address* was added.

class httplib.**HTTPSConnection**(*host*[, *port*[, *key_file*[, *cert_file*[, *strict*[, *timeout*[, *source_address*[, *context*]]]]]]])

A subclass of *HTTPConnection* that uses SSL for communication with secure servers. Default port is 443. If *context* is specified, it must be a *ssl.SSLContext* instance describing the various SSL options.

key_file and *cert_file* are deprecated, please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

Please read *Security considerations* for more information on best practices.

2.0 新版功能.

在 2.6 版更改: *timeout* was added.

在 2.7 版更改: *source_address* was added.

在 2.7.9 版更改: *context* was added.

This class now performs all the necessary certificate and hostname checks by default. To revert to the previous, unverified, behavior ssl._create_unverified_context() can be passed to the *context* parameter.

class httplib.**HTTPResponse**(*sock*, *debuglevel=0*, *strict=0*)

Class whose instances are returned upon successful connection. Not instantiated directly by user.

2.0 新版功能.

class httplib.**HTTPMessage**

An *HTTPMessage* instance is used to hold the headers from an HTTP response. It is implemented using the *mimetools.Message* class and provides utility functions to deal with HTTP Headers. It is not directly instantiated by the users.

The following exceptions are raised as appropriate:

exception httplib.**HTTPException**

The base class of the other exceptions in this module. It is a subclass of *Exception*.

2.0 新版功能.

exception httplib.**NotConnected**

A subclass of *HTTPException*.

2.0 新版功能.

exception httplib.**InvalidURL**

A subclass of *HTTPException*, raised if a port is given and is either non-numeric or empty.

2.3 新版功能.

exception httplib.**UnknownProtocol**

A subclass of *HTTPException*.

2.0 新版功能.

exception httplib.**UnknownTransferEncoding**

A subclass of *HTTPException*.

2.0 新版功能.

exception httplib.**UnimplementedFileMode**

A subclass of *HTTPException*.

2.0 新版功能.

exception httplib.**IncompleteRead**

A subclass of *HTTPException*.

2.0 新版功能.

exception httplib.**ImproperConnectionState**
A subclass of *HTTPException*.

2.0 新版功能.

exception httplib.**CannotSendRequest**
A subclass of *ImproperConnectionState*.

2.0 新版功能.

exception httplib.**CannotSendHeader**
A subclass of *ImproperConnectionState*.

2.0 新版功能.

exception httplib.**ResponseNotReady**
A subclass of *ImproperConnectionState*.

2.0 新版功能.

exception httplib.**BadStatusLine**
A subclass of *HTTPException*. Raised if a server responds with a HTTP status code that we don't understand.

2.0 新版功能.

The constants defined in this module are:

httplib.**HTTP_PORT**
The default port for the HTTP protocol (always 80).

httplib.**HTTPS_PORT**
The default port for the HTTPS protocol (always 443).

and also the following constants for integer status codes:

Constant	Value	Definition
CONTINUE	100	HTTP/1.1, RFC 2616, Section 10.1.1
SWITCHING_PROTOCOLS	101	HTTP/1.1, RFC 2616, Section 10.1.2
PROCESSING	102	WEBDAV, RFC 2518, Section 10.1
OK	200	HTTP/1.1, RFC 2616, Section 10.2.1
CREATED	201	HTTP/1.1, RFC 2616, Section 10.2.2
ACCEPTED	202	HTTP/1.1, RFC 2616, Section 10.2.3
NON_AUTHORITATIVE_INFORMATION	203	HTTP/1.1, RFC 2616, Section 10.2.4
NO_CONTENT	204	HTTP/1.1, RFC 2616, Section 10.2.5
RESET_CONTENT	205	HTTP/1.1, RFC 2616, Section 10.2.6
PARTIAL_CONTENT	206	HTTP/1.1, RFC 2616, Section 10.2.7
MULTI_STATUS	207	WEBDAV RFC 2518, Section 10.2
IM_USED	226	Delta encoding in HTTP, **RFC 3229**, Section 10.4.1
MULTIPLE_CHOICES	300	HTTP/1.1, RFC 2616, Section 10.3.1
MOVED_PERMANENTLY	301	HTTP/1.1, RFC 2616, Section 10.3.2
FOUND	302	HTTP/1.1, RFC 2616, Section 10.3.3
SEE_OTHER	303	HTTP/1.1, RFC 2616, Section 10.3.4
NOT_MODIFIED	304	HTTP/1.1, RFC 2616, Section 10.3.5
USE_PROXY	305	HTTP/1.1, RFC 2616, Section 10.3.6
TEMPORARY_REDIRECT	307	HTTP/1.1, RFC 2616, Section 10.3.8
BAD_REQUEST	400	HTTP/1.1, RFC 2616, Section 10.4.1
UNAUTHORIZED	401	HTTP/1.1, RFC 2616, Section 10.4.2
PAYMENT_REQUIRED	402	HTTP/1.1, RFC 2616, Section 10.4.3

下页继续

表 1 – 续上页

Constant	Value	Definition
FORBIDDEN	403	HTTP/1.1, RFC 2616, Section 10.4.4
NOT_FOUND	404	HTTP/1.1, RFC 2616, Section 10.4.5
METHOD_NOT_ALLOWED	405	HTTP/1.1, RFC 2616, Section 10.4.6
NOT_ACCEPTABLE	406	HTTP/1.1, RFC 2616, Section 10.4.7
PROXY_AUTHENTICATION_REQUIRED	407	HTTP/1.1, RFC 2616, Section 10.4.8
REQUEST_TIMEOUT	408	HTTP/1.1, RFC 2616, Section 10.4.9
CONFLICT	409	HTTP/1.1, RFC 2616, Section 10.4.10
GONE	410	HTTP/1.1, RFC 2616, Section 10.4.11
LENGTH_REQUIRED	411	HTTP/1.1, RFC 2616, Section 10.4.12
PRECONDITION_FAILED	412	HTTP/1.1, RFC 2616, Section 10.4.13
REQUEST_ENTITY_TOO_LARGE	413	HTTP/1.1, RFC 2616, Section 10.4.14
REQUEST_URI_TOO_LONG	414	HTTP/1.1, RFC 2616, Section 10.4.15
UNSUPPORTED_MEDIA_TYPE	415	HTTP/1.1, RFC 2616, Section 10.4.16
REQUESTED_RANGE_NOT_SATISFIABLE	416	HTTP/1.1, RFC 2616, Section 10.4.17
EXPECTATION_FAILED	417	HTTP/1.1, RFC 2616, Section 10.4.18
UNPROCESSABLE_ENTITY	422	WEBDAV, RFC 2518, Section 10.3
LOCKED	423	WEBDAV RFC 2518, Section 10.4
FAILED_DEPENDENCY	424	WEBDAV, RFC 2518, Section 10.5
UPGRADE_REQUIRED	426	HTTP Upgrade to TLS, **RFC 2817**, Section 6
INTERNAL_SERVER_ERROR	500	HTTP/1.1, RFC 2616, Section 10.5.1
NOT_IMPLEMENTED	501	HTTP/1.1, RFC 2616, Section 10.5.2
BAD_GATEWAY	502	HTTP/1.1 RFC 2616, Section 10.5.3
SERVICE_UNAVAILABLE	503	HTTP/1.1, RFC 2616, Section 10.5.4
GATEWAY_TIMEOUT	504	HTTP/1.1 RFC 2616, Section 10.5.5
HTTP_VERSION_NOT_SUPPORTED	505	HTTP/1.1, RFC 2616, Section 10.5.6
INSUFFICIENT_STORAGE	507	WEBDAV, RFC 2518, Section 10.6
NOT_EXTENDED	510	An HTTP Extension Framework, **RFC 2774**, Section 7

httplib.**responses**
This dictionary maps the HTTP 1.1 status codes to the W3C names.

Example: `httplib.responses[httplib.NOT_FOUND]` is `'Not Found'`.

2.5 新版功能.

20.7.1 HTTPConnection Objects

HTTPConnection instances have the following methods:

HTTPConnection.**request**(*method*, *url*[, *body*[, *headers*]])
This will send a request to the server using the HTTP request method *method* and the selector *url*. If the *body* argument is present, it should be a string of data to send after the headers are finished. Alternatively, it may be an open file object, in which case the contents of the file is sent; this file object should support `fileno()` and `read()` methods. The *headers* argument should be a mapping of extra HTTP headers to send with the request.

If one is not provided in *headers*, a `Content-Length` header is added automatically for all methods if the length of the body can be determined, either from the length of the `str` representation, or from the reported size of the file on disk. If *body* is `None` the header is not set except for methods that expect a body (`PUT`, `POST`, and `PATCH`) in which case it is set to `0`.

在 2.6 版更改: *body* can be a file object.

HTTPConnection.**getresponse**()
Should be called after a request is sent to get the response from the server. Returns an *HTTPResponse* instance.

注解: Note that you must have read the whole response before you can send a new request to the server.

HTTPConnection.**set_debuglevel**(*level*)
Set the debugging level (the amount of debugging output printed). The default debug level is 0, meaning no debugging output is printed.

HTTPConnection.**set_tunnel**(*host*, *port=None*, *headers=None*)
Set the host and the port for HTTP Connect Tunnelling. Normally used when it is required to do HTTPS Connection through a proxy server.

The headers argument should be a mapping of extra HTTP headers to send with the CONNECT request.

2.7 新版功能.

HTTPConnection.**connect**()
Connect to the server specified when the object was created.

HTTPConnection.**close**()
Close the connection to the server.

As an alternative to using the request() method described above, you can also send your request step by step, by using the four functions below.

HTTPConnection.**putrequest**(*request*, *selector*[, *skip_host*[, *skip_accept_encoding*]])
This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the *request* string, the *selector* string, and the HTTP version (HTTP/1.1). To disable automatic sending of Host: or Accept-Encoding: headers (for example to accept additional content encodings), specify *skip_host* or *skip_accept_encoding* with non-False values.

在 2.4 版更改: *skip_accept_encoding* argument added.

HTTPConnection.**putheader**(*header*, *argument*[, ...])
Send an **RFC 822**-style header to the server. It sends a line to the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

HTTPConnection.**endheaders**(*message_body=None*)
Send a blank line to the server, signalling the end of the headers. The optional *message_body* argument can be used to pass a message body associated with the request. The message body will be sent in the same packet as the message headers if it is string, otherwise it is sent in a separate packet.

在 2.7 版更改: *message_body* was added.

HTTPConnection.**send**(*data*)
Send data to the server. This should be used directly only after the *endheaders()* method has been called and before *getresponse()* is called.

20.7.2 HTTPResponse Objects

HTTPResponse instances have the following methods and attributes:

HTTPResponse.**read**([*amt*])
Reads and returns the response body, or up to the next *amt* bytes.

HTTPResponse.**getheader**(*name*[, *default*])
Get the contents of the header *name*, or *default* if there is no matching header.

HTTPResponse.**getheaders**()
Return a list of (header, value) tuples.

2.4 新版功能.

HTTPResponse.**fileno**()
Returns the `fileno` of the underlying socket.

HTTPResponse.**msg**
A *mimetools.Message* instance containing the response headers.

HTTPResponse.**version**
HTTP protocol version used by server. 10 for HTTP/1.0, 11 for HTTP/1.1.

HTTPResponse.**status**
Status code returned by server.

HTTPResponse.**reason**
Reason phrase returned by server.

20.7.3 Examples

Here is an example session that uses the `GET` method:

```
>>> import httplib
>>> conn = httplib.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print r1.status, r1.reason
200 OK
>>> data1 = r1.read()
>>> conn.request("GET", "/")
>>> r2 = conn.getresponse()
>>> print r2.status, r2.reason
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

Here is an example session that uses the `HEAD` method. Note that the `HEAD` method never returns any data.

```
>>> import httplib
>>> conn = httplib.HTTPSConnection("www.python.org")
>>> conn.request("HEAD","/")
>>> res = conn.getresponse()
>>> print res.status, res.reason
200 OK
>>> data = res.read()
>>> print len(data)
0
```

(下页继续)

(续上页)

```
>>> data == ''
True
```

Here is an example session that shows how to POST requests:

```
>>> import httplib, urllib
>>> params = urllib.urlencode({'@number': 12524, '@type': 'issue', '@action': 'show'})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...            "Accept": "text/plain"}
>>> conn = httplib.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print response.status, response.reason
302 Found
>>> data = response.read()
>>> data
'Redirecting to <a href="http://bugs.python.org/issue12524">http://bugs.python.org/
↪issue12524</a>'
>>> conn.close()
```

Client side HTTP PUT requests are very similar to POST requests. The difference lies only the server side where HTTP server will allow resources to be created via PUT request. Here is an example session that shows how to do PUT request using httplib:

```
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/foobar
...
>>> import httplib
>>> BODY = "***filecontents***"
>>> conn = httplib.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print response.status, response.reason
200, OK
```

20.8 ftplib —FTP 协议客户端

源代码： Lib/ftplib.py

This module defines the class *FTP* and a few related items. The *FTP* class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other FTP servers. It is also used by the module *urllib* to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see Internet **RFC 959**.

Here's a sample session using the *ftplib* module:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.debian.org')     # connect to host, default port
>>> ftp.login()                     # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')               # change into "debian" directory
```

(下页继续)

(续上页)

```
>>> ftp.retrlines('LIST')           # list directory contents
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> ftp.retrbinary('RETR README', open('README', 'wb').write)
'226 Transfer complete.'
>>> ftp.quit()
```

这个模块定义了以下内容:

class ftplib.**FTP**([*host*[, *user*[, *passwd*[, *acct*[, *timeout*]]]]])

Return a new instance of the *FTP* class. When *host* is given, the method call `connect(host)` is made. When *user* is given, additionally the method call `login(user, passwd, acct)` is made (where *passwd* and *acct* default to the empty string when not given). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if is not specified, the global default timeout setting will be used).

在 2.6 版更改: *timeout* was added.

class ftplib.**FTP_TLS**([*host*[, *user*[, *passwd*[, *acct*[, *keyfile*[, *certfile*[, *context*[, *timeout*]]]]]]]])

A *FTP* subclass which adds TLS support to FTP as described in **RFC 4217**. Connect as usual to port 21 implicitly securing the FTP control connection before authenticating. Securing the data connection requires the user to explicitly ask for it by calling the *prot_p()* method. *context* is a *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Security considerations* for best practices.

keyfile and *certfile* are a legacy alternative to *context* –they can point to PEM-formatted private key and certificate chain files (respectively) for the SSL connection.

2.7 新版功能.

在 2.7.10 版更改: The *context* parameter was added.

Here’s a sample session using the *FTP_TLS* class:

```
>>> from ftplib import FTP_TLS
>>> ftps = FTP_TLS('ftp.python.org')
>>> ftps.login()           # login anonymously before securing control channel
>>> ftps.prot_p()          # switch to secure data connection
>>> ftps.retrlines('LIST') # list directory content securely
total 9
drwxr-xr-x   8 root     wheel        1024 Jan  3  1994 .
drwxr-xr-x   8 root     wheel        1024 Jan  3  1994 ..
drwxr-xr-x   2 root     wheel        1024 Jan  3  1994 bin
drwxr-xr-x   2 root     wheel        1024 Jan  3  1994 etc
d-wxrwxr-x   2 ftp      wheel        1024 Sep  5 13:43 incoming
drwxr-xr-x   2 root     wheel        1024 Nov 17  1993 lib
drwxr-xr-x   6 1094     wheel        1024 Sep 13 19:07 pub
drwxr-xr-x   3 root     wheel        1024 Jan  3  1994 usr
-rw-r--r--   1 root     root          312 Aug  1  1994 welcome.msg
'226 Transfer complete.'
>>> ftps.quit()
>>>
```

exception ftplib.**error_reply**

Exception raised when an unexpected reply is received from the server.

exception ftplib.**error_temp**
Exception raised when an error code signifying a temporary error (response codes in the range 400–499) is received.

exception ftplib.**error_perm**
Exception raised when an error code signifying a permanent error (response codes in the range 500–599) is received.

exception ftplib.**error_proto**
Exception raised when a reply is received from the server that does not fit the response specifications of the File Transfer Protocol, i.e. begin with a digit in the range 1–5.

ftplib.**all_errors**
The set of all exceptions (as a tuple) that methods of *FTP* instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed above as well as *socket.error* and *IOError*.

参见:

Module ***netrc*** Parser for the `.netrc` file format. The file `.netrc` is typically used by FTP clients to load user authentication information before prompting the user.

The file `Tools/scripts/ftpmirror.py` in the Python source distribution is a script that can mirror FTP sites, or portions thereof, using the *ftplib* module. It can be used as an extended example that applies this module.

20.8.1 FTP Objects

Several methods are available in two flavors: one for handling text files and another for binary files. These are named for the command which is used followed by `lines` for the text version or `binary` for the binary version.

FTP instances have the following methods:

FTP.**set_debuglevel**(*level*)
Set the instance's debugging level. This controls the amount of debugging output printed. The default, `0`, produces no debugging output. A value of `1` produces a moderate amount of debugging output, generally a single line per request. A value of `2` or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

FTP.**connect**(*host*[, *port*[, *timeout*]])
Connect to the given host and port. The default port number is `21`, as specified by the FTP protocol specification. It is rarely needed to specify a different port number. This function should be called only once for each instance; it should not be called at all if a host was given when the instance was created. All other methods can only be used after a connection has been made.

The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If no *timeout* is passed, the global default timeout setting will be used.

在 2.6 版更改: *timeout* was added.

FTP.**getwelcome**()
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

FTP.**login**([*user*[, *passwd*[, *acct*]]])
Log in as the given *user*. The *passwd* and *acct* parameters are optional and default to the empty string. If no *user* is specified, it defaults to `'anonymous'`. If *user* is `'anonymous'`, the default *passwd* is `'anonymous@'`. This function should be called only once for each instance, after a connection has been established; it should not be called at all if a host and user were given when the instance was created. Most FTP commands are only allowed after the client has logged in. The *acct* parameter supplies "accounting information"; few systems implement this.

FTP.**abort**()
Abort a file transfer that is in progress. Using this does not always work, but it's worth a try.

FTP.**sendcmd**(*command*)
Send a simple command string to the server and return the response string.

FTP.**voidcmd**(*command*)
Send a simple command string to the server and handle the response. Return nothing if a response code corresponding to success (codes in the range 200–299) is received. Raise *error_reply* otherwise.

FTP.**retrbinary**(*command*, *callback*[, *maxblocksize*[, *rest*]])
Retrieve a file in binary transfer mode. *command* should be an appropriate RETR command: 'RETR filename'. The *callback* function is called for each block of data received, with a single string argument giving the data block. The optional *maxblocksize* argument specifies the maximum chunk size to read on the low-level socket object created to do the actual transfer (which will also be the largest size of the data blocks passed to *callback*). A reasonable default is chosen. *rest* means the same thing as in the *transfercmd()* method.

FTP.**retrlines**(*command*[, *callback*])
Retrieve a file or directory listing in ASCII transfer mode. *command* should be an appropriate RETR command (see *retrbinary()*) or a command such as LIST, NLST or MLSD (usually just the string 'LIST'). LIST retrieves a list of files and information about those files. NLST retrieves a list of file names. On some servers, MLSD retrieves a machine readable list of files and information about those files. The *callback* function is called for each line with a string argument containing the line with the trailing CRLF stripped. The default *callback* prints the line to sys.stdout.

FTP.**set_pasv**(*val*)
Enable "passive" mode if *val* is true, otherwise disable passive mode. (In Python 2.0 and before, passive mode was off by default; in Python 2.1 and later, it is on by default.)

FTP.**storbinary**(*command*, *fp*[, *blocksize*, *callback*, *rest*])
Store a file in binary transfer mode. *command* should be an appropriate STOR command: "STOR filename". *fp* is an open file object which is read until EOF using its read() method in blocks of size *blocksize* to provide the data to be stored. The *blocksize* argument defaults to 8192. *callback* is an optional single parameter callable that is called on each block of data after it is sent. *rest* means the same thing as in the *transfercmd()* method.

在 2.1 版更改: default for *blocksize* added.

在 2.6 版更改: *callback* parameter added.

在 2.7 版更改: *rest* parameter added.

FTP.**storlines**(*command*, *fp*[, *callback*])
Store a file in ASCII transfer mode. *command* should be an appropriate STOR command (see *storbinary()*). Lines are read until EOF from the open file object *fp* using its *readline()* method to provide the data to be stored. *callback* is an optional single parameter callable that is called on each line after it is sent.

在 2.6 版更改: *callback* parameter added.

FTP.**transfercmd**(*cmd*[, *rest*])
Initiate a transfer over the data connection. If the transfer is active, send an EPRT or PORT command and the transfer command specified by *cmd*, and accept the connection. If the server is passive, send an EPSV or PASV command, connect to it, and start the transfer command. Either way, return the socket for the connection.

If optional *rest* is given, a REST command is sent to the server, passing *rest* as an argument. *rest* is usually a byte offset into the requested file, telling the server to restart sending the file's bytes at the requested offset, skipping over the initial bytes. Note however that RFC 959 requires only that *rest* be a string containing characters in the printable range from ASCII code 33 to ASCII code 126. The *transfercmd()* method, therefore, converts *rest* to a string, but no check is performed on the string's contents. If the server does not recognize the REST command, an *error_reply* exception will be raised. If this happens, simply call *transfercmd()* without a *rest* argument.

FTP.**ntransfercmd**(*cmd*[, *rest*])
Like *transfercmd()*, but returns a tuple of the data connection and the expected size of the data. If the expected size could not be computed, None will be returned as the expected size. *cmd* and *rest* means the same thing as in *transfercmd()*.

FTP.**nlst**(*argument*[, ...])
Return a list of file names as returned by the NLST command. The optional *argument* is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the NLST command.

FTP.**dir**(*argument*[, ...])
Produce a directory listing as returned by the LIST command, printing it to standard output. The optional *argument* is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the LIST command. If the last argument is a function, it is used as a *callback* function as for *retrlines()*; the default prints to sys.stdout. This method returns None.

FTP.**rename**(*fromname*, *toname*)
Rename file *fromname* on the server to *toname*.

FTP.**delete**(*filename*)
Remove the file named *filename* from the server. If successful, returns the text of the response, otherwise raises *error_perm* on permission errors or *error_reply* on other errors.

FTP.**cwd**(*pathname*)
Set the current directory on the server.

FTP.**mkd**(*pathname*)
Create a new directory on the server.

FTP.**pwd**()
Return the pathname of the current directory on the server.

FTP.**rmd**(*dirname*)
Remove the directory named *dirname* on the server.

FTP.**size**(*filename*)
Request the size of the file named *filename* on the server. On success, the size of the file is returned as an integer, otherwise None is returned. Note that the SIZE command is not standardized, but is supported by many common server implementations.

FTP.**quit**()
Send a QUIT command to the server and close the connection. This is the "polite" way to close a connection, but it may raise an exception if the server responds with an error to the QUIT command. This implies a call to the *close()* method which renders the *FTP* instance useless for subsequent calls (see below).

FTP.**close**()
Close the connection unilaterally. This should not be applied to an already closed connection such as after a successful call to *quit()*. After this call the *FTP* instance should not be used any more (after a call to *close()* or *quit()* you cannot reopen the connection by issuing another *login()* method).

20.8.2 FTP_TLS Objects

FTP_TLS class inherits from *FTP*, defining these additional objects:

FTP_TLS.**ssl_version**
The SSL version to use (defaults to *ssl.PROTOCOL_SSLv23*).

FTP_TLS.**auth**()
Set up secure control connection by using TLS or SSL, depending on what specified in *ssl_version()* attribute.

FTP_TLS.**prot_p**()
Set up secure data connection.

FTP_TLS.**prot_c**()
Set up clear text data connection.

20.9 poplib —POP3 protocol client

Source code: Lib/poplib.py

This module defines a class, *POP3*, which encapsulates a connection to a POP3 server and implements the protocol as defined in **RFC 1725**. The *POP3* class supports both the minimal and optional command sets. Additionally, this module provides a class *POP3_SSL*, which provides support for connecting to POP3 servers that use SSL as an underlying protocol layer.

Note that POP3, though widely supported, is obsolescent. The implementation quality of POP3 servers varies widely, and too many are quite poor. If your mailserver supports IMAP, you would be better off using the *imaplib.IMAP4* class, as IMAP servers tend to be better implemented.

The *poplib* module provides two classes:

class poplib.**POP3**(*host*[, *port*[, *timeout*]])
This class implements the actual POP3 protocol. The connection is created when the instance is initialized. If *port* is omitted, the standard POP3 port (110) is used. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt (if not specified, the global default timeout setting will be used).

在 2.6 版更改: *timeout* was added.

class poplib.**POP3_SSL**(*host*[, *port*[, *keyfile*[, *certfile*]]])
This is a subclass of *POP3* that connects to the server over an SSL encrypted socket. If *port* is not specified, 995, the standard POP3-over-SSL port is used. *keyfile* and *certfile* are also optional - they can contain a PEM formatted private key and certificate chain file for the SSL connection.

2.4 新版功能.

One exception is defined as an attribute of the *poplib* module:

exception poplib.**error_proto**
Exception raised on any errors from this module (errors from *socket* module are not caught). The reason for the exception is passed to the constructor as a string.

参见:

Module *imaplib* The standard Python IMAP module.

Frequently Asked Questions About Fetchmail The FAQ for the **fetchmail** POP/IMAP client collects information on POP3 server variations and RFC noncompliance that may be useful if you need to write an application based on the POP protocol.

20.9.1 POP3 Objects

All POP3 commands are represented by methods of the same name, in lower-case; most return the response text sent by the server.

An *POP3* instance has the following methods:

POP3.**set_debuglevel**(*level*)
Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

POP3.**getwelcome**()
Returns the greeting string sent by the POP3 server.

POP3.**user**(*username*)
Send user command, response should indicate that a password is required.

POP3.**pass_**(*password*)
Send password, response includes message count and mailbox size. Note: the mailbox on the server is locked until `quit()` is called.

POP3.**apop**(*user*, *secret*)
Use the more secure APOP authentication to log into the POP3 server.

POP3.**rpop**(*user*)
Use RPOP authentication (similar to UNIX r-commands) to log into POP3 server.

POP3.**stat**()
Get mailbox status. The result is a tuple of 2 integers: `(message count, mailbox size)`.

POP3.**list**([*which*])
Request message list, result is in the form `(response, ['mesg_num octets', ...], octets)`. If *which* is set, it is the message to list.

POP3.**retr**(*which*)
Retrieve whole message number *which*, and set its seen flag. Result is in form `(response, ['line', ...], octets)`.

POP3.**dele**(*which*)
Flag message number *which* for deletion. On most servers deletions are not actually performed until QUIT (the major exception is Eudora QPOP, which deliberately violates the RFCs by doing pending deletes on any disconnect).

POP3.**rset**()
Remove any deletion marks for the mailbox.

POP3.**noop**()
Do nothing. Might be used as a keep-alive.

POP3.**quit**()
Signoff: commit changes, unlock mailbox, drop connection.

POP3.**top**(*which*, *howmuch*)
Retrieves the message header plus *howmuch* lines of the message after the header of message number *which*. Result is in form `(response, ['line', ...], octets)`.

The POP3 TOP command this method uses, unlike the RETR command, doesn't set the message's seen flag; unfortunately, TOP is poorly specified in the RFCs and is frequently broken in off-brand servers. Test this method by hand against the POP3 servers you will use before trusting it.

POP3.**uidl**([*which*])
Return message digest (unique id) list. If *which* is specified, result contains the unique id for that message in the form `'response mesgnum uid`, otherwise result is list `(response, ['mesgnum uid', ...], octets)`.

Instances of *POP3_SSL* have no additional methods. The interface of this subclass is identical to its parent.

20.9.2 POP3 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print j
```

At the end of the module, there is a test section that contains a more extensive example of usage.

20.10 imaplib —IMAP4 protocol client

Source code: Lib/imaplib.py

This module defines three classes, *IMAP4*, *IMAP4_SSL* and *IMAP4_stream*, which encapsulate a connection to an IMAP4 server and implement a large subset of the IMAP4rev1 client protocol as defined in **RFC 2060**. It is backward compatible with IMAP4 (**RFC 1730**) servers, but note that the `STATUS` command is not supported in IMAP4.

Three classes are provided by the *imaplib* module, *IMAP4* is the base class:

class imaplib.**IMAP4**([*host*[, *port*]])
This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4 port (143) is used.

Three exceptions are defined as attributes of the *IMAP4* class:

exception IMAP4.**error**
Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

exception IMAP4.**abort**
IMAP4 server errors cause this exception to be raised. This is a sub-class of *IMAP4.error*. Note that closing the instance and instantiating a new one will usually allow recovery from this exception.

exception IMAP4.**readonly**
This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of *IMAP4.error*. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.

There's also a subclass for secure connections:

class imaplib.**IMAP4_SSL**([*host*[, *port*[, *keyfile*[, *certfile*]]]])
This is a subclass derived from *IMAP4* that connects over an SSL encrypted socket (to use this class you need a socket module that was compiled with SSL support). If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4-over-SSL port (993) is used. *keyfile* and *certfile* are also optional - they can contain a PEM formatted private key and certificate chain file for the SSL connection.

The second subclass allows for connections created by a child process:

class imaplib.**IMAP4_stream**(*command*)
This is a subclass derived from *IMAP4* that connects to the `stdin/stdout` file descriptors created by passing *command* to `os.popen2()`.

2.3 新版功能.

The following utility functions are defined:

imaplib.**Internaldate2tuple**(*datestr*)
Parse an IMAP4 `INTERNALDATE` string and return corresponding local time. The return value is a *time.struct_time* instance or `None` if the string has wrong format.

imaplib.**Int2AP**(*num*)
Converts an integer into a string representation using characters from the set [`A` .. `P`].

imaplib.**ParseFlags**(*flagstr*)
Converts an IMAP4 `FLAGS` response to a tuple of individual flags.

imaplib.**Time2Internaldate**(*date_time*)
Convert *date_time* to an IMAP4 `INTERNALDATE` representation. The return value is a string in the form: `"DD-Mmm-YYYY HH:MM:SS +HHMM"` (including double-quotes). The *date_time* argument can be a number (int or float) representing seconds since epoch (as returned by *time.time()*), a 9-tuple representing local time (as returned by *time.localtime()*), or a double-quoted string. In the last case, it is assumed to already be in the correct format.

Note that IMAP4 message numbers change as the mailbox changes; in particular, after an `EXPUNGE` command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.

At the end of the module, there is a test section that contains a more extensive example of usage.

参见:

Documents describing the protocol, and sources and binaries for servers implementing it, can all be found at the University of Washington's *IMAP Information Center* (https://www.washington.edu/imap/).

20.10.1 IMAP4 Objects

All IMAP4rev1 commands are represented by methods of the same name, either upper-case or lower-case.

All arguments to commands are converted to strings, except for `AUTHENTICATE`, and the last argument to `APPEND` which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn't enclosed with either parentheses or double quotes) each string is quoted. However, the *password* argument to the `LOGIN` command is always quoted. If you want to avoid having an argument string quoted (eg: the *flags* argument to `STORE`) then enclose the string in parentheses (eg: `r'(\Deleted)'`).

Each command returns a tuple: `(type, [data, ...])` where *type* is usually `'OK'` or `'NO'`, and *data* is either the text from the command response, or mandated results from the command. Each *data* is either a string, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie: 'literal' value).

The *message_set* options to commands below is a string specifying one or more messages to be acted upon. It may be a simple message number (`'1'`), a range of message numbers (`'2:4'`), or a group of non-contiguous ranges separated by commas (`'1:3,6:9'`). A range can contain an asterisk to indicate an infinite upper bound (`'3:*'`).

An *IMAP4* instance has the following methods:

IMAP4.**append**(*mailbox*, *flags*, *date_time*, *message*)
Append *message* to named mailbox.

IMAP4.**authenticate**(*mechanism*, *authobject*)
Authenticate command —requires response processing.

mechanism specifies which authentication mechanism is to be used - it should appear in the instance variable `capabilities` in the form `AUTH=mechanism`.

authobject must be a callable object:

```
data = authobject(response)
```

It will be called to process server continuation responses. It should return `data` that will be encoded and sent to server. It should return `None` if the client abort response `*` should be sent instead.

IMAP4.**check**()
Checkpoint mailbox on server.

IMAP4.**close**()
Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before `LOGOUT`.

IMAP4.**copy**(*message_set*, *new_mailbox*)
Copy *message_set* messages onto end of *new_mailbox*.

IMAP4.**create**(*mailbox*)
Create new mailbox named *mailbox*.

IMAP4.**delete**(*mailbox*)
Delete old mailbox named *mailbox*.

IMAP4.**deleteacl**(*mailbox*, *who*)
Delete the ACLs (remove any rights) set for who on mailbox.

2.4 新版功能.

IMAP4.**expunge**()
Permanently remove deleted items from selected mailbox. Generates an `EXPUNGE` response for each deleted message. Returned data contains a list of `EXPUNGE` message numbers in order received.

IMAP4.**fetch**(*message_set*, *message_parts*)
Fetch (parts of) messages. *message_parts* should be a string of message part names enclosed within parentheses, eg: `"(UID BODY[TEXT])"`. Returned data are tuples of message part envelope and data.

IMAP4.**getacl**(*mailbox*)
Get the `ACL`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.**getannotation**(*mailbox*, *entry*, *attribute*)
Retrieve the specified `ANNOTATION`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

2.5 新版功能.

IMAP4.**getquota**(*root*)
Get the `quota` *root*'s resource usage and limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

2.3 新版功能.

IMAP4.**getquotaroot**(*mailbox*)
Get the list of `quota roots` for the named *mailbox*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

2.3 新版功能.

IMAP4.**list**([*directory*[, *pattern*]])
List mailbox names in *directory* matching *pattern*. *directory* defaults to the top-level mail folder, and *pattern* defaults to match anything. Returned data contains a list of `LIST` responses.

IMAP4.**login**(*user*, *password*)
Identify the client using a plaintext password. The *password* will be quoted.

IMAP4.**login_cram_md5**(*user*, *password*)
Force use of `CRAM-MD5` authentication when identifying the client to protect the password. Will only work if the server `CAPABILITY` response includes the phrase `AUTH=CRAM-MD5`.

2.3 新版功能.

IMAP4.**logout**()
Shutdown connection to server. Returns server `BYE` response.

IMAP4.**lsub**([*directory*[, *pattern*]])
List subscribed mailbox names in directory matching pattern. *directory* defaults to the top level directory and *pattern* defaults to match any mailbox. Returned data are tuples of message part envelope and data.

IMAP4.**myrights**(*mailbox*)
Show my ACLs for a mailbox (i.e. the rights that I have on mailbox).

2.4 新版功能.

IMAP4.**namespace**()
Returns IMAP namespaces as defined in RFC2342.

2.3 新版功能.

IMAP4.**noop**()
Send `NOOP` to server.

IMAP4.**open**(*host*, *port*)
Opens socket to *port* at *host*. This method is implicitly called by the *IMAP4* constructor. The connection objects established by this method will be used in the *IMAP4.read()*, *IMAP4.readline()*, *IMAP4.send()*, and *IMAP4.shutdown()* methods. You may override this method.

IMAP4.**partial**(*message_num*, *message_part*, *start*, *length*)
Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.

IMAP4.**proxyauth**(*user*)
Assume authentication as *user*. Allows an authorised administrator to proxy into any user's mailbox.

2.3 新版功能.

IMAP4.**read**(*size*)
Reads *size* bytes from the remote server. You may override this method.

IMAP4.**readline**()
Reads one line from the remote server. You may override this method.

IMAP4.**recent**()
Prompt server for an update. Returned data is `None` if no new messages, else value of `RECENT` response.

IMAP4.**rename**(*oldmailbox*, *newmailbox*)
Rename mailbox named *oldmailbox* to *newmailbox*.

IMAP4.**response**(*code*)
Return data for response *code* if received, or None. Returns the given code, instead of the usual type.

IMAP4.**search**(*charset*, *criterion*[, ...])
Search mailbox for matching messages. *charset* may be None, in which case no CHARSET will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified; an exception will be raised when the server returns an error.

示例:

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

IMAP4.**select**([*mailbox*[, *readonly*]])
Select a mailbox. Returned data is the count of messages in *mailbox* (EXISTS response). The default *mailbox* is 'INBOX'. If the *readonly* flag is set, modifications to the mailbox are not allowed.

IMAP4.**send**(*data*)
Sends data to the remote server. You may override this method.

IMAP4.**setacl**(*mailbox*, *who*, *what*)
Set an ACL for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

IMAP4.**setannotation**(*mailbox*, *entry*, *attribute*[, ...])
Set ANNOTATIONs for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

2.5 新版功能.

IMAP4.**setquota**(*root*, *limits*)
Set the quota *root*'s resource *limits*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

2.3 新版功能.

IMAP4.**shutdown**()
Close connection established in open. This method is implicitly called by *IMAP4.logout()*. You may override this method.

IMAP4.**socket**()
Returns socket instance used to connect to server.

IMAP4.**sort**(*sort_criteria*, *charset*, *search_criterion*[, ...])
The sort command is a variant of search with sorting semantics for the results. Returned data contains a space separated list of matching message numbers.

Sort has two arguments before the *search_criterion* argument(s); a parenthesized list of *sort_criteria*, and the searching *charset*. Note that unlike search, the searching *charset* argument is mandatory. There is also a uid sort command which corresponds to sort the way that uid search corresponds to search. The sort command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the numbers of matching messages.

This is an IMAP4rev1 extension command.

IMAP4.**status**(*mailbox*, *names*)
Request named status conditions for *mailbox*.

IMAP4.**store**(*message_set*, *command*, *flag_list*)
Alters flag dispositions for messages in mailbox. *command* is specified by section 6.4.6 of **RFC 2060** as being one of "FLAGS", "+FLAGS", or "-FLAGS", optionally with a suffix of ".SILENT".

For example, to set the delete flag on all messages:

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

IMAP4.**subscribe**(*mailbox*)
Subscribe to new mailbox.

IMAP4.**thread**(*threading_algorithm*, *charset*, *search_criterion*[, ...])
The `thread` command is a variant of `search` with threading semantics for the results. Returned data contains a space separated list of thread members.

Thread members consist of zero or more messages numbers, delimited by spaces, indicating successive parent and child.

Thread has two arguments before the *search_criterion* argument(s); a *threading_algorithm*, and the searching *charset*. Note that unlike `search`, the searching *charset* argument is mandatory. There is also a `uid thread` command which corresponds to `thread` the way that `uid search` corresponds to `search`. The `thread` command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the matching messages threaded according to the specified threading algorithm.

This is an `IMAP4rev1` extension command.

2.4 新版功能.

IMAP4.**uid**(*command*, *arg*[, ...])
Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied; if none are provided, the server will return an error and an exception will be raised.

IMAP4.**unsubscribe**(*mailbox*)
Unsubscribe from old mailbox.

IMAP4.**xatom**(*name*[, *arg*[, ...]])
Allow simple extension commands notified by server in `CAPABILITY` response.

Instances of *IMAP4_SSL* have just one additional method:

IMAP4_SSL.**ssl**()
Returns SSLObject instance used for the secure connection with the server.

The following attributes are defined on instances of *IMAP4*:

IMAP4.**PROTOCOL_VERSION**
The most recent supported protocol in the `CAPABILITY` response from the server.

IMAP4.**debug**
Integer value to control debugging output. The initialize value is taken from the module variable `Debug`. Values greater than three trace each command.

20.10.2 IMAP4 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
import getpass, imaplib

M = imaplib.IMAP4()
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print 'Message %s\n%s\n' % (num, data[0][1])
M.close()
M.logout()
```

20.11 nntplib —NNTP protocol client

Source code: Lib/nntplib.py

This module defines the class *NNTP* which implements the client side of the NNTP protocol. It can be used to implement a news reader or poster, or automated news processors. For more information on NNTP (Network News Transfer Protocol), see Internet **RFC 977**.

Here are two small examples of how it can be used. To list some statistics about a newsgroup and print the subjects of the last 10 articles:

```
>>> s = NNTP('news.gmane.org')
>>> resp, count, first, last, name = s.group('gmane.comp.python.committers')
>>> print 'Group', name, 'has', count, 'articles, range', first, 'to', last
Group gmane.comp.python.committers has 1071 articles, range 1 to 1071
>>> resp, subs = s.xhdr('subject', first + '-' + last)
>>> for id, sub in subs[-10:]: print id, sub
...
1062 Re: Mercurial Status?
1063 Re: [python-committers]  (Windows) buildbots on 3.x
1064 Re: Mercurial Status?
1065 Re: Mercurial Status?
1066 Python 2.6.6 status
1067 Commit Privileges for Ask Solem
1068 Re: Commit Privileges for Ask Solem
1069 Re: Commit Privileges for Ask Solem
1070 Re: Commit Privileges for Ask Solem
1071 2.6.6 rc 2
>>> s.quit()
'205 Bye!'
```

To post an article from a file (this assumes that the article has valid headers, and that you have right to post on the particular newsgroup):

```
>>> s = NNTP('news.gmane.org')
>>> f = open('articlefile')
>>> s.post(f)
```

(下页继续)

(续上页)

```
'240 Article posted successfully.'
>>> s.quit()
'205 Bye!'
```

The module itself defines the following items:

class nntplib.**NNTP**(*host[, port [, user[, password [, readermode] [, usenetrc]]]]*)
Return a new instance of the *NNTP* class, representing a connection to the NNTP server running on host *host*, listening at port *port*. The default *port* is 119. If the optional *user* and *password* are provided, or if suitable credentials are present in `/.netrc` and the optional flag *usenetrc* is true (the default), the `AUTHINFO USER` and `AUTHINFO PASS` commands are used to identify and authenticate the user to the server. If the optional flag *readermode* is true, then a `mode reader` command is sent before authentication is performed. Reader mode is sometimes necessary if you are connecting to an NNTP server on the local machine and intend to call reader-specific commands, such as `group`. If you get unexpected *NNTPPermanentError*s, you might need to set *readermode*. *readermode* defaults to `None`. *usenetrc* defaults to `True`.

在 2.4 版更改: *usenetrc* argument added.

exception nntplib.**NNTPError**
Derived from the standard exception *Exception*, this is the base class for all exceptions raised by the *nntplib* module.

exception nntplib.**NNTPReplyError**
Exception raised when an unexpected reply is received from the server. For backwards compatibility, the exception `error_reply` is equivalent to this class.

exception nntplib.**NNTPTemporaryError**
Exception raised when an error code in the range 400–499 is received. For backwards compatibility, the exception `error_temp` is equivalent to this class.

exception nntplib.**NNTPPermanentError**
Exception raised when an error code in the range 500–599 is received. For backwards compatibility, the exception `error_perm` is equivalent to this class.

exception nntplib.**NNTPProtocolError**
Exception raised when a reply is received from the server that does not begin with a digit in the range 1–5. For backwards compatibility, the exception `error_proto` is equivalent to this class.

exception nntplib.**NNTPDataError**
Exception raised when there is some error in the response data. For backwards compatibility, the exception `error_data` is equivalent to this class.

20.11.1 NNTP Objects

NNTP instances have the following methods. The *response* that is returned as the first item in the return tuple of almost all methods is the server's response: a string beginning with a three-digit code. If the server's response indicates an error, the method raises one of the above exceptions.

NNTP.**getwelcome**()
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

NNTP.**set_debuglevel**(*level*)
Set the instance's debugging level. This controls the amount of debugging output printed. The default, `0`, produces no debugging output. A value of `1` produces a moderate amount of debugging output, generally a single line per request or response. A value of `2` or higher produces the maximum amount of debugging output, logging each line sent and received on the connection (including message text).

NNTP.**newgroups**(*date*, *time*[, *file*])
Send a NEWGROUPS command. The *date* argument should be a string of the form 'yymmdd' indicating the date, and *time* should be a string of the form 'hhmmss' indicating the time. Return a pair (response, groups) where *groups* is a list of group names that are new since the given date and time. If the *file* parameter is supplied, then the output of the NEWGROUPS command is stored in a file. If *file* is a string, then the method will open a file object with that name, write to it then close it. If *file* is a file object, then it will start calling write() on it to store the lines of the command output. If *file* is supplied, then the returned *list* is an empty list.

NNTP.**newnews**(*group*, *date*, *time*[, *file*])
Send a NEWNEWS command. Here, *group* is a group name or '*', and *date* and *time* have the same meaning as for *newgroups()*. Return a pair (response, articles) where *articles* is a list of message ids. If the *file* parameter is supplied, then the output of the NEWNEWS command is stored in a file. If *file* is a string, then the method will open a file object with that name, write to it then close it. If *file* is a file object, then it will start calling write() on it to store the lines of the command output. If *file* is supplied, then the returned *list* is an empty list.

NNTP.**list**([*file*])
Send a LIST command. Return a pair (response, list) where *list* is a list of tuples. Each tuple has the form (group, last, first, flag), where *group* is a group name, *last* and *first* are the last and first article numbers (as strings), and *flag* is 'y' if posting is allowed, 'n' if not, and 'm' if the newsgroup is moderated. (Note the ordering: *last*, *first*.) If the *file* parameter is supplied, then the output of the LIST command is stored in a file. If *file* is a string, then the method will open a file object with that name, write to it then close it. If *file* is a file object, then it will start calling write() on it to store the lines of the command output. If *file* is supplied, then the returned *list* is an empty list.

NNTP.**descriptions**(*grouppattern*)
Send a LIST NEWSGROUPS command, where *grouppattern* is a wildmat string as specified in RFC2980 (it's essentially the same as DOS or UNIX shell wildcard strings). Return a pair (response, list), where *list* is a list of tuples containing (name, title).

2.4 新版功能.

NNTP.**description**(*group*)
Get a description for a single group *group*. If more than one group matches (if 'group' is a real wildmat string), return the first match. If no group matches, return an empty string.

This elides the response code from the server. If the response code is needed, use *descriptions()*.

2.4 新版功能.

NNTP.**group**(*name*)
Send a GROUP command, where *name* is the group name. Return a tuple (response, count, first, last, name) where *count* is the (estimated) number of articles in the group, *first* is the first article number in the group, *last* is the last article number in the group, and *name* is the group name. The numbers are returned as strings.

NNTP.**help**([*file*])
Send a HELP command. Return a pair (response, list) where *list* is a list of help strings. If the *file* parameter is supplied, then the output of the HELP command is stored in a file. If *file* is a string, then the method will open a file object with that name, write to it then close it. If *file* is a file object, then it will start calling write() on it to store the lines of the command output. If *file* is supplied, then the returned *list* is an empty list.

NNTP.**stat**(*id*)
Send a STAT command, where *id* is the message id (enclosed in '<' and '>') or an article number (as a string). Return a triple (response, number, id) where *number* is the article number (as a string) and *id* is the message id (enclosed in '<' and '>').

NNTP.**next**()
Send a NEXT command. Return as for *stat()*.

NNTP.**last**()
Send a LAST command. Return as for *stat()*.

NNTP.**head**(*id*)
Send a HEAD command, where *id* has the same meaning as for *stat()*. Return a tuple (response, number, id, list) where the first three are the same as for *stat()*, and *list* is a list of the article's headers (an uninterpreted list of lines, without trailing newlines).

NNTP.**body**(*id*[, *file*])
Send a BODY command, where *id* has the same meaning as for *stat()*. If the *file* parameter is supplied, then the body is stored in a file. If *file* is a string, then the method will open a file object with that name, write to it then close it. If *file* is a file object, then it will start calling write() on it to store the lines of the body. Return as for *head()*. If *file* is supplied, then the returned *list* is an empty list.

NNTP.**article**(*id*)
Send an ARTICLE command, where *id* has the same meaning as for *stat()*. Return as for *head()*.

NNTP.**slave**()
Send a SLAVE command. Return the server's *response*.

NNTP.**xhdr**(*header*, *string*[, *file*])
Send an XHDR command. This command is not defined in the RFC but is a common extension. The *header* argument is a header keyword, e.g. 'subject'. The *string* argument should have the form 'first-last' where *first* and *last* are the first and last article numbers to search. Return a pair (response, list), where *list* is a list of pairs (id, text), where *id* is an article number (as a string) and *text* is the text of the requested header for that article. If the *file* parameter is supplied, then the output of the XHDR command is stored in a file. If *file* is a string, then the method will open a file object with that name, write to it then close it. If *file* is a file object, then it will start calling write() on it to store the lines of the command output. If *file* is supplied, then the returned *list* is an empty list.

NNTP.**post**(*file*)
Post an article using the POST command. The *file* argument is an open file object which is read until EOF using its *readline()* method. It should be a well-formed news article, including the required headers. The *post()* method automatically escapes lines beginning with ..

NNTP.**ihave**(*id*, *file*)
Send an IHAVE command. *id* is a message id (enclosed in '<' and '>'). If the response is not an error, treat *file* exactly as for the *post()* method.

NNTP.**date**()
Return a triple (response, date, time), containing the current date and time in a form suitable for the *newnews()* and *newgroups()* methods. This is an optional NNTP extension, and may not be supported by all servers.

NNTP.**xgtitle**(*name*[, *file*])
Process an XGTITLE command, returning a pair (response, list), where *list* is a list of tuples containing (name, title). If the *file* parameter is supplied, then the output of the XGTITLE command is stored in a file. If *file* is a string, then the method will open a file object with that name, write to it then close it. If *file* is a file object, then it will start calling write() on it to store the lines of the command output. If *file* is supplied, then the returned *list* is an empty list. This is an optional NNTP extension, and may not be supported by all servers.

RFC2980 says "It is suggested that this extension be deprecated". Use *descriptions()* or *description()* instead.

NNTP.**xover**(*start*, *end*[, *file*])
Return a pair (resp, list). *list* is a list of tuples, one for each article in the range delimited by the *start* and *end* article numbers. Each tuple is of the form (article number, subject, poster, date, id, references, size, lines). If the *file* parameter is supplied, then the output of the XOVER command is stored in a file. If *file* is a string, then the method will open a file object with that name, write to it then close it.

If *file* is a file object, then it will start calling `write()` on it to store the lines of the command output. If *file* is supplied, then the returned *list* is an empty list. This is an optional NNTP extension, and may not be supported by all servers.

`NNTP.`**`xpath`**`(`*id*`)`
Return a pair `(resp, path)`, where *path* is the directory path to the article with message ID *id*. This is an optional NNTP extension, and may not be supported by all servers.

`NNTP.`**`quit`**`()`
Send a `QUIT` command and close the connection. Once this method has been called, no other methods of the NNTP object should be called.

20.12 `smtplib` —SMTP 协议客户端

Source code: Lib/smtplib.py

The *smtplib* module defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon. For details of SMTP and ESMTP operation, consult **RFC 821** (Simple Mail Transfer Protocol) and **RFC 1869** (SMTP Service Extensions).

class `smtplib.`**`SMTP`**`(`[*host*[, *port*[, *local_hostname*[, *timeout*]]]]`)`
An *SMTP* instance encapsulates an SMTP connection. It has methods that support a full repertoire of SMTP and ESMTP operations. If the optional host and port parameters are given, the SMTP *connect()* method is called with those parameters during initialization. If specified, *local_hostname* is used as the FQDN of the local host in the HELO/EHLO command. Otherwise, the local hostname is found using *socket.getfqdn()*. If the *connect()* call returns anything other than a success code, an *SMTPConnectError* is raised. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). If the timeout expires, *socket.timeout* is raised.

For normal use, you should only require the initialization/connect, *sendmail()*, and *SMTP.quit()* methods. An example is included below.

在 2.6 版更改: *timeout* was added.

class `smtplib.`**`SMTP_SSL`**`(`[*host*[, *port*[, *local_hostname*[, *keyfile*[, *certfile*[, *timeout*]]]]]]`)`
An *SMTP_SSL* instance behaves exactly the same as instances of *SMTP*. *SMTP_SSL* should be used for situations where SSL is required from the beginning of the connection and using `starttls()` is not appropriate. If *host* is not specified, the local host is used. If *port* is omitted, the standard SMTP-over-SSL port (465) is used. *local_hostname* has the same meaning as it does for the *SMTP* class. *keyfile* and *certfile* are also optional, and can contain a PEM formatted private key and certificate chain file for the SSL connection. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). If the timeout expires, *socket.timeout* is raised.

2.6 新版功能.

class `smtplib.`**`LMTP`**`(`[*host*[, *port*[, *local_hostname*]]]`)`
The LMTP protocol, which is very similar to ESMTP, is heavily based on the standard SMTP client. It's common to use Unix sockets for LMTP, so our `connect()` method must support that as well as a regular host:port server. *local_hostname* has the same meaning as it does for the *SMTP* class. To specify a Unix socket, you must use an absolute path for *host*, starting with a '/'.

Authentication is supported, using the regular SMTP mechanism. When using a Unix socket, LMTP generally don't support or require any authentication, but your mileage might vary.

2.6 新版功能.

A nice selection of exceptions is defined as well:

exception smtplib.**SMTPException**
The base exception class for all the other exceptions provided by this module.

exception smtplib.**SMTPServerDisconnected**
This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the *SMTP* instance before connecting it to a server.

exception smtplib.**SMTPResponseException**
Base class for all exceptions that include an SMTP error code. These exceptions are generated in some instances when the SMTP server returns an error code. The error code is stored in the `smtp_code` attribute of the error, and the `smtp_error` attribute is set to the error message.

exception smtplib.**SMTPSenderRefused**
Sender address refused. In addition to the attributes set by on all *SMTPResponseException* exceptions, this sets 'sender' to the string that the SMTP server refused.

exception smtplib.**SMTPRecipientsRefused**
All recipient addresses refused. The errors for each recipient are accessible through the attribute `recipients`, which is a dictionary of exactly the same sort as *SMTP.sendmail()* returns.

exception smtplib.**SMTPDataError**
The SMTP server refused to accept the message data.

exception smtplib.**SMTPConnectError**
Error occurred during establishment of a connection with the server.

exception smtplib.**SMTPHeloError**
The server refused our `HELO` message.

exception smtplib.**SMTPAuthenticationError**
SMTP authentication went wrong. Most probably the server didn't accept the username/password combination provided.

参见:

RFC 821 - Simple Mail Transfer Protocol Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

RFC 1869 - SMTP Service Extensions Definition of the ESMTP extensions for SMTP. This describes a framework for extending SMTP with new commands, supporting dynamic discovery of the commands provided by the server, and defines a few additional commands.

20.12.1 SMTP Objects

An *SMTP* instance has the following methods:

`SMTP`.**set_debuglevel**(*level*)
Set the debug output level. A true value for *level* results in debug messages for connection and for all messages sent to and received from the server.

`SMTP`.**docmd**(*cmd*[, *argstring*])
Send a command *cmd* to the server. The optional argument *argstring* is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, *SMTPServerDisconnected* will be raised.

SMTP.**connect**([*host*[, *port*]])
Connect to a host on a given port. The defaults are to connect to the local host at the standard SMTP port (25). If the hostname ends with a colon (':') followed by a number, that suffix will be stripped off and the number interpreted as the port number to use. This method is automatically invoked by the constructor if a host is specified during instantiation. Returns a 2-tuple of the response code and message sent by the server in its connection response.

SMTP.**helo**([*hostname*])
Identify yourself to the SMTP server using HELO. The hostname argument defaults to the fully qualified domain name of the local host. The message returned by the server is stored as the helo_resp attribute of the object.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the *sendmail()* when necessary.

SMTP.**ehlo**([*hostname*])
Identify yourself to an ESMTP server using EHLO. The hostname argument defaults to the fully qualified domain name of the local host. Examine the response for ESMTP option and store them for use by *has_extn()*. Also sets several informational attributes: the message returned by the server is stored as the ehlo_resp attribute, does_esmtp is set to true or false depending on whether the server supports ESMTP, and esmtp_features will be a dictionary containing the names of the SMTP service extensions this server supports, and their parameters (if any).

Unless you wish to use *has_extn()* before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by *sendmail()* when necessary.

SMTP.**ehlo_or_helo_if_needed**()
This method call *ehlo()* and or *helo()* if there has been no previous EHLO or HELO command this session. It tries ESMTP EHLO first.

SMTPHeloError The server didn't reply properly to the HELO greeting.

2.6 新版功能.

SMTP.**has_extn**(*name*)
Return *True* if *name* is in the set of SMTP service extensions returned by the server, *False* otherwise. Case is ignored.

SMTP.**verify**(*address*)
Check the validity of an address on this server using SMTP VRFY. Returns a tuple consisting of code 250 and a full **RFC 822** address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

注解: Many sites disable SMTP VRFY in order to foil spammers.

SMTP.**login**(*user*, *password*)
Log in on an SMTP server that requires authentication. The arguments are the username and the password to authenticate with. If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. This method will return normally if the authentication was successful, or may raise the following exceptions:

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTPAuthenticationError The server didn't accept the username/password combination.

SMTPException No suitable authentication method was found.

SMTP.**starttls**([*keyfile*[, *certfile*]])
Put the SMTP connection in TLS (Transport Layer Security) mode. All SMTP commands that follow will be encrypted. You should then call *ehlo()* again.

If *keyfile* and *certfile* are provided, these are passed to the *socket* module's *ssl()* function.

If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first.

在 2.6 版更改.

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTPException The server does not support the STARTTLS extension.

在 2.6 版更改.

RuntimeError SSL/TLS support is not available to your Python interpreter.

SMTP.**sendmail**(*from_addr*, *to_addrs*, *msg*[, *mail_options*, *rcpt_options*])

Send mail. The required arguments are an **RFC 822** from-address string, a list of **RFC 822** to-address strings (a bare string will be treated as a list with 1 address), and a message string. The caller may pass a list of ESMTP options (such as 8bitmime) to be used in MAIL FROM commands as *mail_options*. ESMTP options (such as DSN commands) that should be used with all RCPT commands can be passed as *rcpt_options*. (If you need to use different ESMTP options to different recipients you have to use the low-level methods such as mail(), rcpt() and data() to send the message.)

注解: The *from_addr* and *to_addrs* parameters are used to construct the message envelope used by the transport agents. The *SMTP* does not modify the message headers in any way.

If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If EHLO fails, HELO will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will raise an exception. That is, if this method does not raise an exception, then someone should get your mail. If this method does not raise an exception, it returns a dictionary, with one entry for each recipient that was refused. Each entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

This method may raise the following exceptions:

SMTPRecipientsRefused All recipients were refused. Nobody got the mail. The recipients attribute of the exception object is a dictionary with information about the refused recipients (like the one returned when at least one recipient was accepted).

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTPSenderRefused The server didn't accept the *from_addr*.

SMTPDataError The server replied with an unexpected error code (other than a refusal of a recipient).

Unless otherwise noted, the connection will be open even after an exception is raised.

SMTP.**quit**()

Terminate the SMTP session and close the connection. Return the result of the SMTP QUIT command.

在 2.6 版更改: Return a value.

Low-level methods corresponding to the standard SMTP/ESMTP commands HELP, RSET, NOOP, MAIL, RCPT, and DATA are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.

20.12.2 SMTP Example

This example prompts the user for addresses needed in the message envelope ('To' and 'From' addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered; this example doesn't do any processing of the **RFC 822** headers. In particular, the 'To' and 'From' addresses must be included in the message headers explicitly.

```
import smtplib

def prompt(prompt):
    return raw_input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs  = prompt("To: ").split()
print "Enter message, end with ^D (Unix) or ^Z (Windows):"

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n"
       % (fromaddr, ", ".join(toaddrs)))
while 1:
    try:
        line = raw_input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print "Message length is " + repr(len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

注解: In general, you will want to use the *email* package's features to construct an email message, which you can then convert to a string and send via `sendmail()`; see *email: Examples*.

20.13 smtpd —SMTP 服务器

源代码: Lib/smtpd.py

This module offers several classes to implement SMTP servers. One is a generic do-nothing implementation, which can be overridden, while the other two offer specific mail-sending strategies.

20.13.1 SMTPServer 对象

class smtpd.**SMTPServer**(*localaddr*, *remoteaddr*)

Create a new *SMTPServer* object, which binds to local address *localaddr*. It will treat *remoteaddr* as an upstream SMTP relayer. Both *localaddr* and *remoteaddr* should be a *(host, port)* tuple. The object inherits from *asyncore.dispatcher*, and so will insert itself into *asyncore*'s event loop on instantiation.

process_message(*peer*, *mailfrom*, *rcpttos*, *data*)

Raise *NotImplementedError* exception. Override this in subclasses to do something useful with this message. Whatever was passed in the constructor as *remoteaddr* will be available as the `_remoteaddr` attribute. *peer* is the remote host's address, *mailfrom* is the envelope originator, *rcpttos* are the envelope recipients and *data* is a string containing the contents of the e-mail (which should be in **RFC 2822** format).

20.13.2 DebuggingServer 对象

class smtpd.**DebuggingServer**(*localaddr*, *remoteaddr*)

Create a new debugging server. Arguments are as per *SMTPServer*. Messages will be discarded, and printed on stdout.

20.13.3 PureProxy 对象

class smtpd.**PureProxy**(*localaddr*, *remoteaddr*)

Create a new pure proxy server. Arguments are as per *SMTPServer*. Everything will be relayed to *remoteaddr*. Note that running this has a good chance to make you into an open relay, so please be careful.

20.13.4 MailmanProxy 对象

class smtpd.**MailmanProxy**(*localaddr*, *remoteaddr*)

Create a new pure proxy server. Arguments are as per *SMTPServer*. Everything will be relayed to *remoteaddr*, unless local mailman configurations knows about an address, in which case it will be handled via mailman. Note that running this has a good chance to make you into an open relay, so please be careful.

20.14 telnetlib —Telnet client

Source code: Lib/telnetlib.py

The *telnetlib* module provides a *Telnet* class that implements the Telnet protocol. See **RFC 854** for details about the protocol. In addition, it provides symbolic constants for the protocol characters (see below), and for the telnet options. The symbolic names of the telnet options follow the definitions in `arpa/telnet.h`, with the leading `TELOPT_` removed. For symbolic names of options which are traditionally not included in `arpa/telnet.h`, see the module source itself.

The symbolic constants for the telnet commands are: IAC, DONT, DO, WONT, WILL, SE (Subnegotiation End), NOP (No Operation), DM (Data Mark), BRK (Break), IP (Interrupt process), AO (Abort output), AYT (Are You There), EC (Erase Character), EL (Erase Line), GA (Go Ahead), SB (Subnegotiation Begin).

class telnetlib.**Telnet**([*host*[, *port*[, *timeout*]]])

Telnet represents a connection to a Telnet server. The instance is initially not connected by default; the *open()* method must be used to establish a connection. Alternatively, the host name and optional port number can be passed to the constructor, to, in which case the connection to the server will be established before the constructor

returns. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

Do not reopen an already connected instance.

This class has many `read_*()` methods. Note that some of them raise *EOFError* when the end of the connection is read, because they can return an empty string for other reasons. See the individual descriptions below.

在 2.6 版更改: *timeout* was added.

参见:

RFC 854 - Telnet Protocol Specification Definition of the Telnet protocol.

20.14.1 Telnet Objects

Telnet instances have the following methods:

`Telnet.`**`read_until`**`(`*expected*`[`, *timeout*`])`
Read until a given string, *expected*, is encountered or until *timeout* seconds have passed.

When no match is found, return whatever is available instead, possibly the empty string. Raise *EOFError* if the connection is closed and no cooked data is available.

`Telnet.`**`read_all`**`()`
Read all data until EOF; block until connection closed.

`Telnet.`**`read_some`**`()`
Read at least one byte of cooked data unless EOF is hit. Return `''` if EOF is hit. Block if no data is immediately available.

`Telnet.`**`read_very_eager`**`()`
Read everything that can be without blocking in I/O (eager).

Raise *EOFError* if connection closed and no cooked data available. Return `''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

`Telnet.`**`read_eager`**`()`
Read readily available data.

Raise *EOFError* if connection closed and no cooked data available. Return `''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

`Telnet.`**`read_lazy`**`()`
Process and return data already in the queues (lazy).

Raise *EOFError* if connection closed and no data available. Return `''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

`Telnet.`**`read_very_lazy`**`()`
Return any data available in the cooked queue (very lazy).

Raise *EOFError* if connection closed and no data available. Return `''` if no cooked data available otherwise. This method never blocks.

`Telnet.`**`read_sb_data`**`()`
Return the data collected between a SB/SE pair (suboption begin/end). The callback should access these data when it was invoked with a `SE` command. This method never blocks.

2.3 新版功能.

Telnet.**open**(*host*[, *port*[, *timeout*]])
Connect to a host. The optional second argument is the port number, which defaults to the standard Telnet port (23). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

Do not try to reopen an already connected instance.

在 2.6 版更改: *timeout* was added.

Telnet.**msg**(*msg*[, **args*])
Print a debug message when the debug level is > 0. If extra arguments are present, they are substituted in the message using the standard string formatting operator.

Telnet.**set_debuglevel**(*debuglevel*)
Set the debug level. The higher the value of *debuglevel*, the more debug output you get (on `sys.stdout`).

Telnet.**close**()
关闭连接对象。

Telnet.**get_socket**()
Return the socket object used internally.

Telnet.**fileno**()
Return the file descriptor of the socket object used internally.

Telnet.**write**(*buffer*)
Write a string to the socket, doubling any IAC characters. This can block if the connection is blocked. May raise *socket.error* if the connection is closed.

Telnet.**interact**()
Interaction function, emulates a very dumb Telnet client.

Telnet.**mt_interact**()
Multithreaded version of *interact()*.

Telnet.**expect**(*list*[, *timeout*])
Read until one from a list of a regular expressions matches.

The first argument is a list of regular expressions, either compiled (`regex objects`) or uncompiled (strings). The optional second argument is a timeout, in seconds; the default is to block indefinitely.

Return a tuple of three items: the index in the list of the first regular expression that matches; the match object returned; and the text read up till and including the match.

If end of file is found and no text was read, raise *EOFError*. Otherwise, when nothing matches, return `(-1, None, text)` where *text* is the text received so far (may be the empty string if a timeout happened).

If a regular expression ends with a greedy match (such as `.*`) or if more than one expression can match the same input, the results are non-deterministic, and may depend on the I/O timing.

Telnet.**set_option_negotiation_callback**(*callback*)
Each time a telnet option is read on the input flow, this *callback* (if set) is called with the following parameters: callback(telnet socket, command (DO/DONT/WILL/WONT), option). No other action is done afterwards by telnetlib.

20.14.2 Telnet Example

A simple example illustrating typical use:

```
import getpass
import sys
import telnetlib

HOST = "localhost"
user = raw_input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until("login: ")
tn.write(user + "\n")
if password:
    tn.read_until("Password: ")
    tn.write(password + "\n")

tn.write("ls\n")
tn.write("exit\n")

print tn.read_all()
```

20.15 uuid —UUID objects according to RFC 4122

2.5 新版功能.

This module provides immutable *UUID* objects (the *UUID* class) and the functions *uuid1()*, *uuid3()*, *uuid4()*, *uuid5()* for generating version 1, 3, 4, and 5 UUIDs as specified in **RFC 4122**.

If all you want is a unique ID, you should probably call *uuid1()* or *uuid4()*. Note that *uuid1()* may compromise privacy since it creates a UUID containing the computer's network address. *uuid4()* creates a random UUID.

class uuid.**UUID**([*hex*[, *bytes*[, *bytes_le*[, *fields*[, *int*[, *version*]]]]])

Create a UUID from either a string of 32 hexadecimal digits, a string of 16 bytes in big-endian order as the *bytes* argument, a string of 16 bytes in little-endian order as the *bytes_le* argument, a tuple of six integers (32-bit *time_low*, 16-bit *time_mid*, 16-bit *time_hi_version*, 8-bit *clock_seq_hi_variant*, 8-bit *clock_seq_low*, 48-bit *node*) as the *fields* argument, or a single 128-bit integer as the *int* argument. When a string of hex digits is given, curly braces, hyphens, and a URN prefix are all optional. For example, these expressions all yield the same UUID:

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes='\x12\x34\x56\x78'*4)
UUID(bytes_le='\x78\x56\x34\x12\x34\x12\x78\x56' +
              '\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

Exactly one of *hex*, *bytes*, *bytes_le*, *fields*, or *int* must be given. The *version* argument is optional; if given, the resulting UUID will have its variant and version number set according to RFC 4122, overriding bits in the given *hex*, *bytes*, *bytes_le*, *fields*, or *int*.

UUID instances have these read-only attributes:

UUID.**bytes**
The UUID as a 16-byte string (containing the six integer fields in big-endian byte order).

UUID.**bytes_le**
The UUID as a 16-byte string (with *time_low*, *time_mid*, and *time_hi_version* in little-endian byte order).

UUID.**fields**
以元组形式存放的 UUID 的 6 个整数域，有六个单独的属性和两个派生属性：

域	含义
time_low	UUID 的前 32 位
time_mid	接前一域的 16 位
time_hi_version	接前一域的 16 位
clock_seq_hi_variant	接前一域的 8 位
clock_seq_low	接前一域的 8 位
node	UUID 的最后 48 位
time	UUID 的总长 60 位的时间戳
clock_seq	14 位的序列号

UUID.**hex**
The UUID as a 32-character hexadecimal string.

UUID.**int**
The UUID as a 128-bit integer.

UUID.**urn**
The UUID as a URN as specified in RFC 4122.

UUID.**variant**
The UUID variant, which determines the internal layout of the UUID. This will be one of the constants *RESERVED_NCS*, *RFC_4122*, *RESERVED_MICROSOFT*, or *RESERVED_FUTURE*.

UUID.**version**
The UUID version number (1 through 5, meaningful only when the variant is *RFC_4122*).

The *uuid* module defines the following functions:

uuid.**getnode**()
Get the hardware address as a 48-bit positive integer. The first time this runs, it may launch a separate program, which could be quite slow. If all attempts to obtain the hardware address fail, we choose a random 48-bit number with its eighth bit set to 1 as recommended in RFC 4122. "Hardware address" means the MAC address of a network interface, and on a machine with multiple network interfaces the MAC address of any one of them may be returned.

uuid.**uuid1**([*node*[, *clock_seq*]])
Generate a UUID from a host ID, sequence number, and the current time. If *node* is not given, *getnode()* is used to obtain the hardware address. If *clock_seq* is given, it is used as the sequence number; otherwise a random 14-bit sequence number is chosen.

uuid.**uuid3**(*namespace*, *name*)
Generate a UUID based on the MD5 hash of a namespace identifier (which is a UUID) and a name (which is a string).

uuid.**uuid4**()
Generate a random UUID.

uuid.**uuid5**(*namespace*, *name*)
Generate a UUID based on the SHA-1 hash of a namespace identifier (which is a UUID) and a name (which is a string).

The *uuid* module defines the following namespace identifiers for use with *uuid3()* or *uuid5()*.

uuid.**NAMESPACE_DNS**
When this namespace is specified, the *name* string is a fully-qualified domain name.

uuid.**NAMESPACE_URL**
When this namespace is specified, the *name* string is a URL.

uuid.**NAMESPACE_OID**
When this namespace is specified, the *name* string is an ISO OID.

uuid.**NAMESPACE_X500**
When this namespace is specified, the *name* string is an X.500 DN in DER or a text output format.

The *uuid* module defines the following constants for the possible values of the `variant` attribute:

uuid.**RESERVED_NCS**
Reserved for NCS compatibility.

uuid.**RFC_4122**
Specifies the UUID layout given in **RFC 4122**.

uuid.**RESERVED_MICROSOFT**
Reserved for Microsoft compatibility.

uuid.**RESERVED_FUTURE**
Reserved for future definition.

参见:

RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace This specification defines a Uniform Resource Name namespace for UUIDs, the internal format of UUIDs, and methods of generating UUIDs.

20.15.1 示例

Here are some examples of typical usage of the *uuid* module:

```
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
```

(下页继续)

(续上页)

```
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```

20.16 urlparse —Parse URLs into components

注解: The *urlparse* module is renamed to `urllib.parse` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/urlparse.py

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a "relative URL" to an absolute URL given a "base URL."

The module has been designed to match the Internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `mailto`, `mms`, `news`, `nntp`, `prospero`, `rsync`, `rtsp`, `rtspu`, `sftp`, `shttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`.

2.5 新版功能: Support for the `sftp` and `sips` schemes.

The *urlparse* module defines the following functions:

`urlparse.`**`urlparse`**(*urlstring*[, *scheme*[, *allow_fragments*]])

Parse a URL into six components, returning a 6-tuple. This corresponds to the general structure of a URL: `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the *path* component, which is retained if present. For example:

```
>>> from urlparse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> o.scheme
'http'
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```

Following the syntax specifications in **RFC 1808**, urlparse recognizes a netloc only if it is properly introduced by '//'. Otherwise the input is presumed to be a relative URL and thus to start with a path component.

```
>>> from urlparse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
            query='', fragment='')
```

If the *scheme* argument is specified, it gives the default addressing scheme, to be used only if the URL does not specify one. The default value for this argument is the empty string.

If the *allow_fragments* argument is false, fragment identifiers are not recognized and parsed as part of the preceding component, even if the URL' s addressing scheme normally does support them. The default value for this argument is `True`.

The return value is actually an instance of a subclass of `tuple`. This class has the following additional read-only convenience attributes:

Attribute	Index	Value	Value if not present
`scheme`	0	URL scheme specifier	*scheme* parameter
`netloc`	1	Network location part	empty string
`path`	2	Hierarchical path	empty string
`params`	3	Parameters for last path element	empty string
`query`	4	Query component	empty string
`fragment`	5	Fragment identifier	empty string
`username`		User name	`None`
`password`		Password	`None`
`hostname`		Host name (lower case)	`None`
`port`		Port number as integer, if present	`None`

See section *Results of urlparse() and urlsplit()* for more information on the result object.

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a `ValueError`. If the URL is decomposed before parsing, or is not a Unicode string, no error will be raised.

在 2.5 版更改: Added attributes to return value.

在 2.7 版更改: Added IPv6 URL parsing capabilities.

在 2.7.17 版更改: Characters that affect netloc parsing under NFKC normalization will now raise `ValueError`.

`urlparse.`**`parse_qs`**(*qs*[, *keep_blank_values*[, *strict_parsing*[, *max_num_fields*]]])

Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a `ValueError` if there are more than *max_num_fields* fields read.

Use the `urllib.urlencode()` function to convert such dictionaries into query strings.

2.6 新版功能: Copied from the `cgi` module.

在 2.7.16 版更改: Added *max_num_fields* parameter.

`urlparse.`**`parse_qsl`**(*qs*[, *keep_blank_values*[, *strict_parsing*[, *max_num_fields*]]])

Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a list of name, value pairs.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a `ValueError` if there are more than *max_num_fields* fields read.

Use the `urllib.urlencode()` function to convert such lists of pairs into query strings.

2.6 新版功能: Copied from the `cgi` module.

在 2.7.16 版更改: Added *max_num_fields* parameter.

`urlparse.`**`urlunparse`**(*parts*)

Construct a URL from a tuple as returned by `urlparse()`. The *parts* argument can be any six-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

`urlparse.`**`urlsplit`**(*urlstring*[, *scheme*[, *allow_fragments*]])

This is similar to `urlparse()`, but does not split the params from the URL. This should generally be used instead of `urlparse()` if the more recent URL syntax allowing parameters to be applied to each segment of the *path* portion of the URL (see **RFC 2396**) is wanted. A separate function is needed to separate the path segments and parameters. This function returns a 5-tuple: (addressing scheme, network location, path, query, fragment identifier).

The return value is actually an instance of a subclass of `tuple`. This class has the following additional read-only convenience attributes:

Attribute	Index	Value	Value if not present
`scheme`	0	URL scheme specifier	*scheme* parameter
`netloc`	1	Network location part	empty string
`path`	2	Hierarchical path	empty string
`query`	3	Query component	empty string
`fragment`	4	Fragment identifier	empty string
`username`		User name	`None`
`password`		Password	`None`
`hostname`		Host name (lower case)	`None`
`port`		Port number as integer, if present	`None`

See section *Results of urlparse() and urlsplit()* for more information on the result object.

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a `ValueError`. If the URL is decomposed before parsing, or is not a Unicode string, no error will be raised.

2.2 新版功能.

在 2.5 版更改: Added attributes to return value.

在 2.7.17 版更改: Characters that affect netloc parsing under NFKC normalization will now raise *ValueError*.

urlparse.**urlunsplit**(*parts*)
Combine the elements of a tuple as returned by *urlsplit()* into a complete URL as a string. The *parts* argument can be any five-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

2.2 新版功能.

urlparse.**urljoin**(*base*, *url*[, *allow_fragments*])
Construct a full ("absolute") URL by combining a "base URL" (*base*) with another URL (*url*). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example:

```
>>> from urlparse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The *allow_fragments* argument has the same meaning and default as for *urlparse()*.

注解: If *url* is an absolute URL (that is, starting with `//` or `scheme://`), the *url*'s host name and/or scheme will be present in the result. For example:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

If you do not want that behavior, preprocess the *url* with *urlsplit()* and *urlunsplit()*, removing possible *scheme* and *netloc* parts.

urlparse.**urldefrag**(*url*)
If *url* contains a fragment identifier, returns a modified version of *url* with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in *url*, returns *url* unmodified and an empty string.

参见:

RFC 3986 - Uniform Resource Identifiers This is the current standard (STD66). Any changes to urlparse module should conform to this. Certain deviations could be observed, which are mostly for backward compatibility purposes and for certain de-facto parsing requirements as commonly observed in major browsers.

RFC 2732 - Format for Literal IPv6 Addresses in URL's. This specifies the parsing requirements of IPv6 URLs.

RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

RFC 2368 - The mailto URL scheme. Parsing requirements for mailto URL schemes.

RFC 1808 - Relative Uniform Resource Locators This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of "Abnormal Examples" which govern the treatment of border cases.

RFC 1738 - Uniform Resource Locators (URL) This specifies the formal syntax and semantics of absolute URLs.

20.16.1 Results of `urlparse()` and `urlsplit()`

The result objects from the *urlparse()* and *urlsplit()* functions are subclasses of the *tuple* type. These subclasses add the attributes described in those functions, as well as provide an additional method:

ParseResult.**geturl**()
Return the re-combined version of the original URL as a string. This may differ from the original URL in that the scheme will always be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed.

The result of this method is a fixpoint if passed back through the original parsing function:

```
>>> import urlparse
>>> url = 'HTTP://www.Python.org/doc/#'
```

```
>>> r1 = urlparse.urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
```

```
>>> r2 = urlparse.urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

2.5 新版功能.

The following classes provide the implementations of the parse results:

class urlparse.**ParseResult**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)
Concrete class for *urlparse()* results.

class urlparse.**SplitResult**(*scheme*, *netloc*, *path*, *query*, *fragment*)
Concrete class for *urlsplit()* results.

20.17 SocketServer —A framework for network servers

注解: The *SocketServer* module has been renamed to socketserver in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/SocketServer.py

The *SocketServer* module simplifies the task of writing network servers.

There are four basic concrete server classes:

class SocketServer.**TCPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
This uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. If *bind_and_activate* is true, the constructor automatically attempts to invoke *server_bind()* and *server_activate()*. The other parameters are passed to the *BaseServer* base class.

class SocketServer.**UDPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for *TCPServer*.

class SocketServer.**UnixStreamServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

class SocketServer.**UnixDatagramServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they're not available on non-Unix platforms. The parameters are the same as for *TCPServer*.

These four classes process requests *synchronously*; each request must be completed before the next request can be started. This isn't suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the *ForkingMixIn* and *ThreadingMixIn* mix-in classes can be used to support asynchronous behaviour.

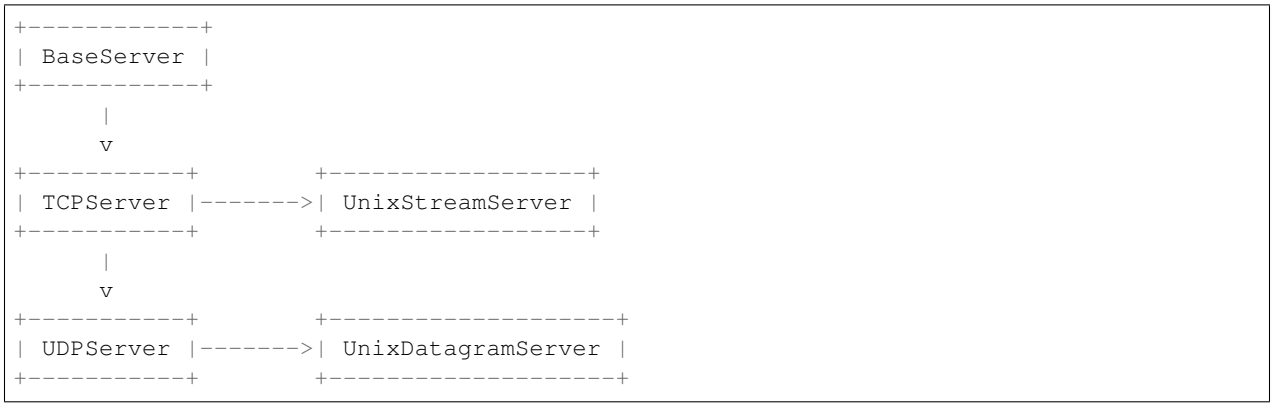
Creating a server requires several steps. First, you must create a request handler class by subclassing the *BaseRequestHandler* class and overriding its *handle()* method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server's address and the request handler class. Then call the *handle_request()* or *serve_forever()* method of the server object to process one or many requests. Finally, call *server_close()* to close the socket.

When inheriting from *ThreadingMixIn* for threaded connection behavior, you should explicitly declare how you want your threads to behave on an abrupt shutdown. The *ThreadingMixIn* class defines an attribute *daemon_threads*, which indicates whether or not the server should wait for thread termination. You should set the flag explicitly if you would like threads to behave autonomously; the default is *False*, meaning that Python will not exit until all threads created by *ThreadingMixIn* have exited.

Server classes have the same external methods and attributes, no matter what network protocol they use.

20.17.1 Server Creation Notes

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types:



Note that *UnixDatagramServer* derives from *UDPServer*, not from *UnixStreamServer* —the only difference between an IP and a Unix stream server is the address family, which is simply repeated in both Unix server classes.

class SocketServer.**ForkingMixIn**

class SocketServer.**ThreadingMixIn**

Forking and threading versions of each type of server can be created using these mix-in classes. For instance, *ThreadingUDPServer* is created as follows:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

The mix-in class comes first, since it overrides a method defined in *UDPServer*. Setting the various attributes also changes the behavior of the underlying server mechanism.

ForkingMixIn and the Forking classes mentioned below are only available on POSIX platforms that support *fork()*.

class SocketServer.**ForkingTCPServer**
class SocketServer.**ForkingUDPServer**
class SocketServer.**ThreadingTCPServer**
class SocketServer.**ThreadingUDPServer**
These classes are pre-defined using the mix-in classes.

To implement a service, you must derive a class from *BaseRequestHandler* and redefine its *handle()* method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses *StreamRequestHandler* or *DatagramRequestHandler*.

Of course, you still have to use your head! For instance, it makes no sense to use a forking server if the service contains state in memory that can be modified by different requests, since the modifications in the child process would never reach the initial state kept in the parent process and passed to each child. In this case, you can use a threading server, but you will probably have to use locks to protect the integrity of the shared data.

On the other hand, if you are building an HTTP server where all data is stored externally (for instance, in the file system), a synchronous class will essentially render the service "deaf" while one request is being handled –which may be for a very long time if a client is slow to receive all the data it has requested. Here a threading or forking server is appropriate.

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a synchronous server and doing an explicit fork in the request handler class *handle()* method.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor *fork()* (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use *select()* to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used). See *asyncore* for another way to manage this.

20.17.2 Server Objects

class SocketServer.**BaseServer**(*server_address*, *RequestHandlerClass*)
This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses. The two parameters are stored in the respective *server_address* and *RequestHandlerClass* attributes.

fileno()
Return an integer file descriptor for the socket on which the server is listening. This function is most commonly passed to *select.select()*, to allow monitoring multiple servers in the same process.

handle_request()
Process a single request. This function calls the following methods in order: *get_request()*, *verify_request()*, and *process_request()*. If the user-provided *handle()* method of the handler class raises an exception, the server's *handle_error()* method will be called. If no request is received within *timeout* seconds, *handle_timeout()* will be called and *handle_request()* will return.

serve_forever(*poll_interval=0.5*)
Handle requests until an explicit *shutdown()* request. Poll for shutdown every *poll_interval* seconds. Ignores the *timeout* attribute. If you need to do periodic tasks, do them in another thread.

shutdown()
Tell the *serve_forever()* loop to stop and wait until it does.

2.6 新版功能.

server_close()
Clean up the server. May be overridden.

2.6 新版功能.

address_family
The family of protocols to which the server's socket belongs. Common examples are *socket.AF_INET* and *socket.AF_UNIX*.

RequestHandlerClass
The user-provided request handler class; an instance of this class is created for each request.

server_address
The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the *socket* module for details. For Internet protocols, this is a tuple containing a string giving the address, and an integer port number: `('127.0.0.1', 80)`, for example.

socket
The socket object on which the server will listen for incoming requests.

The server classes support the following class variables:

allow_reuse_address
Whether the server will allow the reuse of an address. This defaults to *False*, and can be set in subclasses to change the policy.

request_queue_size
The size of the request queue. If it takes a long time to process a single request, any requests that arrive while the server is busy are placed into a queue, up to *request_queue_size* requests. Once the queue is full, further requests from clients will get a "Connection denied" error. The default value is usually 5, but this can be overridden by subclasses.

socket_type
The type of socket used by the server; *socket.SOCK_STREAM* and *socket.SOCK_DGRAM* are two common values.

timeout
Timeout duration, measured in seconds, or *None* if no timeout is desired. If *handle_request()* receives no incoming requests within the timeout period, the *handle_timeout()* method is called.

There are various server methods that can be overridden by subclasses of base server classes like *TCPServer*; these methods aren't useful to external users of the server object.

finish_request(*request*, *client_address*)
Actually processes the request by instantiating *RequestHandlerClass* and calling its *handle()* method.

get_request()
Must accept a request from the socket, and return a 2-tuple containing the *new* socket object to be used to communicate with the client, and the client's address.

handle_error(*request*, *client_address*)
This function is called if the *handle()* method of a *RequestHandlerClass* instance raises an exception. The default action is to print the traceback to standard output and continue handling further requests.

handle_timeout()
This function is called when the *timeout* attribute has been set to a value other than *None* and the timeout period has passed with no requests being received. The default action for forking servers is to collect the status of any child processes that have exited, while in threading servers this method does nothing.

process_request(*request*, *client_address*)
Calls *finish_request()* to create an instance of the *RequestHandlerClass*. If desired, this function can create a new process or thread to handle the request; the *ForkingMixIn* and *ThreadingMixIn* classes do this.

server_activate()
Called by the server's constructor to activate the server. The default behavior for a TCP server just invokes *listen()* on the server's socket. May be overridden.

server_bind()
Called by the server's constructor to bind the socket to the desired address. May be overridden.

verify_request(*request*, *client_address*)
Must return a Boolean value; if the value is *True*, the request will be processed, and if it's *False*, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always returns *True*.

20.17.3 Request Handler Objects

class SocketServer.**BaseRequestHandler**
This is the superclass of all request handler objects. It defines the interface, given below. A concrete request handler subclass must define a new *handle()* method, and can override any of the other methods. A new instance of the subclass is created for each request.

setup()
Called before the *handle()* method to perform any initialization actions required. The default implementation does nothing.

handle()
This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it; the request is available as `self.request`; the client address as `self.client_address`; and the server instance as `self.server`, in case it needs access to per-server information.

The type of `self.request` is different for datagram or stream services. For stream services, `self.request` is a socket object; for datagram services, `self.request` is a pair of string and socket.

finish()
Called after the *handle()* method to perform any clean-up actions required. The default implementation does nothing. If *setup()* raises an exception, this function will not be called.

class SocketServer.**StreamRequestHandler**
class SocketServer.**DatagramRequestHandler**
These *BaseRequestHandler* subclasses override the *setup()* and *finish()* methods, and provide `self.rfile` and `self.wfile` attributes. The `self.rfile` and `self.wfile` attributes can be read or written, respectively, to get the request data or return data to the client.

20.17.4 例子

SocketServer.TCPServer Example

This is the server side:

```
import SocketServer

class MyTCPHandler(SocketServer.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print "{} wrote:".format(self.client_address[0])
        print self.data
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    server = SocketServer.TCPServer((HOST, PORT), MyTCPHandler)

    # Activate the server; this will keep running until you
    # interrupt the program with Ctrl-C
    server.serve_forever()
```

An alternative request handler class that makes use of streams (file-like objects that simplify communication by providing the standard file interface):

```
class MyTCPHandler(SocketServer.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the handler;
        # we can now use e.g. readline() instead of raw recv() calls
        self.data = self.rfile.readline().strip()
        print "{} wrote:".format(self.client_address[0])
        print self.data
        # Likewise, self.wfile is a file-like object used to write back
        # to the client
        self.wfile.write(self.data.upper())
```

The difference is that the `readline()` call in the second handler will call `recv()` multiple times until it encounters a newline character, while the single `recv()` call in the first handler will just return what has been sent from the client in one `sendall()` call.

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(data + "\n")

    # Receive data from the server and shut down
    received = sock.recv(1024)
finally:
    sock.close()

print "Sent:     {}".format(data)
print "Received: {}".format(received)
```

The output of the example should look something like this:

Server:

```
$ python TCPServer.py
127.0.0.1 wrote:
hello world with TCP
127.0.0.1 wrote:
python is nice
```

Client:

```
$ python TCPClient.py hello world with TCP
Sent:     hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent:     python is nice
Received: PYTHON IS NICE
```

SocketServer.UDPServer Example

This is the server side:

```
import SocketServer

class MyUDPHandler(SocketServer.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
    """

    def handle(self):
```

(下页继续)

(续上页)

```
        data = self.request[0].strip()
        socket = self.request[1]
        print "{} wrote:".format(self.client_address[0])
        print data
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    server = SocketServer.UDPServer((HOST, PORT), MyUDPHandler)
    server.serve_forever()
```

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(data + "\n", (HOST, PORT))
received = sock.recv(1024)

print "Sent:     {}".format(data)
print "Received: {}".format(received)
```

The output of the example should look exactly like for the TCP server example.

Asynchronous Mixins

To build asynchronous handlers, use the *ThreadingMixIn* and *ForkingMixIn* classes.

An example for the *ThreadingMixIn* class:

```
import socket
import threading
import SocketServer

class ThreadedTCPRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
        data = self.request.recv(1024)
        cur_thread = threading.current_thread()
        response = "{}: {}".format(cur_thread.name, data)
        self.request.sendall(response)

class ThreadedTCPServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    pass

def client(ip, port, message):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((ip, port))
```

(下页继续)

(续上页)

```
    try:
        sock.sendall(message)
        response = sock.recv(1024)
        print "Received: {}".format(response)
    finally:
        sock.close()

if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    ip, port = server.server_address

    # Start a thread with the server -- that thread will then start one
    # more thread for each request
    server_thread = threading.Thread(target=server.serve_forever)
    # Exit the server thread when the main thread terminates
    server_thread.daemon = True
    server_thread.start()
    print "Server loop running in thread:", server_thread.name

    client(ip, port, "Hello World 1")
    client(ip, port, "Hello World 2")
    client(ip, port, "Hello World 3")

    server.shutdown()
    server.server_close()
```

The output of the example should look something like this:

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

The *ForkingMixIn* class is used in the same way, except that the server will spawn a new process for each request. Available only on POSIX platforms that support *fork()*.

20.18 BaseHTTPServer —Basic HTTP server

注解: The *BaseHTTPServer* module has been merged into `http.server` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/BaseHTTPServer.py

This module defines two classes for implementing HTTP servers (Web servers). Usually, this module isn't used directly, but is used as a basis for building functioning Web servers. See the *SimpleHTTPServer* and *CGIHTTPServer* modules.

The first class, *HTTPServer*, is a *SocketServer.TCPServer* subclass, and therefore implements the *SocketServer.BaseServer* interface. It creates and listens at the HTTP socket, dispatching the requests to a handler. Code to create and run the server looks like this:

```
def run(server_class=BaseHTTPServer.HTTPServer,
        handler_class=BaseHTTPServer.BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class BaseHTTPServer.**HTTPServer**(*server_address*, *RequestHandlerClass*)

This class builds on the TCPServer class by storing the server address as instance variables named server_name and server_port. The server is accessible by the handler, typically through the handler's server instance variable.

class BaseHTTPServer.**BaseHTTPRequestHandler**(*request*, *client_address*, *server*)

This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). *BaseHTTPRequestHandler* provides a number of class and instance variables, and methods for use by subclasses.

The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method SPAM, the do_SPAM() method will be called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to override or extend the __init__() method.

BaseHTTPRequestHandler has the following instance variables:

client_address
Contains a tuple of the form (host, port) referring to the client's address.

server
Contains the server instance.

command
Contains the command (request type). For example, 'GET'.

path
Contains the request path.

request_version
Contains the version string from the request. For example, 'HTTP/1.0'.

headers
Holds an instance of the class specified by the *MessageClass* class variable. This instance parses and manages the headers in the HTTP request.

rfile
Contains an input stream, positioned at the start of the optional input data.

wfile
Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream.

BaseHTTPRequestHandler has the following class variables:

server_version
Specifies the server software version. You may want to override this. The format is multiple whitespace-separated strings, where each string is of the form name[/version]. For example, 'BaseHTTP/0.2'.

sys_version
Contains the Python system version, in a form usable by the *version_string* method and the *server_version* class variable. For example, `'Python/1.4'`.

error_message_format
Specifies a format string for building an error response to the client. It uses parenthesized, keyed format specifiers, so the format operand must be a dictionary. The *code* key should be an integer, specifying the numeric HTTP error code value. *message* should be a string containing a (detailed) error message of what occurred, and *explain* should be an explanation of the error code number. Default *message* and *explain* values can found in the *responses* class variable.

error_content_type
Specifies the Content-Type HTTP header of error responses sent to the client. The default value is `'text/html'`.

2.6 新版功能: Previously, the content type was always `'text/html'`.

protocol_version
This specifies the HTTP protocol version used in responses. If set to `'HTTP/1.1'`, the server will permit HTTP persistent connections; however, your server *must* then include an accurate `Content-Length` header (using *send_header()*) in all of its responses to clients. For backwards compatibility, the setting defaults to `'HTTP/1.0'`.

MessageClass
Specifies a *rfc822.Message*-like class to parse HTTP headers. Typically, this is not overridden, and it defaults to *mimetools.Message*.

responses
This variable contains a mapping of error code integers to two-element tuples containing a short and long message. For example, `{code: (shortmessage, longmessage)}`. The *shortmessage* is usually used as the *message* key in an error response, and *longmessage* as the *explain* key (see the *error_message_format* class variable).

A *BaseHTTPRequestHandler* instance has the following methods:

handle()
Calls *handle_one_request()* once (or, if persistent connections are enabled, multiple times) to handle incoming HTTP requests. You should never need to override it; instead, implement appropriate `do_*()` methods.

handle_one_request()
This method will parse and dispatch the request to the appropriate `do_*()` method. You should never need to override it.

send_error(*code*[, *message*])
Sends and logs a complete error reply to the client. The numeric *code* specifies the HTTP error code, with *message* as optional, more specific text. A complete set of headers is sent, followed by text composed using the *error_message_format* class variable. The body will be empty if the method is HEAD or the response code is one of the following: `1xx`, `204 No Content`, `205 Reset Content`, `304 Not Modified`.

send_response(*code*[, *message*])
Sends a response header and logs the accepted request. The HTTP response line is sent, followed by *Server* and *Date* headers. The values for these two headers are picked up from the *version_string()* and *date_time_string()* methods, respectively.

send_header(*keyword*, *value*)
Writes a specific HTTP header to the output stream. *keyword* should specify the header keyword, with *value* specifying its value.

end_headers()
Sends a blank line, indicating the end of the HTTP headers in the response.

log_request([*code*[, *size*]])
Logs an accepted (successful) request. *code* should specify the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the *size* parameter.

log_error(...)
Logs an error when a request cannot be fulfilled. By default, it passes the message to *log_message()*, so it takes the same arguments (*format* and additional values).

log_message(*format*, ...)
Logs an arbitrary message to `sys.stderr`. This is typically overridden to create custom error logging mechanisms. The *format* argument is a standard printf-style format string, where the additional arguments to *log_message()* are applied as inputs to the formatting. The client ip address and current date and time are prefixed to every message logged.

version_string()
Returns the server software's version string. This is a combination of the *server_version* and *sys_version* class variables.

date_time_string([*timestamp*])
Returns the date and time given by *timestamp* (which must be in the format returned by *time.time()*), formatted for a message header. If *timestamp* is omitted, it uses the current date and time.

The result looks like `'Sun, 06 Nov 1994 08:49:37 GMT'`.

2.5 新版功能: The *timestamp* parameter.

log_date_time_string()
Returns the current date and time, formatted for logging.

address_string()
Returns the client address, formatted for logging. A name lookup is performed on the client's IP address.

20.18.1 More examples

To create a server that doesn't run forever, but until some condition is fulfilled:

```
def run_while_true(server_class=BaseHTTPServer.HTTPServer,
                   handler_class=BaseHTTPServer.BaseHTTPRequestHandler):
    """
    This assumes that keep_running() is a function of no arguments which
    is tested initially and after each request.  If its return value
    is true, the server continues.
    """
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    while keep_running():
        httpd.handle_request()
```

参见:

Module ***CGIHTTPServer*** Extended request handler that supports CGI scripts.

Module ***SimpleHTTPServer*** Basic request handler that limits response to files actually under the document root.

20.19 SimpleHTTPServer —Simple HTTP request handler

注解: The *SimpleHTTPServer* module has been merged into `http.server` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

警告: *SimpleHTTPServer* is not recommended for production. It only implements basic security checks.

The *SimpleHTTPServer* module defines a single class, *SimpleHTTPRequestHandler*, which is interface-compatible with *BaseHTTPServer.BaseHTTPRequestHandler*.

The *SimpleHTTPServer* module defines the following class:

class SimpleHTTPServer.**SimpleHTTPRequestHandler**(*request*, *client_address*, *server*)

This class serves files from the current directory and below, directly mapping the directory structure to HTTP requests.

A lot of the work, such as parsing the request, is done by the base class *BaseHTTPServer.BaseHTTPRequestHandler*. This class implements the *do_GET()* and *do_HEAD()* functions.

The following are defined as class-level attributes of *SimpleHTTPRequestHandler*:

server_version

This will be `"SimpleHTTP/" + __version__`, where `__version__` is defined at the module level.

extensions_map

A dictionary mapping suffixes into MIME types. The default is signified by an empty string, and is considered to be `application/octet-stream`. The mapping is used case-insensitively, and so should contain only lower-cased keys.

The *SimpleHTTPRequestHandler* class defines the following methods:

do_HEAD()

This method serves the `'HEAD'` request type: it sends the headers it would send for the equivalent `GET` request. See the *do_GET()* method for a more complete explanation of the possible headers.

do_GET()

The request is mapped to a local file by interpreting the request as a path relative to the current working directory.

If the request was mapped to a directory, the directory is checked for a file named `index.html` or `index.htm` (in that order). If found, the file's contents are returned; otherwise a directory listing is generated by calling the `list_directory()` method. This method uses *os.listdir()* to scan the directory, and returns a `404` error response if the `listdir()` fails.

If the request was mapped to a file, it is opened and the contents are returned. Any *IOError* exception in opening the requested file is mapped to a `404`, `'File not found'` error. Otherwise, the content type is guessed by calling the `guess_type()` method, which in turn uses the *extensions_map* variable.

A `'Content-type:'` header with the guessed content type is output, followed by a `'Content-Length:'` header with the file's size and a `'Last-Modified:'` header with the file's modification time.

Then follows a blank line signifying the end of the headers, and then the contents of the file are output. If the file's MIME type starts with `text/` the file is opened in text mode; otherwise binary mode is used.

The *test()* function in the *SimpleHTTPServer* module is an example which creates a server using the *SimpleHTTPRequestHandler* as the Handler.

2.5 新版功能: The 'Last-Modified' header.

The *SimpleHTTPServer* module can be used in the following manner in order to set up a very basic web server serving files relative to the current directory.

```
import SimpleHTTPServer
import SocketServer

PORT = 8000

Handler = SimpleHTTPServer.SimpleHTTPRequestHandler

httpd = SocketServer.TCPServer(("", PORT), Handler)

print "serving at port", PORT
httpd.serve_forever()
```

The *SimpleHTTPServer* module can also be invoked directly using the `-m` switch of the interpreter with a `port number` argument. Similar to the previous example, this serves the files relative to the current directory.

```
python -m SimpleHTTPServer 8000
```

参见:

Module *BaseHTTPServer* Base class implementation for Web server and request handler.

20.20 CGIHTTPServer —CGI-capable HTTP request handler

注解: The *CGIHTTPServer* module has been merged into `http.server` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

The *CGIHTTPServer* module defines a request-handler class, interface compatible with *BaseHTTPServer.BaseHTTPRequestHandler* and inherits behavior from *SimpleHTTPServer.SimpleHTTPRequestHandler* but can also run CGI scripts.

注解: This module can run CGI scripts on Unix and Windows systems.

注解: CGI scripts run by the *CGIHTTPRequestHandler* class cannot execute redirects (HTTP code 302), because code 200 (script output follows) is sent prior to execution of the CGI script. This pre-empts the status code.

The *CGIHTTPServer* module defines the following class:

class CGIHTTPServer.**CGIHTTPRequestHandler**(*request*, *client_address*, *server*)

This class is used to serve either files or output of CGI scripts from the current directory and below. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in *SimpleHTTPServer.SimpleHTTPRequestHandler*.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script. Only directory-based CGI are used —the other common server configuration is to treat special extensions as denoting

CGI scripts.

The `do_GET()` and `do_HEAD()` functions are modified to run CGI scripts and serve the output, instead of serving files, if the request leads to somewhere below the `cgi_directories` path.

The *CGIHTTPRequestHandler* defines the following data member:

cgi_directories
This defaults to `['/cgi-bin', '/htbin']` and describes directories to treat as containing CGI scripts.

The *CGIHTTPRequestHandler* defines the following methods:

do_POST()
This method serves the `'POST'` request type, only allowed for CGI scripts. Error 501, "Can only POST to CGI scripts", is output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI script will be translated to error 403.

For example usage, see the implementation of the *test()* function.

参见:

Module *BaseHTTPServer* Base class implementation for Web server and request handler.

20.21 cookielib —Cookie handling for HTTP clients

注解: The *cookielib* module has been renamed to `http.cookiejar` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

2.4 新版功能.

Source code: Lib/cookielib.py

The *cookielib* module defines classes for automatic handling of HTTP cookies. It is useful for accessing web sites that require small pieces of data –*cookies* –to be set on the client machine by an HTTP response from a web server, and then returned to the server in later HTTP requests.

Both the regular Netscape cookie protocol and the protocol defined by **RFC 2965** are handled. RFC 2965 handling is switched off by default. **RFC 2109** cookies are parsed as Netscape cookies and subsequently treated either as Netscape or RFC 2965 cookies according to the 'policy' in effect. Note that the great majority of cookies on the Internet are Netscape cookies. *cookielib* attempts to follow the de-facto Netscape cookie protocol (which differs substantially from that set out in the original Netscape specification), including taking note of the `max-age` and `port` cookie-attributes introduced with RFC 2965.

注解: The various named parameters found in *Set-Cookie* and *Set-Cookie2* headers (eg. `domain` and `expires`) are conventionally referred to as *attributes*. To distinguish them from Python attributes, the documentation for this module uses the term *cookie-attribute* instead.

The module defines the following exception:

exception cookielib.**LoadError**
Instances of *FileCookieJar* raise this exception on failure to load cookies from a file.

注解: For backwards-compatibility with Python 2.4 (which raised an `IOError`), `LoadError` is a subclass of `IOError`.

The following classes are provided:

class cookielib.**CookieJar**(*policy=None*)
policy is an object implementing the `CookiePolicy` interface.

The `CookieJar` class stores HTTP cookies. It extracts cookies from HTTP requests, and returns them in HTTP responses. `CookieJar` instances automatically expire contained cookies when necessary. Subclasses are also responsible for storing and retrieving cookies from a file or database.

class cookielib.**FileCookieJar**(*filename*, *delayload=None*, *policy=None*)
policy is an object implementing the `CookiePolicy` interface. For the other arguments, see the documentation for the corresponding attributes.

A `CookieJar` which can load cookies from, and perhaps save cookies to, a file on disk. Cookies are **NOT** loaded from the named file until either the `load()` or `revert()` method is called. Subclasses of this class are documented in section *FileCookieJar subclasses and co-operation with web browsers*.

class cookielib.**CookiePolicy**
This class is responsible for deciding whether each cookie should be accepted from / returned to the server.

class cookielib.**DefaultCookiePolicy**(*blocked_domains=None*, *allowed_domains=None*, *netscape=True*, *rfc2965=False*, *rfc2109_as_netscape=None*, *hide_cookie2=False*, *strict_domain=False*, *strict_rfc2965_unverifiable=True*, *strict_ns_unverifiable=False*, *strict_ns_domain=DefaultCookiePolicy.DomainLiberal*, *strict_ns_set_initial_dollar=False*, *strict_ns_set_path=False*)
Constructor arguments should be passed as keyword arguments only. *blocked_domains* is a sequence of domain names that we never accept cookies from, nor return cookies to. *allowed_domains* if not `None`, this is a sequence of the only domains for which we accept and return cookies. For all other arguments, see the documentation for `CookiePolicy` and `DefaultCookiePolicy` objects.

`DefaultCookiePolicy` implements the standard accept / reject rules for Netscape and RFC 2965 cookies. By default, RFC 2109 cookies (ie. cookies received in a `Set-Cookie` header with a version cookie-attribute of 1) are treated according to the RFC 2965 rules. However, if RFC 2965 handling is turned off or `rfc2109_as_netscape` is `True`, RFC 2109 cookies are 'downgraded' by the `CookieJar` instance to Netscape cookies, by setting the `version` attribute of the `Cookie` instance to 0. `DefaultCookiePolicy` also provides some parameters to allow some fine-tuning of policy.

class cookielib.**Cookie**
This class represents Netscape, RFC 2109 and RFC 2965 cookies. It is not expected that users of `cookielib` construct their own `Cookie` instances. Instead, if necessary, call `make_cookies()` on a `CookieJar` instance.

参见:

Module `urllib2` URL opening with automatic cookie handling.

Module `Cookie` HTTP cookie classes, principally useful for server-side code. The `cookielib` and `Cookie` modules do not depend on each other.

https://curl.haxx.se/rfc/cookie_spec.html The specification of the original Netscape cookie protocol. Though this is still the dominant protocol, the 'Netscape cookie protocol' implemented by all the major browsers (and `cookielib`) only bears a passing resemblance to the one sketched out in `cookie_spec.html`.

RFC 2109 - HTTP State Management Mechanism Obsoleted by RFC 2965. Uses *Set-Cookie* with version=1.

RFC 2965 - HTTP State Management Mechanism The Netscape protocol with the bugs fixed. Uses *Set-Cookie2* in place of *Set-Cookie*. Not widely used.

http://kristol.org/cookie/errata.html Unfinished errata to RFC 2965.

RFC 2964 - Use of HTTP State Management

20.21.1 CookieJar and FileCookieJar Objects

CookieJar objects support the *iterator* protocol for iterating over contained *Cookie* objects.

CookieJar has the following methods:

CookieJar.**add_cookie_header**(*request*)
Add correct *Cookie* header to *request*.

If policy allows (ie. the `rfc2965` and `hide_cookie2` attributes of the *CookieJar*'s *CookiePolicy* instance are true and false respectively), the *Cookie2* header is also added when appropriate.

The *request* object (usually a *urllib2.Request* instance) must support the methods `get_full_url()`, `get_host()`, `get_type()`, `unverifiable()`, `get_origin_req_host()`, `has_header()`, `get_header()`, `header_items()`, and `add_unredirected_header()`,as documented by *urllib2*.

CookieJar.**extract_cookies**(*response*, *request*)
Extract cookies from HTTP *response* and store them in the *CookieJar*, where allowed by policy.

The *CookieJar* will look for allowable *Set-Cookie* and *Set-Cookie2* headers in the *response* argument, and store cookies as appropriate (subject to the *CookiePolicy.set_ok()* method's approval).

The *response* object (usually the result of a call to *urllib2.urlopen()*, or similar) should support an `info()` method, which returns an object with a `getallmatchingheaders()` method (usually a *mimetools.Message* instance).

The *request* object (usually a *urllib2.Request* instance) must support the methods `get_full_url()`, `get_host()`, `unverifiable()`, and `get_origin_req_host()`, as documented by *urllib2*. The request is used to set default values for cookie-attributes as well as for checking that the cookie is allowed to be set.

CookieJar.**set_policy**(*policy*)
Set the *CookiePolicy* instance to be used.

CookieJar.**make_cookies**(*response*, *request*)
Return sequence of *Cookie* objects extracted from *response* object.

See the documentation for *extract_cookies()* for the interfaces required of the *response* and *request* arguments.

CookieJar.**set_cookie_if_ok**(*cookie*, *request*)
Set a *Cookie* if policy says it's OK to do so.

CookieJar.**set_cookie**(*cookie*)
Set a *Cookie*, without checking with policy to see whether or not it should be set.

CookieJar.**clear**([*domain*[, *path*[, *name*]]])
Clear some cookies.

If invoked without arguments, clear all cookies. If given a single argument, only cookies belonging to that *domain* will be removed. If given two arguments, cookies belonging to the specified *domain* and URL *path* are removed. If given three arguments, then the cookie with the specified *domain*, *path* and *name* is removed.

Raises *KeyError* if no matching cookie exists.

CookieJar.**clear_session_cookies**()
Discard all session cookies.

Discards all contained cookies that have a true `discard` attribute (usually because they had either no `max-age` or `expires` cookie-attribute, or an explicit `discard` cookie-attribute). For interactive browsers, the end of a session usually corresponds to closing the browser window.

Note that the `save()` method won't save session cookies anyway, unless you ask otherwise by passing a true *ignore_discard* argument.

FileCookieJar implements the following additional methods:

FileCookieJar.**save**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Save cookies to a file.

This base class raises *NotImplementedError*. Subclasses may leave this method unimplemented.

filename is the name of file in which to save cookies. If *filename* is not specified, `self.filename` is used (whose default is the value passed to the constructor, if any); if `self.filename` is *None*, *ValueError* is raised.

ignore_discard: save even cookies set to be discarded. *ignore_expires*: save even cookies that have expired

The file is overwritten if it already exists, thus wiping all the cookies it contains. Saved cookies can be restored later using the *load()* or *revert()* methods.

FileCookieJar.**load**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Load cookies from a file.

Old cookies are kept unless overwritten by newly loaded ones.

Arguments are as for *save()*.

The named file must be in the format understood by the class, or *LoadError* will be raised. Also, *IOError* may be raised, for example if the file does not exist.

注解： For backwards-compatibility with Python 2.4 (which raised an *IOError*), *LoadError* is a subclass of *IOError*.

FileCookieJar.**revert**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Clear all cookies and reload cookies from a saved file.

revert() can raise the same exceptions as *load()*. If there is a failure, the object's state will not be altered.

FileCookieJar instances have the following public attributes:

FileCookieJar.**filename**
Filename of default file in which to keep cookies. This attribute may be assigned to.

FileCookieJar.**delayload**
If true, load cookies lazily from disk. This attribute should not be assigned to. This is only a hint, since this only affects performance, not behaviour (unless the cookies on disk are changing). A *CookieJar* object may ignore it. None of the *FileCookieJar* classes included in the standard library lazily loads cookies.

20.21.2 FileCookieJar subclasses and co-operation with web browsers

The following *CookieJar* subclasses are provided for reading and writing.

class cookielib.**MozillaCookieJar**(*filename*, *delayload=None*, *policy=None*)
A *FileCookieJar* that can load from and save cookies to disk in the Mozilla `cookies.txt` file format (which is also used by the Lynx and Netscape browsers).

注解: Version 3 of the Firefox web browser no longer writes cookies in the `cookies.txt` file format.

注解: This loses information about RFC 2965 cookies, and also about newer or non-standard cookie-attributes such as `port`.

警告: Back up your cookies before saving if you have cookies whose loss / corruption would be inconvenient (there are some subtleties which may lead to slight changes in the file over a load / save round-trip).

Also note that cookies saved while Mozilla is running will get clobbered by Mozilla.

class cookielib.**LWPCookieJar**(*filename*, *delayload=None*, *policy=None*)
A *FileCookieJar* that can load from and save cookies to disk in format compatible with the libwww-perl library's `Set-Cookie3` file format. This is convenient if you want to store cookies in a human-readable file.

20.21.3 CookiePolicy Objects

Objects implementing the *CookiePolicy* interface have the following methods:

CookiePolicy.**set_ok**(*cookie*, *request*)
Return boolean value indicating whether cookie should be accepted from server.

cookie is a *cookielib.Cookie* instance. *request* is an object implementing the interface defined by the documentation for *CookieJar.extract_cookies()*.

CookiePolicy.**return_ok**(*cookie*, *request*)
Return boolean value indicating whether cookie should be returned to server.

cookie is a *cookielib.Cookie* instance. *request* is an object implementing the interface defined by the documentation for *CookieJar.add_cookie_header()*.

CookiePolicy.**domain_return_ok**(*domain*, *request*)
Return false if cookies should not be returned, given cookie domain.

This method is an optimization. It removes the need for checking every cookie with a particular domain (which might involve reading many files). Returning true from *domain_return_ok()* and *path_return_ok()* leaves all the work to *return_ok()*.

If *domain_return_ok()* returns true for the cookie domain, *path_return_ok()* is called for the cookie path. Otherwise, *path_return_ok()* and *return_ok()* are never called for that cookie domain. If *path_return_ok()* returns true, *return_ok()* is called with the *Cookie* object itself for a full check. Otherwise, *return_ok()* is never called for that cookie path.

Note that *domain_return_ok()* is called for every *cookie* domain, not just for the *request* domain. For example, the function might be called with both `".example.com"` and `"www.example.com"` if the request domain is `"www.example.com"`. The same goes for *path_return_ok()*.

The *request* argument is as documented for *return_ok()*.

CookiePolicy.**path_return_ok**(*path*, *request*)
Return false if cookies should not be returned, given cookie path.

See the documentation for *domain_return_ok()*.

In addition to implementing the methods above, implementations of the *CookiePolicy* interface must also supply the following attributes, indicating which protocols should be used, and how. All of these attributes may be assigned to.

CookiePolicy.**netscape**
Implement Netscape protocol.

CookiePolicy.**rfc2965**
Implement RFC 2965 protocol.

CookiePolicy.**hide_cookie2**
Don't add *Cookie2* header to requests (the presence of this header indicates to the server that we understand RFC 2965 cookies).

The most useful way to define a *CookiePolicy* class is by subclassing from *DefaultCookiePolicy* and overriding some or all of the methods above. *CookiePolicy* itself may be used as a 'null policy' to allow setting and receiving any and all cookies (this is unlikely to be useful).

20.21.4 DefaultCookiePolicy Objects

Implements the standard rules for accepting and returning cookies.

Both RFC 2965 and Netscape cookies are covered. RFC 2965 handling is switched off by default.

The easiest way to provide your own policy is to override this class and call its methods in your overridden implementations before adding your own additional checks:

```
import cookielib
class MyCookiePolicy(cookielib.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not cookielib.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

In addition to the features required to implement the *CookiePolicy* interface, this class allows you to block and allow domains from setting and receiving cookies. There are also some strictness switches that allow you to tighten up the rather loose Netscape protocol rules a little bit (at the cost of blocking some benign cookies).

A domain blacklist and whitelist is provided (both off by default). Only domains not in the blacklist and present in the whitelist (if the whitelist is active) participate in cookie setting and returning. Use the *blocked_domains* constructor argument, and `blocked_domains()` and `set_blocked_domains()` methods (and the corresponding argument and methods for *allowed_domains*). If you set a whitelist, you can turn it off again by setting it to *None*.

Domains in block or allow lists that do not start with a dot must equal the cookie domain to be matched. For example, `"example.com"` matches a blacklist entry of `"example.com"`, but `"www.example.com"` does not. Domains that do start with a dot are matched by more specific domains too. For example, both `"www.example.com"` and `"www.coyote.example.com"` match `".example.com"` (but `"example.com"` itself does not). IP addresses are an exception, and must match exactly. For example, if blocked_domains contains `"192.168.1.2"` and `".168.1.2"`, 192.168.1.2 is blocked, but 193.168.1.2 is not.

DefaultCookiePolicy implements the following additional methods:

DefaultCookiePolicy.**blocked_domains**()
Return the sequence of blocked domains (as a tuple).

DefaultCookiePolicy.**set_blocked_domains**(*blocked_domains*)
Set the sequence of blocked domains.

DefaultCookiePolicy.**is_blocked**(*domain*)
Return whether *domain* is on the blacklist for setting or receiving cookies.

DefaultCookiePolicy.**allowed_domains**()
Return *None*, or the sequence of allowed domains (as a tuple).

DefaultCookiePolicy.**set_allowed_domains**(*allowed_domains*)
Set the sequence of allowed domains, or *None*.

DefaultCookiePolicy.**is_not_allowed**(*domain*)
Return whether *domain* is not on the whitelist for setting or receiving cookies.

DefaultCookiePolicy instances have the following attributes, which are all initialised from the constructor arguments of the same name, and which may all be assigned to.

DefaultCookiePolicy.**rfc2109_as_netscape**
If true, request that the *CookieJar* instance downgrade RFC 2109 cookies (ie. cookies received in a *Set-Cookie* header with a version cookie-attribute of 1) to Netscape cookies by setting the version attribute of the *Cookie* instance to 0. The default value is *None*, in which case RFC 2109 cookies are downgraded if and only if RFC 2965 handling is turned off. Therefore, RFC 2109 cookies are downgraded by default.

2.5 新版功能.

General strictness switches:

DefaultCookiePolicy.**strict_domain**
Don't allow sites to set two-component domains with country-code top-level domains like `.co.uk`, `.gov.uk`, `.co.nz`.etc. This is far from perfect and isn't guaranteed to work!

RFC 2965 protocol strictness switches:

DefaultCookiePolicy.**strict_rfc2965_unverifiable**
Follow RFC 2965 rules on unverifiable transactions (usually, an unverifiable transaction is one resulting from a redirect or a request for an image hosted on another site). If this is false, cookies are *never* blocked on the basis of verifiability

Netscape protocol strictness switches:

DefaultCookiePolicy.**strict_ns_unverifiable**
Apply RFC 2965 rules on unverifiable transactions even to Netscape cookies.

DefaultCookiePolicy.**strict_ns_domain**
Flags indicating how strict to be with domain-matching rules for Netscape cookies. See below for acceptable values.

DefaultCookiePolicy.**strict_ns_set_initial_dollar**
Ignore cookies in Set-Cookie: headers that have names starting with `'$'`.

DefaultCookiePolicy.**strict_ns_set_path**
Don't allow setting cookies whose path doesn't path-match request URI.

`strict_ns_domain` is a collection of flags. Its value is constructed by or-ing together (for example, `DomainStrictNoDots|DomainStrictNonDomain` means both flags are set).

DefaultCookiePolicy.**DomainStrictNoDots**
When setting cookies, the 'host prefix' must not contain a dot (eg. `www.foo.bar.com` can't set a cookie for `.bar.com`, because `www.foo` contains a dot).

DefaultCookiePolicy.**DomainStrictNonDomain**
Cookies that did not explicitly specify a `domain` cookie-attribute can only be returned to a domain equal to the domain that set the cookie (eg. `spam.example.com` won't be returned cookies from `example.com` that had no `domain` cookie-attribute).

DefaultCookiePolicy.**DomainRFC2965Match**
When setting cookies, require a full RFC 2965 domain-match.

The following attributes are provided for convenience, and are the most useful combinations of the above flags:

DefaultCookiePolicy.**DomainLiberal**
Equivalent to 0 (ie. all of the above Netscape domain strictness flags switched off).

DefaultCookiePolicy.**DomainStrict**
Equivalent to `DomainStrictNoDots|DomainStrictNonDomain`.

20.21.5 Cookie Objects

Cookie instances have Python attributes roughly corresponding to the standard cookie-attributes specified in the various cookie standards. The correspondence is not one-to-one, because there are complicated rules for assigning default values, because the `max-age` and `expires` cookie-attributes contain equivalent information, and because RFC 2109 cookies may be 'downgraded' by *cookielib* from version 1 to version 0 (Netscape) cookies.

Assignment to these attributes should not be necessary other than in rare circumstances in a *CookiePolicy* method. The class does not enforce internal consistency, so you should know what you're doing if you do that.

Cookie.**version**
Integer or *None*. Netscape cookies have *version* 0. RFC 2965 and RFC 2109 cookies have a `version` cookie-attribute of 1. However, note that *cookielib* may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case *version* is 0.

Cookie.**name**
Cookie name (a string).

Cookie.**value**
Cookie value (a string), or *None*.

Cookie.**port**
String representing a port or a set of ports (eg. '80', or '80,8080'), or *None*.

Cookie.**path**
Cookie path (a string, eg. `'/acme/rocket_launchers'`).

Cookie.**secure**
`True` if cookie should only be returned over a secure connection.

Cookie.**expires**
Integer expiry date in seconds since epoch, or *None*. See also the *is_expired()* method.

Cookie.**discard**
`True` if this is a session cookie.

Cookie.**comment**
String comment from the server explaining the function of this cookie, or *None*.

Cookie.**comment_url**
URL linking to a comment from the server explaining the function of this cookie, or *None*.

Cookie.**rfc2109**
`True` if this cookie was received as an RFC 2109 cookie (ie. the cookie arrived in a *Set-Cookie* header, and

the value of the Version cookie-attribute in that header was 1). This attribute is provided because *cookielib* may ‘downgrade’ RFC 2109 cookies to Netscape cookies, in which case *version* is 0.

2.5 新版功能.

`Cookie.`**port_specified**
`True` if a port or set of ports was explicitly specified by the server (in the *Set-Cookie* / *Set-Cookie2* header).

`Cookie.`**domain_specified**
`True` if a domain was explicitly specified by the server.

`Cookie.`**domain_initial_dot**
`True` if the domain explicitly specified by the server began with a dot (`'.'`).

Cookies may have additional non-standard cookie-attributes. These may be accessed using the following methods:

`Cookie.`**has_nonstandard_attr**(*name*)
Return true if cookie has the named cookie-attribute.

`Cookie.`**get_nonstandard_attr**(*name*, *default=None*)
If cookie has the named cookie-attribute, return its value. Otherwise, return *default*.

`Cookie.`**set_nonstandard_attr**(*name*, *value*)
Set the value of the named cookie-attribute.

The *Cookie* class also defines the following method:

`Cookie.`**is_expired**([*now=None*])
`True` if cookie has passed the time at which the server requested it should expire. If *now* is given (in seconds since the epoch), return whether the cookie has expired at the specified time.

20.21.6 Examples

The first example shows the most common usage of *cookielib*:

```
import cookielib, urllib2
cj = cookielib.CookieJar()
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

This example illustrates how to open a URL using your Netscape, Mozilla, or Lynx cookies (assumes Unix/Netscape convention for location of the cookies file):

```
import os, cookielib, urllib2
cj = cookielib.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

The next example illustrates the use of *DefaultCookiePolicy*. Turn on RFC 2965 cookies, be more strict about domains when setting and returning Netscape cookies, and block some domains from setting cookies or having them returned:

```
import urllib2
from cookielib import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=DefaultCookiePolicy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
```

(下页继续)

(续上页)

```
cj = CookieJar(policy)
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

20.22 Cookie —HTTP state management

注解: The *Cookie* module has been renamed to `http.cookies` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

Source code: Lib/Cookie.py

The *Cookie* module defines classes for abstracting the concept of cookies, an HTTP state management mechanism. It supports both simple string-only cookies, and provides an abstraction for having any serializable data-type as cookie value.

The module formerly strictly applied the parsing rules described in the **RFC 2109** and **RFC 2068** specifications. It has since been discovered that MSIE 3.0x doesn't follow the character rules outlined in those specs and also many current day browsers and servers have relaxed parsing rules when comes to Cookie handling. As a result, the parsing rules used are a bit less strict.

The character set, *string.ascii_letters*, *string.digits* and `!#$%&'*+-.^_`|~` denote the set of valid characters allowed by this module in Cookie name (as *key*).

注解: On encountering an invalid cookie, *CookieError* is raised, so if your cookie data comes from a browser you should always prepare for invalid data and catch *CookieError* on parsing.

exception Cookie.**CookieError**
Exception failing because of **RFC 2109** invalidity: incorrect attributes, incorrect *Set-Cookie* header, etc.

class Cookie.**BaseCookie**([*input*])
This class is a dictionary-like object whose keys are strings and whose values are *Morsel* instances. Note that upon setting a key to a value, the value is first converted to a *Morsel* containing the key and the value.

If *input* is given, it is passed to the *load()* method.

class Cookie.**SimpleCookie**([*input*])
This class derives from *BaseCookie* and overrides `value_decode()` and `value_encode()` to be the identity and *str()* respectively.

class Cookie.**SerialCookie**([*input*])
This class derives from *BaseCookie* and overrides `value_decode()` and `value_encode()` to be the *pickle.loads()* and *pickle.dumps()*.

2.3 版后已移除: Reading pickled values from untrusted cookie data is a huge security hole, as pickle strings can be crafted to cause arbitrary code to execute on your server. It is supported for backwards compatibility only, and may eventually go away.

class Cookie.**SmartCookie**([*input*])
This class derives from *BaseCookie*. It overrides `value_decode()` to be *pickle.loads()* if it is a valid pickle, and otherwise the value itself. It overrides `value_encode()` to be *pickle.dumps()* unless it is a string, in which case it returns the value itself.

2.3 版后已移除: The same security warning from *SerialCookie* applies here.

A further security note is warranted. For backwards compatibility, the *Cookie* module exports a class named `Cookie` which is just an alias for *SmartCookie*. This is probably a mistake and will likely be removed in a future version. You should not use the `Cookie` class in your applications, for the same reason why you should not use the *SerialCookie* class.

参见:

Module *cookielib* HTTP cookie handling for web *clients*. The *cookielib* and *Cookie* modules do not depend on each other.

RFC 2109 - HTTP State Management Mechanism This is the state management specification implemented by this module.

20.22.1 Cookie Objects

`BaseCookie.`**`value_decode`**(*val*)
Return a decoded value from a string representation. Return value can be any type. This method does nothing in *BaseCookie* —it exists so it can be overridden.

`BaseCookie.`**`value_encode`**(*val*)
Return an encoded value. *val* can be any type, but return value must be a string. This method does nothing in *BaseCookie* —it exists so it can be overridden.

In general, it should be the case that *value_encode()* and *value_decode()* are inverses on the range of *value_decode*.

`BaseCookie.`**`output`**([*attrs*[, *header*[, *sep*]]])
Return a string representation suitable to be sent as HTTP headers. *attrs* and *header* are sent to each *Morsel*'s *output()* method. *sep* is used to join the headers together, and is by default the combination `'\r\n'` (CRLF).

在 2.5 版更改: The default separator has been changed from `'\n'` to match the cookie specification.

`BaseCookie.`**`js_output`**([*attrs*])
Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP headers was sent.

The meaning for *attrs* is the same as in *output()*.

`BaseCookie.`**`load`**(*rawdata*)
If *rawdata* is a string, parse it as an `HTTP_COOKIE` and add the values found there as *Morsel*s. If it is a dictionary, it is equivalent to:

```
for k, v in rawdata.items():
    cookie[k] = v
```

20.22.2 Morsel Objects

`class` `Cookie.`**`Morsel`**
Abstract a key/value pair, which has some **RFC 2109** attributes.

Morsels are dictionary-like objects, whose set of keys is constant —the valid **RFC 2109** attributes, which are

- `expires`
- `path`
- `comment`

- domain
- max-age
- secure
- version
- httponly

The attribute httponly specifies that the cookie is only transferred in HTTP requests, and is not accessible through JavaScript. This is intended to mitigate some forms of cross-site scripting.

The keys are case-insensitive.

2.6 新版功能: The httponly attribute was added.

Morsel.**value**
The value of the cookie.

Morsel.**coded_value**
The encoded value of the cookie —this is what should be sent.

Morsel.**key**
The name of the cookie.

Morsel.**set**(*key*, *value*, *coded_value*)
Set the *key*, *value* and *coded_value* attributes.

Morsel.**isReservedKey**(*K*)
Whether *K* is a member of the set of keys of a *Morsel*.

Morsel.**output**([*attrs*[, *header*]])
Return a string representation of the Morsel, suitable to be sent as an HTTP header. By default, all the attributes are included, unless *attrs* is given, in which case it should be a list of attributes to use. *header* is by default "Set-Cookie:".

Morsel.**js_output**([*attrs*])
Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP header was sent.

The meaning for *attrs* is the same as in *output()*.

Morsel.**OutputString**([*attrs*])
Return a string representing the Morsel, without any surrounding HTTP or JavaScript.

The meaning for *attrs* is the same as in *output()*.

20.22.3 Example

The following example demonstrates how to use the *Cookie* module.

```
>>> import Cookie
>>> C = Cookie.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print C # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print C.output() # same thing
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
```

(下页继续)

(续上页)

```
>>> C = Cookie.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print C.output(header="Cookie:")
Cookie: rocky=road; Path=/cookie
>>> print C.output(attrs=[], header="Cookie:")
Cookie: rocky=road
>>> C = Cookie.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string (HTTP header)
>>> print C
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = Cookie.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\"; fudge=\\012;";')
>>> print C
Set-Cookie: keebler="E=everybody; L=\"Loves\"; fudge=\012;"
>>> C = Cookie.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print C
Set-Cookie: oreo=doublestuff; Path=/
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = Cookie.SimpleCookie()
>>> C["number"] = 7 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number=7
Set-Cookie: string=seven
>>> # SerialCookie and SmartCookie are deprecated
>>> # using it can cause security loopholes in your code.
>>> C = Cookie.SerialCookie()
>>> C["number"] = 7
>>> C["string"] = "seven"
>>> C["number"].value
7
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number="I7\012."
Set-Cookie: string="S'seven'\012p1\012."
>>> C = Cookie.SmartCookie()
>>> C["number"] = 7
>>> C["string"] = "seven"
>>> C["number"].value
7
>>> C["string"].value
'seven'
>>> print C
Set-Cookie: number="I7\012."
Set-Cookie: string=seven
```

20.23 xmlrpclib —XML-RPC client access

注解: The *xmlrpclib* module has been renamed to `xmlrpc.client` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

2.2 新版功能.

Source code: Lib/xmlrpclib.py

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP(S) as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

警告: The *xmlrpclib* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML 漏洞*.

在 2.7.9 版更改: For HTTPS URIs, *xmlrpclib* now performs all the necessary certificate and hostname checks by default.

class xmlrpclib.**ServerProxy**(*uri*[, *transport*[, *encoding*[, *verbose*[, *allow_none*[, *use_datetime*[, *context*]]]]]])

A *ServerProxy* instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal `SafeTransport` instance for https: URLs and an internal HTTP `Transport` instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag.

The following parameters govern the use of the returned proxy instance. If *allow_none* is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a *TypeError*. This is a commonly-used extension to the XML-RPC specification, but isn't supported by all clients and servers; see http://ontosys.com/xml-rpc/extensions.php for a description. The *use_datetime* flag can be used to cause date/time values to be presented as *datetime.datetime* objects; this is false by default. *datetime.datetime* objects may be passed to calls.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: `http://user:pass@host:port/path`. The `user:pass` portion will be base64-encoded as an HTTP 'Authorization' header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password. If an HTTPS URL is provided, *context* may be *ssl.SSLContext* and configures the SSL settings of the underlying HTTPS connection.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):

XML-RPC type	Python type
`boolean`	*bool*
`int` or `i4`	*int* or *long* in range from -2147483648 to 2147483647.
`double`	*float*
`string`	*str* or *unicode*
`array`	`list` or *tuple* containing conformable elements. Arrays are returned as `lists`.
`struct`	*dict*. Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their *__dict__* attribute is transmitted.
`dateTime.iso8601`	*DateTime* or *datetime.datetime*. Returned type depends on values of the *use_datetime* flags.
`base64`	*Binary*
`nil`	The `None` constant. Passing is allowed only if *allow_none* is true.

This is the full set of data types supported by XML-RPC. Method calls may also raise a special *Fault* instance, used to signal XML-RPC server errors, or *ProtocolError* used to signal an error in the HTTP/HTTPS transport layer. Both *Fault* and *ProtocolError* derive from a base class called `Error`. Note that even though starting with Python 2.2 you can subclass built-in types, the xmlrpclib module currently does not marshal instances of such subclasses.

When passing strings, characters special to XML such as <, >, and `&` will be automatically escaped. However, it's the caller's responsibility to ensure that the string is free of characters that aren't allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return); failing to do this will result in an XML-RPC request that isn't well-formed XML. If you have to pass arbitrary strings via XML-RPC, use the *Binary* wrapper class described below.

`Server` is retained as an alias for *ServerProxy* for backwards compatibility. New code should use *ServerProxy*.

在 2.5 版更改: The *use_datetime* flag was added.

在 2.6 版更改: Instances of *new-style classes* can be passed in if they have an *__dict__* attribute and don't have a base class that is marshalled in a special way.

在 2.7.9 版更改: Added the *context* argument.

参见:

XML-RPC HOWTO A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

XML-RPC Introspection Describes the XML-RPC protocol extension for introspection.

XML-RPC Specification The official specification.

Unofficial XML-RPC Errata Fredrik Lundh's "unofficial errata, intended to clarify certain details in the XML-RPC specification, as well as hint at 'best practices' to use when designing your own XML-RPC implementations."

20.23.1 ServerProxy Objects

A *ServerProxy* instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a *Fault* or *ProtocolError* object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved `system` attribute:

`ServerProxy.system.`**`listMethods`**`()`
This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

`ServerProxy.system.`**`methodSignature`**`(`*name*`)`
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply "string, array". If it expects three integers and returns a string, its signature is "string, int, int, int".

If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other than list.

`ServerProxy.system.`**`methodHelp`**`(`*name*`)`
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.

20.23.2 Boolean Objects

This class may be initialized from any Python value; the instance returned depends only on its truth value. It supports various Python operators through `__cmp__()`, `__repr__()`, `__int__()`, and `__nonzero__()` methods, all implemented in the obvious ways.

It also has the following method, supported mainly for internal use by the unmarshalling code:

`Boolean.`**`encode`**`(`*out*`)`
Write the XML-RPC encoding of this Boolean item to the out stream object.

A working example follows. The server code:

```
import xmlrpclib
from SimpleXMLRPCServer import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(is_even, "is_even")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
print "3 is even: %s" % str(proxy.is_even(3))
print "100 is even: %s" % str(proxy.is_even(100))
```

20.23.3 DateTime Objects

class xmlrpclib.**DateTime**

This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a *datetime.datetime* instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*string*)

Accept a string as the instance's new time value.

encode(*out*)

Write the XML-RPC encoding of this *DateTime* item to the *out* stream object.

It also supports certain of Python's built-in operators through `__cmp__()` and `__repr__()` methods.

A working example follows. The server code:

```
import datetime
from SimpleXMLRPCServer import SimpleXMLRPCServer
import xmlrpclib

def today():
    today = datetime.datetime.today()
    return xmlrpclib.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(today, "today")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpclib
import datetime

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print "Today: %s" % converted.strftime("%d.%m.%Y, %H:%M")
```

20.23.4 Binary Objects

class xmlrpclib.**Binary**
This class may be initialized from string data (which may include NULs). The primary access to the content of a *Binary* object is provided by an attribute:

data
The binary data encapsulated by the *Binary* instance. The data is provided as an 8-bit string.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*string*)
Accept a base64 string and decode it as the instance's new data.

encode(*out*)
Write the XML-RPC base 64 encoding of this binary item to the *out* stream object.

The encoded data will have newlines every 76 characters as per RFC 2045 section 6.8, which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python's built-in operators through a `__cmp__()` method.

Example usage of the binary objects. We're going to transfer an image over XMLRPC:

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
import xmlrpclib

def python_logo():
     with open("python_logo.jpg", "rb") as handle:
          return xmlrpclib.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

The client gets the image and saves it to a file:

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

20.23.5 Fault Objects

class xmlrpclib.**Fault**
A *Fault* object encapsulates the content of an XML-RPC fault tag. Fault objects have the following attributes:

faultCode
A string indicating the fault type.

faultString
A string containing a diagnostic message associated with the fault.

In the following example we're going to intentionally cause a *Fault* by returning a complex type object. The server code:

```
from SimpleXMLRPCServer import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning a
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(add, 'add')

server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpclib.Fault as err:
    print "A fault occurred"
    print "Fault code: %d" % err.faultCode
    print "Fault string: %s" % err.faultString
```

20.23.6 ProtocolError Objects

class xmlrpclib.**ProtocolError**

A *ProtocolError* object describes a protocol error in the underlying transport layer (such as a 404 'not found' error if the server named by the URI does not exist). It has the following attributes:

url
The URI or URL that triggered the error.

errcode
The error code.

errmsg
The error message or diagnostic string.

headers
A string containing the headers of the HTTP/HTTPS request that triggered the error.

In the following example we're going to intentionally cause a *ProtocolError* by providing a URI that doesn't point to an XMLRPC server:

```
import xmlrpclib

# create a ServerProxy with a URI that doesn't respond to XMLRPC requests
proxy = xmlrpclib.ServerProxy("http://www.google.com/")

try:
    proxy.some_method()
except xmlrpclib.ProtocolError as err:
    print "A protocol error occurred"
    print "URL: %s" % err.url
    print "HTTP/HTTPS headers: %s" % err.headers
```

(下页继续)

(续上页)

```
    print "Error code: %d" % err.errcode
    print "Error message: %s" % err.errmsg
```

20.23.7 MultiCall Objects

2.4 新版功能.

The *MultiCall* object provides a way to encapsulate multiple calls to a remote server into a single request[1].

class xmlrpclib.**MultiCall**(*server*)

Create an object used to boxcar method calls. *server* is the eventual target of the call. Calls can be made to the result object, but they will immediately return `None`, and only store the call name and parameters in the *MultiCall* object. Calling the object itself causes all stored calls to be transmitted as a single `system.multicall` request. The result of this call is a *generator*; iterating over this generator yields the individual results.

A usage example of this class follows. The server code

```
from SimpleXMLRPCServer import SimpleXMLRPCServer

def add(x,y):
    return x+y

def subtract(x, y):
    return x-y

def multiply(x, y):
    return x*y

def divide(x, y):
    return x/y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
multicall = xmlrpclib.MultiCall(proxy)
multicall.add(7,3)
multicall.subtract(7,3)
multicall.multiply(7,3)
multicall.divide(7,3)
result = multicall()

print "7+3=%d, 7-3=%d, 7*3=%d, 7/3=%d" % tuple(result)
```

[1] This approach has been first presented in a discussion on xmlrpc.com.

20.23.8 Convenience Functions

xmlrpclib.**boolean**(*value*)
Convert any Python value to one of the XML-RPC Boolean constants, `True` or `False`.

xmlrpclib.**dumps**(*params*[, *methodname*[, *methodresponse*[, *encoding*[, *allow_none*]]]])
Convert *params* into an XML-RPC request. or into a response if *methodresponse* is true. *params* can be either a tuple of arguments or an instance of the `Fault` exception class. If *methodresponse* is true, only a single value can be returned, meaning that *params* must be of length 1. *encoding*, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python's `None` value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for *allow_none*.

xmlrpclib.**loads**(*data*[, *use_datetime*])
Convert an XML-RPC request or response into Python objects, a `(params, methodname)`. *params* is a tuple of argument; *methodname* is a string, or `None` if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a `Fault` exception. The *use_datetime* flag can be used to cause date/time values to be presented as `datetime.datetime` objects; this is false by default.

在 2.5 版更改: The *use_datetime* flag was added.

20.23.9 Example of Client Usage

```
# simple test program (from the XML-RPC specification)
from xmlrpclib import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
server = ServerProxy("http://betty.userland.com")

print server

try:
    print server.examples.getStateName(41)
except Error as v:
    print "ERROR", v
```

To access an XML-RPC server through a HTTP proxy, you need to define a custom transport. The following example shows how:

```
import xmlrpclib, httplib

class ProxiedTransport(xmlrpclib.Transport):
    def set_proxy(self, proxy):
        self.proxy = proxy

    def make_connection(self, host):
        self.realhost = host
        h = httplib.HTTPConnection(self.proxy)
        return h

    def send_request(self, connection, handler, request_body):
        connection.putrequest("POST", 'http://%s%s' % (self.realhost, handler))

    def send_host(self, connection, host):
        connection.putheader('Host', self.realhost)

p = ProxiedTransport()
```

(下页继续)

(续上页)

```
p.set_proxy('proxy-server:8080')
server = xmlrpclib.ServerProxy('http://time.xmlrpc.com/RPC2', transport=p)
print server.currentTime.getCurrentTime()
```

20.23.10 Example of Client and Server Usage

See *SimpleXMLRPCServer Example*.

20.24 SimpleXMLRPCServer —Basic XML-RPC server

注解: The *SimpleXMLRPCServer* module has been merged into `xmlrpc.server` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

2.2 新版功能.

Source code: Lib/SimpleXMLRPCServer.py

The *SimpleXMLRPCServer* module provides a basic server framework for XML-RPC servers written in Python. Servers can either be free standing, using *SimpleXMLRPCServer*, or embedded in a CGI environment, using *CGIXMLRPCRequestHandler*.

class SimpleXMLRPCServer.**SimpleXMLRPCServer**(*addr[, requestHandler[, logRequests[, allow_none[, encoding[, bind_and_activate]]]]]*)

Create a new server instance. This class provides methods for registration of functions that can be called by the XML-RPC protocol. The *requestHandler* parameter should be a factory for request handler instances; it defaults to *SimpleXMLRPCRequestHandler*. The *addr* and *requestHandler* parameters are passed to the *SocketServer.TCPServer* constructor. If *logRequests* is true (the default), requests will be logged; setting this parameter to false will turn off logging. The *allow_none* and *encoding* parameters are passed on to *xmlrpclib* and control the XML-RPC responses that will be returned from the server. The *bind_and_activate* parameter controls whether `server_bind()` and `server_activate()` are called immediately by the constructor; it defaults to true. Setting it to false allows code to manipulate the *allow_reuse_address* class variable before the address is bound.

在 2.5 版更改: The *allow_none* and *encoding* parameters were added.

在 2.6 版更改: The *bind_and_activate* parameter was added.

class SimpleXMLRPCServer.**CGIXMLRPCRequestHandler**([*allow_none*[, *encoding*]])

Create a new instance to handle XML-RPC requests in a CGI environment. The *allow_none* and *encoding* parameters are passed on to *xmlrpclib* and control the XML-RPC responses that will be returned from the server.

2.3 新版功能.

在 2.5 版更改: The *allow_none* and *encoding* parameters were added.

class SimpleXMLRPCServer.**SimpleXMLRPCRequestHandler**

Create a new request handler instance. This request handler supports `POST` requests and modifies logging so that the *logRequests* parameter to the *SimpleXMLRPCServer* constructor parameter is honored.

20.24.1 SimpleXMLRPCServer Objects

The *SimpleXMLRPCServer* class is based on *SocketServer.TCPServer* and provides a means of creating simple, stand alone XML-RPC servers.

SimpleXMLRPCServer.**register_function**(*function*[, *name*])
Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with *function*, otherwise `function.__name__` will be used. *name* can be either a normal or Unicode string, and may contain characters not legal in Python identifiers, including the period character.

SimpleXMLRPCServer.**register_instance**(*instance*[, *allow_dotted_names*])
Register an object which is used to expose method names which have not been registered using *register_function()*. If *instance* contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request. Its API is `def _dispatch(self, method, params)` (note that *params* does not represent a variable argument list). If it calls an underlying function to perform its task, that function is called as `func(*params)`, expanding the parameter list. The return value from `_dispatch()` is returned to the client as the result. If *instance* does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method.

If the optional *allow_dotted_names* argument is true and the instance does not have a `_dispatch()` method, then if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

> **警告:** Enabling the *allow_dotted_names* option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your machine. Only use this option on a secure, closed network.

在 2.3.5, 版更改: 2.4.1 *allow_dotted_names* was added to plug a security hole; prior versions are insecure.

SimpleXMLRPCServer.**register_introspection_functions**()
Registers the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

2.3 新版功能.

SimpleXMLRPCServer.**register_multicall_functions**()
Registers the XML-RPC multicall function system.multicall.

SimpleXMLRPCRequestHandler.**rpc_paths**
An attribute value that must be a tuple listing valid path portions of the URL for receiving XML-RPC requests. Requests posted to other paths will result in a 404 "no such page" HTTP error. If this tuple is empty, all paths will be considered valid. The default value is `('/', '/RPC2')`.

2.5 新版功能.

SimpleXMLRPCRequestHandler.**encode_threshold**
If this attribute is not `None`, responses larger than this value will be encoded using the *gzip* transfer encoding, if permitted by the client. The default is `1400` which corresponds roughly to a single TCP packet.

2.7 新版功能.

SimpleXMLRPCServer Example

Server code:

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
from SimpleXMLRPCServer import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
server = SimpleXMLRPCServer(("localhost", 8000),
                            requestHandler=RequestHandler)
server.register_introspection_functions()

# Register pow() function; this will use the value of
# pow.__name__ as the name, which is just 'pow'.
server.register_function(pow)

# Register a function under a different name
def adder_function(x,y):
    return x + y
server.register_function(adder_function, 'add')

# Register an instance; all the methods of the instance are
# published as XML-RPC methods (in this case, just 'div').
class MyFuncs:
    def div(self, x, y):
        return x // y

server.register_instance(MyFuncs())

# Run the server's main loop
server.serve_forever()
```

The following client code will call the methods made available by the preceding server:

```
import xmlrpclib

s = xmlrpclib.ServerProxy('http://localhost:8000')
print s.pow(2,3)  # Returns 2**3 = 8
print s.add(2,3)  # Returns 5
print s.div(5,2)  # Returns 5//2 = 2

# Print list of available methods
print s.system.listMethods()
```

The following *SimpleXMLRPCServer* example is included in the module *Lib/SimpleXMLRPCServer.py*:

```
server = SimpleXMLRPCServer(("localhost", 8000))
server.register_function(pow)
server.register_function(lambda x,y: x+y, 'add')
server.register_multicall_functions()
server.serve_forever()
```

This demo server can be run from the command line as:

```
python -m SimpleXMLRPCServer
```

Example client code which talks to the above server is included with *Lib/xmlrpclib.py*:

```
server = ServerProxy("http://localhost:8000")
print server
multi = MultiCall(server)
multi.pow(2, 9)
multi.add(5, 1)
multi.add(24, 11)
try:
    for response in multi():
        print response
except Error, v:
    print "ERROR", v
```

And the client can be invoked directly using the following command:

```
python -m xmlrpclib
```

20.24.2 CGIXMLRPCRequestHandler

The *CGIXMLRPCRequestHandler* class can be used to handle XML-RPC requests sent to Python CGI scripts.

CGIXMLRPCRequestHandler.**register_function**(*function*[, *name*])
Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with function, otherwise *function.__name__* will be used. *name* can be either a normal or Unicode string, and may contain characters not legal in Python identifiers, including the period character.

CGIXMLRPCRequestHandler.**register_instance**(*instance*)
Register an object which is used to expose method names which have not been registered using *register_function()*. If instance contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request; the return value is returned to the client as the result. If instance does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method; if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

CGIXMLRPCRequestHandler.**register_introspection_functions**()
Register the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

CGIXMLRPCRequestHandler.**register_multicall_functions**()
Register the XML-RPC multicall function `system.multicall`.

CGIXMLRPCRequestHandler.**handle_request**([*request_text* = *None*])
Handle an XML-RPC request. If *request_text* is given, it should be the POST data provided by the HTTP server, otherwise the contents of stdin will be used.

Example:

```
class MyFuncs:
    def div(self, x, y): return x // y


handler = CGIXMLRPCRequestHandler()
```

(下页继续)

(续上页)

```
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

20.25 DocXMLRPCServer —Self-documenting XML-RPC server

注解: The *DocXMLRPCServer* module has been merged into `xmlrpc.server` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

2.3 新版功能.

The *DocXMLRPCServer* module extends the classes found in *SimpleXMLRPCServer* to serve HTML documentation in response to HTTP GET requests. Servers can either be free standing, using *DocXMLRPCServer*, or embedded in a CGI environment, using *DocCGIXMLRPCRequestHandler*.

class DocXMLRPCServer.**DocXMLRPCServer**(*addr*[, *requestHandler*[, *logRequests*[, *allow_none*[, *encoding*[, *bind_and_activate*]]]]])

Create a new server instance. All parameters have the same meaning as for *SimpleXMLRPCServer.SimpleXMLRPCServer*; *requestHandler* defaults to *DocXMLRPCRequestHandler*.

class DocXMLRPCServer.**DocCGIXMLRPCRequestHandler**

Create a new instance to handle XML-RPC requests in a CGI environment.

class DocXMLRPCServer.**DocXMLRPCRequestHandler**

Create a new request handler instance. This request handler supports XML-RPC POST requests, documentation GET requests, and modifies logging so that the *logRequests* parameter to the *DocXMLRPCServer* constructor parameter is honored.

20.25.1 DocXMLRPCServer Objects

The *DocXMLRPCServer* class is derived from *SimpleXMLRPCServer.SimpleXMLRPCServer* and provides a means of creating self-documenting, stand alone XML-RPC servers. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocXMLRPCServer.**set_server_title**(*server_title*)

Set the title used in the generated HTML documentation. This title will be used inside the HTML “title” element.

DocXMLRPCServer.**set_server_name**(*server_name*)

Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a “h1” element.

DocXMLRPCServer.**set_server_documentation**(*server_documentation*)

Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

20.25.2 DocCGIXMLRPCRequestHandler

The *DocCGIXMLRPCRequestHandler* class is derived from *SimpleXMLRPCServer.CGIXMLRPCRequestHandler* and provides a means of creating self-documenting, XML-RPC CGI scripts. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocCGIXMLRPCRequestHandler.**set_server_title**(*server_title*)
Set the title used in the generated HTML documentation. This title will be used inside the HTML "title" element.

DocCGIXMLRPCRequestHandler.**set_server_name**(*server_name*)
Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a "h1" element.

DocCGIXMLRPCRequestHandler.**set_server_documentation**(*server_documentation*)
Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

CHAPTER 21

多媒体服务

本章描述的模块实现了主要用于多媒体应用的各种算法或接口。它们可在安装时自行决定。这是一个概述：

21.1 audioop —Manipulate raw audio data

The *audioop* module contains some useful operations on sound fragments. It operates on sound fragments consisting of signed integer samples 8, 16 or 32 bits wide, stored in Python strings. This is the same format as used by the *al* and *sunaudiodev* modules. All scalar items are integers, unless specified otherwise.

This module provides support for a-LAW, u-LAW and Intel/DVI ADPCM encodings.

A few of the more complicated operations only take 16-bit samples, otherwise the sample size (in bytes) is always a parameter of the operation.

The module defines the following variables and functions:

exception audioop.**error**
This exception is raised on all errors, such as unknown number of bytes per sample, etc.

audioop.**add**(*fragment1*, *fragment2*, *width*)
Return a fragment which is the addition of the two samples passed as parameters. *width* is the sample width in bytes, either `1`, `2` or `4`. Both fragments should have the same length. Samples are truncated in case of overflow.

audioop.**adpcm2lin**(*adpcmfragment*, *width*, *state*)
Decode an Intel/DVI ADPCM coded fragment to a linear fragment. See the description of *lin2adpcm()* for details on ADPCM coding. Return a tuple `(sample, newstate)` where the sample has the width specified in *width*.

audioop.**alaw2lin**(*fragment*, *width*)
Convert sound fragments in a-LAW encoding to linearly encoded sound fragments. a-LAW encoding always uses 8 bits samples, so *width* refers only to the sample width of the output fragment here.

2.5 新版功能.

audioop.**avg**(*fragment*, *width*)
Return the average over all samples in the fragment.

audioop.**avgpp**(*fragment*, *width*)
Return the average peak-peak value over all samples in the fragment. No filtering is done, so the usefulness of this routine is questionable.

audioop.**bias**(*fragment*, *width*, *bias*)
Return a fragment that is the original fragment with a bias added to each sample. Samples wrap around in case of overflow.

audioop.**cross**(*fragment*, *width*)
Return the number of zero crossings in the fragment passed as an argument.

audioop.**findfactor**(*fragment*, *reference*)
Return a factor *F* such that `rms(add(fragment, mul(reference, -F)))` is minimal, i.e., return the factor with which you should multiply *reference* to make it match as well as possible to *fragment*. The fragments should both contain 2-byte samples.

The time taken by this routine is proportional to `len(fragment)`.

audioop.**findfit**(*fragment*, *reference*)
Try to match *reference* as well as possible to a portion of *fragment* (which should be the longer fragment). This is (conceptually) done by taking slices out of *fragment*, using *findfactor()* to compute the best match, and minimizing the result. The fragments should both contain 2-byte samples. Return a tuple `(offset, factor)` where *offset* is the (integer) offset into *fragment* where the optimal match started and *factor* is the (floating-point) factor as per *findfactor()*.

audioop.**findmax**(*fragment*, *length*)
Search *fragment* for a slice of length *length* samples (not bytes!) with maximum energy, i.e., return *i* for which `rms(fragment[i*2:(i+length)*2])` is maximal. The fragments should both contain 2-byte samples.

The routine takes time proportional to `len(fragment)`.

audioop.**getsample**(*fragment*, *width*, *index*)
Return the value of sample *index* from the fragment.

audioop.**lin2adpcm**(*fragment*, *width*, *state*)
Convert samples to 4 bit Intel/DVI ADPCM encoding. ADPCM coding is an adaptive coding scheme, whereby each 4 bit number is the difference between one sample and the next, divided by a (varying) step. The Intel/DVI ADPCM algorithm has been selected for use by the IMA, so it may well become a standard.

state is a tuple containing the state of the coder. The coder returns a tuple `(adpcmfrag, newstate)`, and the *newstate* should be passed to the next call of *lin2adpcm()*. In the initial call, `None` can be passed as the state. *adpcmfrag* is the ADPCM coded fragment packed 2 4-bit values per byte.

audioop.**lin2alaw**(*fragment*, *width*)
Convert samples in the audio fragment to a-LAW encoding and return this as a Python string. a-LAW is an audio encoding format whereby you get a dynamic range of about 13 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

2.5 新版功能.

audioop.**lin2lin**(*fragment*, *width*, *newwidth*)
Convert samples between 1-, 2- and 4-byte formats.

注解： In some audio formats, such as .WAV files, 16 and 32 bit samples are signed, but 8 bit samples are unsigned. So when converting to 8 bit wide samples for these formats, you need to also add 128 to the result:

```
new_frames = audioop.lin2lin(frames, old_width, 1)
new_frames = audioop.bias(new_frames, 1, 128)
```

The same, in reverse, has to be applied when converting from 8 to 16 or 32 bit width samples.

---

audioop.**lin2ulaw**(*fragment*, *width*)
Convert samples in the audio fragment to u-LAW encoding and return this as a Python string. u-LAW is an audio encoding format whereby you get a dynamic range of about 14 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

audioop.**max**(*fragment*, *width*)
Return the maximum of the *absolute value* of all samples in a fragment.

audioop.**maxpp**(*fragment*, *width*)
Return the maximum peak-peak value in the sound fragment.

audioop.**minmax**(*fragment*, *width*)
Return a tuple consisting of the minimum and maximum values of all samples in the sound fragment.

audioop.**mul**(*fragment*, *width*, *factor*)
Return a fragment that has all samples in the original fragment multiplied by the floating-point value *factor*. Samples are truncated in case of overflow.

audioop.**ratecv**(*fragment*, *width*, *nchannels*, *inrate*, *outrate*, *state*[, *weightA*[, *weightB*]])
Convert the frame rate of the input fragment.

state is a tuple containing the state of the converter. The converter returns a tuple `(newfragment, newstate)`, and *newstate* should be passed to the next call of *ratecv()*. The initial call should pass `None` as the state.

The *weightA* and *weightB* arguments are parameters for a simple digital filter and default to `1` and `0` respectively.

audioop.**reverse**(*fragment*, *width*)
Reverse the samples in a fragment and returns the modified fragment.

audioop.**rms**(*fragment*, *width*)
Return the root-mean-square of the fragment, i.e. `sqrt(sum(S_i^2)/n)`.

This is a measure of the power in an audio signal.

audioop.**tomono**(*fragment*, *width*, *lfactor*, *rfactor*)
Convert a stereo fragment to a mono fragment. The left channel is multiplied by *lfactor* and the right channel by *rfactor* before adding the two channels to give a mono signal.

audioop.**tostereo**(*fragment*, *width*, *lfactor*, *rfactor*)
Generate a stereo fragment from a mono fragment. Each pair of samples in the stereo fragment are computed from the mono sample, whereby left channel samples are multiplied by *lfactor* and right channel samples by *rfactor*.

audioop.**ulaw2lin**(*fragment*, *width*)
Convert sound fragments in u-LAW encoding to linearly encoded sound fragments. u-LAW encoding always uses 8 bits samples, so *width* refers only to the sample width of the output fragment here.

Note that operations such as *mul()* or *max()* make no distinction between mono and stereo fragments, i.e. all samples are treated equal. If this is a problem the stereo fragment should be split into two mono fragments first and recombined later. Here is an example of how to do that:

```
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(lsample, width, lfactor)
    rsample = audioop.mul(rsample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

If you use the ADPCM coder to build network packets and you want your protocol to be stateless (i.e. to be able to tolerate packet loss) you should not only transmit the data but also the state. Note that you should send the *initial* state (the one you passed to *lin2adpcm()*) along to the decoder, not the final state (as returned by the coder). If you want to use *struct.Struct* to store the state in binary you can code the first element (the predicted value) in 16 bits and the second (the delta index) in 8.

The ADPCM coders have never been tried against other ADPCM coders, only against themselves. It could well be that I misinterpreted the standards in which case they will not be interoperable with the respective standards.

The `find*()` routines might look a bit funny at first sight. They are primarily meant to do echo cancellation. A reasonably fast way to do this is to pick the most energetic piece of the output sample, locate that in the input sample and subtract the whole output sample from the input sample:

```
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)    # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
    #              out_test)
    prefill = '\0'*(pos+ipos)*2
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata, 2, -factor) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

21.2 imageop —Manipulate raw image data

2.6 版后已移除: The *imageop* module has been removed in Python 3.

The *imageop* module contains some useful operations on images. It operates on images consisting of 8 or 32 bit pixels stored in Python strings. This is the same format as used by `gl.lrectwrite()` and the *imgfile* module.

The module defines the following variables and functions:

exception imageop.**error**
: This exception is raised on all errors, such as unknown number of bits per pixel, etc.

imageop.**crop**(*image*, *psize*, *width*, *height*, *x0*, *y0*, *x1*, *y1*)
: Return the selected part of *image*, which should be *width* by *height* in size and consist of pixels of *psize* bytes. *x0*, *y0*, *x1* and *y1* are like the `gl.lrectread()` parameters, i.e. the boundary is included in the new image. The new boundaries need not be inside the picture. Pixels that fall outside the old image will have their value set to zero. If *x0* is bigger than *x1* the new image is mirrored. The same holds for the y coordinates.

imageop.**scale**(*image*, *psize*, *width*, *height*, *newwidth*, *newheight*)
: Return *image* scaled to size *newwidth* by *newheight*. No interpolation is done, scaling is done by simple-minded pixel duplication or removal. Therefore, computer-generated images or dithered images will not look nice after scaling.

imageop.**tovideo**(*image*, *psize*, *width*, *height*)
: Run a vertical low-pass filter over an image. It does so by computing each destination pixel as the average of two vertically-aligned source pixels. The main use of this routine is to forestall excessive flicker if the image is displayed on a video device that uses interlacing, hence the name.

imageop.**grey2mono**(*image*, *width*, *height*, *threshold*)
: Convert an 8-bit deep greyscale image to a 1-bit deep image by thresholding all the pixels. The resulting image is tightly packed and is probably only useful as an argument to *mono2grey()*.

imageop.**dither2mono**(*image*, *width*, *height*)
Convert an 8-bit greyscale image to a 1-bit monochrome image using a (simple-minded) dithering algorithm.

imageop.**mono2grey**(*image*, *width*, *height*, *p0*, *p1*)
Convert a 1-bit monochrome image to an 8 bit greyscale or color image. All pixels that are zero-valued on input get value *p0* on output and all one-value input pixels get value *p1* on output. To convert a monochrome black-and-white image to greyscale pass the values `0` and `255` respectively.

imageop.**grey2grey4**(*image*, *width*, *height*)
Convert an 8-bit greyscale image to a 4-bit greyscale image without dithering.

imageop.**grey2grey2**(*image*, *width*, *height*)
Convert an 8-bit greyscale image to a 2-bit greyscale image without dithering.

imageop.**dither2grey2**(*image*, *width*, *height*)
Convert an 8-bit greyscale image to a 2-bit greyscale image with dithering. As for *dither2mono()*, the dithering algorithm is currently very simple.

imageop.**grey42grey**(*image*, *width*, *height*)
Convert a 4-bit greyscale image to an 8-bit greyscale image.

imageop.**grey22grey**(*image*, *width*, *height*)
Convert a 2-bit greyscale image to an 8-bit greyscale image.

imageop.**backward_compatible**
If set to 0, the functions in this module use a non-backward compatible way of representing multi-byte pixels on little-endian systems. The SGI for which this module was originally written is a big-endian system, so setting this variable will have no effect. However, the code wasn't originally intended to run on anything else, so it made assumptions about byte order which are not universal. Setting this variable to 0 will cause the byte order to be reversed on little-endian systems, so that it then is the same as on big-endian systems.

21.3 aifc —Read and write AIFF and AIFC files

Source code: Lib/aifc.py

This module provides support for reading and writing AIFF and AIFF-C files. AIFF is Audio Interchange File Format, a format for storing digital audio samples in a file. AIFF-C is a newer version of the format that includes the ability to compress the audio data.

注解: Some operations may only work under IRIX; these will raise *ImportError* when attempting to import the `cl` module, which is only available on IRIX.

Audio files have a number of parameters that describe the audio data. The sampling rate or frame rate is the number of times per second the sound is sampled. The number of channels indicate if the audio is mono, stereo, or quadro. Each frame consists of one sample per channel. The sample size is the size in bytes of each sample. Thus a frame consists of *nchannels***samplesize* bytes, and a second's worth of audio consists of *nchannels***samplesize***framerate* bytes.

For example, CD quality audio has a sample size of two bytes (16 bits), uses two channels (stereo) and has a frame rate of 44,100 frames/second. This gives a frame size of 4 bytes (2*2), and a second's worth occupies 2*2*44100 bytes (176,400 bytes).

Module *aifc* defines the following function:

aifc.**open**(*file*[, *mode*])
Open an AIFF or AIFF-C file and return an object instance with methods that are described below. The argument

file is either a string naming a file or a file object. *mode* must be `'r'` or `'rb'` when the file must be opened for reading, or `'w'` or `'wb'` when the file must be opened for writing. If omitted, `file.mode` is used if it exists, otherwise `'rb'` is used. When used for writing, the file object should be seekable, unless you know ahead of time how many samples you are going to write in total and use `writeframesraw()` and `setnframes()`.

Objects returned by *open()* when a file is opened for reading have the following methods:

`aifc.`**`getnchannels`**`()`
Return the number of audio channels (1 for mono, 2 for stereo).

`aifc.`**`getsampwidth`**`()`
Return the size in bytes of individual samples.

`aifc.`**`getframerate`**`()`
Return the sampling rate (number of audio frames per second).

`aifc.`**`getnframes`**`()`
Return the number of audio frames in the file.

`aifc.`**`getcomptype`**`()`
Return a four-character string describing the type of compression used in the audio file. For AIFF files, the returned value is `'NONE'`.

`aifc.`**`getcompname`**`()`
Return a human-readable description of the type of compression used in the audio file. For AIFF files, the returned value is `'not compressed'`.

`aifc.`**`getparams`**`()`
Return a tuple consisting of all of the above values in the above order.

`aifc.`**`getmarkers`**`()`
Return a list of markers in the audio file. A marker consists of a tuple of three elements. The first is the mark ID (an integer), the second is the mark position in frames from the beginning of the data (an integer), the third is the name of the mark (a string).

`aifc.`**`getmark`**`(`*id*`)`
Return the tuple as described in *getmarkers()* for the mark with the given *id*.

`aifc.`**`readframes`**`(`*nframes*`)`
Read and return the next *nframes* frames from the audio file. The returned data is a string containing for each frame the uncompressed samples of all channels.

`aifc.`**`rewind`**`()`
Rewind the read pointer. The next *readframes()* will start from the beginning.

`aifc.`**`setpos`**`(`*pos*`)`
Seek to the specified frame number.

`aifc.`**`tell`**`()`
Return the current frame number.

`aifc.`**`close`**`()`
Close the AIFF file. After calling this method, the object can no longer be used.

Objects returned by *open()* when a file is opened for writing have all the above methods, except for `readframes()` and `setpos()`. In addition the following methods exist. The `get*()` methods can only be called after the corresponding `set*()` methods have been called. Before the first `writeframes()` or `writeframesraw()`, all parameters except for the number of frames must be filled in.

`aifc.`**`aiff`**`()`
Create an AIFF file. The default is that an AIFF-C file is created, unless the name of the file ends in `'.aiff'` in which case the default is an AIFF file.

aifc.**aifc**()
Create an AIFF-C file. The default is that an AIFF-C file is created, unless the name of the file ends in '.aiff' in which case the default is an AIFF file.

aifc.**setnchannels**(*nchannels*)
Specify the number of channels in the audio file.

aifc.**setsampwidth**(*width*)
Specify the size in bytes of audio samples.

aifc.**setframerate**(*rate*)
Specify the sampling frequency in frames per second.

aifc.**setnframes**(*nframes*)
Specify the number of frames that are to be written to the audio file. If this parameter is not set, or not set correctly, the file needs to support seeking.

aifc.**setcomptype**(*type*, *name*)
Specify the compression type. If not specified, the audio data will not be compressed. In AIFF files, compression is not possible. The name parameter should be a human-readable description of the compression type, the type parameter should be a four-character string. Currently the following compression types are supported: NONE, ULAW, ALAW, G722.

aifc.**setparams**(*nchannels*, *sampwidth*, *framerate*, *comptype*, *compname*)
Set all the above parameters at once. The argument is a tuple consisting of the various parameters. This means that it is possible to use the result of a *getparams()* call as argument to *setparams()*.

aifc.**setmark**(*id*, *pos*, *name*)
Add a mark with the given id (larger than 0), and the given name at the given position. This method can be called at any time before *close()*.

aifc.**tell**()
Return the current write position in the output file. Useful in combination with *setmark()*.

aifc.**writeframes**(*data*)
Write data to the output file. This method can only be called after the audio file parameters have been set.

aifc.**writeframesraw**(*data*)
Like *writeframes()*, except that the header of the audio file is not updated.

aifc.**close**()
Close the AIFF file. The header of the file is updated to reflect the actual size of the audio data. After calling this method, the object can no longer be used.

21.4 sunau —读写 Sun AU 文件

源代码: Lib/sunau.py

---

sunau 模拟提供了一个处理 Sun AU 声音格式的便利接口。请注意此模块与*aifc* 和*wave* 是兼容接口的。

音频文件由标头和数据组成。标头的字段为:

域	目录
magic word	四个字节 `.snd`
header size	标头的大小，包括信息，以字节为单位。
data size	数据的物理大小，以字节为单位。
编码	指示音频样本的编码方式。
sample rate	采样率
# of channels	的通道数。
info	提供音频文件描述的 ASCII 字符串（用空字节填充）。

Apart from the info field, all header fields are 4 bytes in size. They are all 32-bit unsigned integers encoded in big-endian byte order.

The *sunau* module defines the following functions:

sunau.**open**(*file*, *mode*)
If *file* is a string, open the file by that name, otherwise treat it as a seekable file-like object. *mode* can be any of

'r' 只读模式。

'w' 只写模式。

Note that it does not allow read/write files.

A *mode* of `'r'` returns an `AU_read` object, while a *mode* of `'w'` or `'wb'` returns an `AU_write` object.

sunau.**openfp**(*file*, *mode*)
同*open()*，用于向后兼容。

The *sunau* module defines the following exception:

exception sunau.**Error**
An error raised when something is impossible because of Sun AU specs or implementation deficiency.

The *sunau* module defines the following data items:

sunau.**AUDIO_FILE_MAGIC**
An integer every valid Sun AU file begins with, stored in big-endian form. This is the string `.snd` interpreted as an integer.

sunau.**AUDIO_FILE_ENCODING_MULAW_8**
sunau.**AUDIO_FILE_ENCODING_LINEAR_8**
sunau.**AUDIO_FILE_ENCODING_LINEAR_16**
sunau.**AUDIO_FILE_ENCODING_LINEAR_24**
sunau.**AUDIO_FILE_ENCODING_LINEAR_32**
sunau.**AUDIO_FILE_ENCODING_ALAW_8**
Values of the encoding field from the AU header which are supported by this module.

sunau.**AUDIO_FILE_ENCODING_FLOAT**
sunau.**AUDIO_FILE_ENCODING_DOUBLE**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G721**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G722**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G723_3**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G723_5**
Additional known values of the encoding field from the AU header, but which are not supported by this module.

21.4.1 AU_read 对象

AU_read objects, as returned by *open()* above, have the following methods:

AU_read.**close**()
Close the stream, and make the instance unusable. (This is called automatically on deletion.)

AU_read.**getnchannels**()
Returns number of audio channels (1 for mono, 2 for stereo).

AU_read.**getsampwidth**()
返回采样字节长度。

AU_read.**getframerate**()
返回采样频率。

AU_read.**getnframes**()
返回音频总帧数。

AU_read.**getcomptype**()
Returns compression type. Supported compression types are `'ULAW'`, `'ALAW'` and `'NONE'`.

AU_read.**getcompname**()
Human-readable version of *getcomptype()*. The supported types have the respective names `'CCITT G.711 u-law'`, `'CCITT G.711 A-law'` and `'not compressed'`.

AU_read.**getparams**()
Returns a tuple `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, equivalent to output of the `get*()` methods.

AU_read.**readframes**(*n*)
Reads and returns at most *n* frames of audio, as a string of bytes. The data will be returned in linear format. If the original data is in u-LAW format, it will be converted.

AU_read.**rewind**()
重置文件指针至音频开头.

以下两个方法都使用指针，具体实现由其底层决定。

AU_read.**setpos**(*pos*)
Set the file pointer to the specified position. Only values returned from *tell()* should be used for *pos*.

AU_read.**tell**()
Return current file pointer position. Note that the returned value has nothing to do with the actual position in the file.

The following two functions are defined for compatibility with the *aifc*, and don't do anything interesting.

AU_read.**getmarkers**()
返回 None。

AU_read.**getmark**(*id*)
引发错误异常。

21.4.2 AU_write 对象

AU_write objects, as returned by *open()* above, have the following methods:

AU_write.**setnchannels**(*n*)
设置声道数。

AU_write.**setsampwidth**(*n*)
Set the sample width (in bytes.)

AU_write.**setframerate**(*n*)
Set the frame rate.

AU_write.**setnframes**(*n*)
Set the number of frames. This can be later changed, when and if more frames are written.

AU_write.**setcomptype**(*type*, *name*)
Set the compression type and description. Only `'NONE'` and `'ULAW'` are supported on output.

AU_write.**setparams**(*tuple*)
The *tuple* should be `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, with values valid for the `set*()` methods. Set all parameters.

AU_write.**tell**()
Return current position in the file, with the same disclaimer for the *AU_read.tell()* and *AU_read.setpos()* methods.

AU_write.**writeframesraw**(*data*)
写入音频数据但不更新 *nframes*。

AU_write.**writeframes**(*data*)
写入音频数据并更新 *nframes*。

AU_write.**close**()
Make sure *nframes* is correct, and close the file.

This method is called upon deletion.

Note that it is invalid to set any parameters after calling `writeframes()` or `writeframesraw()`.

21.5 wave —读写 WAV 格式文件

源代码: Lib/wave.py

---

wave 模块提供了一个处理 WAV 声音格式的便利接口。它不支持压缩/解压，但是支持单声道/立体声。

wave 模块定义了以下函数和异常:

wave.**open**(*file*[, *mode*])
If *file* is a string, open the file by that name, otherwise treat it as a seekable file-like object. *mode* can be any of

'r','rb' 只读模式。

'w','wb' 只写模式。

注意不支持同时读写 WAV 文件。

A *mode* of `'r'` or `'rb'` returns a `Wave_read` object, while a *mode* of `'w'` or `'wb'` returns a `Wave_write` object. If *mode* is omitted and a file-like object is passed as *file*, `file.mode` is used as the default value for *mode* (the `'b'` flag is still added if necessary).

如果操作的是文件对象，当使用 wave 对象的 `close()` 方法时，并不会真正关闭文件对象，这需要调用者负责来关闭文件对象。

`wave.`**`openfp`**(*file*, *mode*)
同*open()*，用于向后兼容。

exception `wave.`**`Error`**
当不符合 WAV 格式或无法操作时引发的错误。

21.5.1 Wave_read 对象

由*open()* 返回的 Wave_read 对象，有以下几种方法:

`Wave_read.`**`close`**()
关闭*wave* 打开的数据流并使对象不可用。当对象销毁时会自动调用。

`Wave_read.`**`getnchannels`**()
返回声道数量（`1` 为单声道，`2` 为立体声）

`Wave_read.`**`getsampwidth`**()
返回采样字节长度。

`Wave_read.`**`getframerate`**()
返回采样频率。

`Wave_read.`**`getnframes`**()
返回音频总帧数。

`Wave_read.`**`getcomptype`**()
返回压缩类型（只支持 `'NONE'` 类型）

`Wave_read.`**`getcompname`**()
getcomptype() 的通俗版本。使用 `'not compressed'` 代替 `'NONE'`。

`Wave_read.`**`getparams`**()
Returns a tuple `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, equivalent to output of the `get*()` methods.

`Wave_read.`**`readframes`**(*n*)
Reads and returns at most *n* frames of audio, as a string of bytes.

`Wave_read.`**`rewind`**()
重置文件指针至音频开头.

后面两个方法是为了和*aifc* 保持兼容，实际不做任何事情。

`Wave_read.`**`getmarkers`**()
返回 `None`。

`Wave_read.`**`getmark`**(*id*)
引发错误异常。

以下两个方法都使用指针，具体实现由其底层决定。

`Wave_read.`**`setpos`**(*pos*)
设置文件指针到指定位置。

`Wave_read.`**`tell`**()
返回当前文件指针位置。

21.5.2 Wave_write 对象

由*open()* 返回的 Wave_write 对象，有以下几种方法:

Wave_write.**close**()
Make sure *nframes* is correct, and close the file if it was opened by *wave*. This method is called upon object collection.

Wave_write.**setnchannels**(*n*)
设置声道数。

Wave_write.**setsampwidth**(*n*)
设置采样字节长度为 *n*。

Wave_write.**setframerate**(*n*)
设置采样频率为 *n*。

Wave_write.**setnframes**(*n*)
Set the number of frames to *n*. This will be changed later if more frames are written.

Wave_write.**setcomptype**(*type*, *name*)
设置压缩格式。目前只支持 NONE 即无压缩格式。

Wave_write.**setparams**(*tuple*)
tuple 应该是 (nchannels, sampwidth, framerate, nframes, comptype, compname)，每项的值应可用于 set*() 方法。设置所有形参。

Wave_write.**tell**()
返回当前文件指针，其指针含义和*Wave_read.tell()* 以及*Wave_read.setpos()* 是一致的。

Wave_write.**writeframesraw**(*data*)
写入音频数据但不更新 *nframes*。

Wave_write.**writeframes**(*data*)
Write audio frames and make sure *nframes* is correct.

注意在调用 writeframes() 或 writeframesraw() 之后再设置任何格式参数是无效的，而且任何这样的尝试将引发*wave.Error*。

21.6 chunk —读取 IFF 分块数据

本模块提供了一个读取使用 EA IFF 85 分块的数据的接口 chunks.[1] 这种格式使用的场合有 Audio Interchange File Format (AIFF/AIFF-C) 和 Real Media File Format (RMFF) 等。与它们密切相关的 WAVE 音频文件也可使用此模块来读取。

一个 chunk 具有以下结构:

偏移	长度	目录
0	4	区块 ID
4	4	大端字节顺序的块大小，不包括头
8	*n*	数据字节，其中 *n* 是前一字段中给出的大小
8 + *n*	0 或 1	如果 *n* 为奇数且使用块对齐，则需要填充字节

ID 是一个 4 字节的字符串，用于标识块的类型。

大小字段（32 位的值，使用大端字节序编码）给出分块数据的大小，不包括 8 字节的标头。

[1] “EA IFF 85” 交换格式文件标准, Jerry Morrison, Electronic Arts, 1985 年 1 月。

使用由一个或更多分块组成的 IFF 类型文件。此处定义的*Chunk* 类的建议使用方式是在每个分块开始时实例化一个实例并从实例读取直到其末尾，在那之后可以再实例化新的实例。到达文件末尾时，创建新实例将会失败并引发*EOFError* 异常。

class chunk.**Chunk**(*file*[, *align*, *bigendian*, *inclheader*])

Class which represents a chunk. The *file* argument is expected to be a file-like object. An instance of this class is specifically allowed. The only method that is needed is *read()*. If the methods *seek()* and *tell()* are present and don't raise an exception, they are also used. If these methods are present and raise an exception, they are expected to not have altered the object. If the optional argument *align* is true, chunks are assumed to be aligned on 2-byte boundaries. If *align* is false, no alignment is assumed. The default value is true. If the optional argument *bigendian* is false, the chunk size is assumed to be in little-endian order. This is needed for WAVE audio files. The default value is true. If the optional argument *inclheader* is true, the size given in the chunk header includes the size of the header. The default value is false.

Chunk 对象支持下列方法：

getname()
返回分块的名称（ID）。这是分块的头 4 个字节。

getsize()
返回分块的大小。

close()
关闭并跳转到分块的末尾。这不会关闭下层的文件。

The remaining methods will raise *IOError* if called after the *close()* method has been called.

isatty()
返回 False。

seek(*pos*[, *whence*])
设置分块的当前位置。*whence* 参数为可选项并且默认为 0 (绝对文件定位)；其他值还有 1 (相对当前位置查找) 和 2 (相对文件末尾查找)。没有返回值。如果下层文件不支持查找，则只允许向前查找。

tell()
将当前位置返回到分块。

read([*size*])
Read at most *size* bytes from the chunk (less if the read hits the end of the chunk before obtaining *size* bytes). If the *size* argument is negative or omitted, read all data until the end of the chunk. The bytes are returned as a string object. An empty string is returned when the end of the chunk is encountered immediately.

skip()
Skip to the end of the chunk. All further calls to *read()* for the chunk will return ''. If you are not interested in the contents of the chunk, this method should be called so that the file points to the start of the next chunk.

备注

21.7 colorsys —颜色系统间的转换

源代码： Lib/colorsys.py

---

colorsys 模块定义了计算机显示器所用的 RGB (Red Green Blue) 色彩空间与三种其他色彩坐标系统 YIQ, HLS (Hue Lightness Saturation) 和 HSV (Hue Saturation Value) 表示的颜色值之间的双向转换。所有这些色彩空

间的坐标都使用浮点数值来表示。在 YIQ 空间中，Y 坐标取值为 0 和 1 之间，而 I 和 Q 坐标均可以为正数或负数。在所有其他空间中，坐标取值均为 0 和 1 之间。

参见:

More information about color spaces can be found at https://www.poynton.com/ColorFAQ.html and https://www.cambridgeincolour.com/tutorials/color-spaces.htm.

colorsys 模块定义了如下函数:

colorsys.**rgb_to_yiq**(*r*, *g*, *b*)
把颜色从 RGB 值转为 YIQ 值。

colorsys.**yiq_to_rgb**(*y*, *i*, *q*)
把颜色从 YIQ 值转为 RGB 值。

colorsys.**rgb_to_hls**(*r*, *g*, *b*)
把颜色从 RGB 值转为 HLS 值。

colorsys.**hls_to_rgb**(*h*, *l*, *s*)
把颜色从 HLS 值转为 RGB 值。

colorsys.**rgb_to_hsv**(*r*, *g*, *b*)
把颜色从 RGB 值转为 HSV 值。

colorsys.**hsv_to_rgb**(*h*, *s*, *v*)
把颜色从 HSV 值转为 RGB 值。

示例:

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

21.8 imghdr —推测图像类型

源代码 Lib/imghdr.py

---

imghdr 模块推测文件或字节流中的图像的类型。

imghdr 模块定义了以下类型:

imghdr.**what**(*filename*[, *h*])
测试包含在命名为 *filename* 的文件中的图像数据，并且返回描述此类图片的字符串。如果可选的 *h* 被提供，*filename* 将被忽略并且 *h* 包含将被测试的二进制流。

接下来的图像类型是可识别的，返回值来自 *what()*:

值	图像格式
'rgb'	SGI 图像库文件
'gif'	GIF 87a 和 89a 文件
'pbm'	便携式位图文件
'pgm'	便携式灰度图文件
'ppm'	便携式像素表文件
'tiff'	TIFF 文件
'rast'	Sun 光栅文件
'xbm'	X 位图文件
'jpeg'	JFIF 或 Exif 格式的 JPEG 数据
'bmp'	BMP 文件
'png'	便携式网络图像

2.5 新版功能: Exif detection.

你可以扩展此*imghdr* 可以被追加的这个变量识别的文件格式的列表:

imghdr.**tests**
执行单个测试的函数列表。每个函数都有两个参数: 字节流和类似开放文件的对象。当*what()* 用字节流调用时，类文件对象将是 None。

如果测试成功，这个测试函数应当返回一个描述图像类型的字符串，否则返回 None。

示例:

```
>>> import imghdr
>>> imghdr.what('bass.gif')
'gif'
```

21.9 sndhdr —推测声音文件的类型

源代码 Lib/sndhdr.py

The *sndhdr* provides utility functions which attempt to determine the type of sound data which is in a file. When these functions are able to determine what type of sound data is stored in a file, they return a tuple `(type, sampling_rate, channels, frames, bits_per_sample)`. The value for *type* indicates the data type and will be one of the strings `'aifc'`, `'aiff'`, `'au'`, `'hcom'`, `'sndr'`, `'sndt'`, `'voc'`, `'wav'`, `'8svx'`, `'sb'`, `'ub'`, or `'ul'`. The *sampling_rate* will be either the actual value or `0` if unknown or difficult to decode. Similarly, *channels* will be either the number of channels or `0` if it cannot be determined or if the value is difficult to decode. The value for *frames* will be either the number of frames or `-1`. The last item in the tuple, *bits_per_sample*, will either be the sample size in bits or `'A'` for A-LAW or `'U'` for u-LAW.

sndhdr.**what**(*filename*)
Determines the type of sound data stored in the file *filename* using *whathdr()*. If it succeeds, returns a tuple as described above, otherwise None is returned.

sndhdr.**whathdr**(*filename*)
Determines the type of sound data stored in a file based on the file header. The name of the file is given by *filename*. This function returns a tuple as described above on success, or None.

21.10 ossaudiodev —Access to OSS-compatible audio devices

2.3 新版功能.

This module allows you to access the OSS (Open Sound System) audio interface. OSS is available for a wide range of open-source and commercial Unices, and is the standard audio interface for Linux and recent versions of FreeBSD.

参见:

Open Sound System Programmer's Guide the official documentation for the OSS C API

The module defines a large number of constants supplied by the OSS device driver; see <sys/soundcard.h> on either Linux or FreeBSD for a listing.

ossaudiodev defines the following variables and functions:

exception ossaudiodev.**OSSAudioError**
This exception is raised on certain errors. The argument is a string describing what went wrong.

(If *ossaudiodev* receives an error from a system call such as open(), write(), or ioctl(), it raises *IOError*. Errors detected directly by *ossaudiodev* result in *OSSAudioError*.)

(For backwards compatibility, the exception class is also available as ossaudiodev.error.)

ossaudiodev.**open**(*mode*)
ossaudiodev.**open**(*device*, *mode*)
Open an audio device and return an OSS audio device object. This object supports many file-like methods, such as read(), write(), and fileno() (although there are subtle differences between conventional Unix read/write semantics and those of OSS audio devices). It also supports a number of audio-specific methods; see below for the complete list of methods.

device is the audio device filename to use. If it is not specified, this module first looks in the environment variable AUDIODEV for a device to use. If not found, it falls back to /dev/dsp.

mode is one of 'r' for read-only (record) access, 'w' for write-only (playback) access and 'rw' for both. Since many sound cards only allow one process to have the recorder or player open at a time, it is a good idea to open the device only for the activity needed. Further, some sound cards are half-duplex: they can be opened for reading or writing, but not both at once.

Note the unusual calling syntax: the *first* argument is optional, and the second is required. This is a historical artifact for compatibility with the older linuxaudiodev module which *ossaudiodev* supersedes.

ossaudiodev.**openmixer**([*device*])
Open a mixer device and return an OSS mixer device object. *device* is the mixer device filename to use. If it is not specified, this module first looks in the environment variable MIXERDEV for a device to use. If not found, it falls back to /dev/mixer.

21.10.1 Audio Device Objects

Before you can write to or read from an audio device, you must call three methods in the correct order:

1. setfmt() to set the output format
2. channels() to set the number of channels
3. speed() to set the sample rate

Alternately, you can use the setparameters() method to set all three audio parameters at once. This is more convenient, but may not be as flexible in all cases.

The audio device objects returned by *open()* define the following methods and (read-only) attributes:

oss_audio_device.**close**()
Explicitly close the audio device. When you are done writing to or reading from an audio device, you should explicitly close it. A closed device cannot be used again.

oss_audio_device.**fileno**()
Return the file descriptor associated with the device.

oss_audio_device.**read**(*size*)
Read *size* bytes from the audio input and return them as a Python string. Unlike most Unix device drivers, OSS audio devices in blocking mode (the default) will block *read()* until the entire requested amount of data is available.

oss_audio_device.**write**(*data*)
Write the Python string *data* to the audio device and return the number of bytes written. If the audio device is in blocking mode (the default), the entire string is always written (again, this is different from usual Unix device semantics). If the device is in non-blocking mode, some data may not be written —see *writeall()*.

oss_audio_device.**writeall**(*data*)
Write the entire Python string *data* to the audio device: waits until the audio device is able to accept data, writes as much data as it will accept, and repeats until *data* has been completely written. If the device is in blocking mode (the default), this has the same effect as *write()*; *writeall()* is only useful in non-blocking mode. Has no return value, since the amount of data written is always equal to the amount of data supplied.

The following methods each map to exactly one `ioctl()` system call. The correspondence is obvious: for example, `setfmt()` corresponds to the `SNDCTL_DSP_SETFMT` ioctl, and `sync()` to `SNDCTL_DSP_SYNC` (this can be useful when consulting the OSS documentation). If the underlying `ioctl()` fails, they all raise *IOError*.

oss_audio_device.**nonblock**()
Put the device into non-blocking mode. Once in non-blocking mode, there is no way to return it to blocking mode.

oss_audio_device.**getfmts**()
Return a bitmask of the audio output formats supported by the soundcard. Some of the formats supported by OSS are:

格式	描述
AFMT_MU_LAW	a logarithmic encoding (used by Sun `.au` files and `/dev/audio`)
AFMT_A_LAW	a logarithmic encoding
AFMT_IMA_ADPCM	a 4:1 compressed format defined by the Interactive Multimedia Association
AFMT_U8	Unsigned, 8-bit audio
AFMT_S16_LE	Signed, 16-bit audio, little-endian byte order (as used by Intel processors)
AFMT_S16_BE	Signed, 16-bit audio, big-endian byte order (as used by 68k, PowerPC, Sparc)
AFMT_S8	Signed, 8 bit audio
AFMT_U16_LE	Unsigned, 16-bit little-endian audio
AFMT_U16_BE	Unsigned, 16-bit big-endian audio

Consult the OSS documentation for a full list of audio formats, and note that most devices support only a subset of these formats. Some older devices only support `AFMT_U8`; the most common format used today is `AFMT_S16_LE`.

oss_audio_device.**setfmt**(*format*)
Try to set the current audio format to *format*—see *getfmts()* for a list. Returns the audio format that the device was set to, which may not be the requested format. May also be used to return the current audio format—do this by passing an "audio format" of `AFMT_QUERY`.

oss_audio_device.**channels**(*nchannels*)
Set the number of output channels to *nchannels*. A value of 1 indicates monophonic sound, 2 stereophonic. Some devices may have more than 2 channels, and some high-end devices may not support mono. Returns the number of channels the device was set to.

oss_audio_device.**speed**(*samplerate*)
Try to set the audio sampling rate to *samplerate* samples per second. Returns the rate actually set. Most sound devices don't support arbitrary sampling rates. Common rates are:

采样率	描述
8000	/dev/audio 的默认采样率
11025	语音录音
22050	
44100	CD 质量的音频（16 位采样和 2 通道）
96000	DVD 质量的音频（24 位采样）

oss_audio_device.**sync**()
Wait until the sound device has played every byte in its buffer. (This happens implicitly when the device is closed.) The OSS documentation recommends closing and re-opening the device rather than using *sync()*.

oss_audio_device.**reset**()
Immediately stop playing or recording and return the device to a state where it can accept commands. The OSS documentation recommends closing and re-opening the device after calling *reset()*.

oss_audio_device.**post**()
Tell the driver that there is likely to be a pause in the output, making it possible for the device to handle the pause more intelligently. You might use this after playing a spot sound effect, before waiting for user input, or before doing disk I/O.

The following convenience methods combine several ioctls, or one ioctl and some simple calculations.

oss_audio_device.**setparameters**(*format*, *nchannels*, *samplerate*[, *strict=False*])
Set the key audio sampling parameters—sample format, number of channels, and sampling rate—in one method call. *format*, *nchannels*, and *samplerate* should be as specified in the *setfmt()*, *channels()*, and *speed()* methods. If *strict* is true, *setparameters()* checks to see if each parameter was actually set to the requested value, and raises *OSSAudioError* if not. Returns a tuple (*format*, *nchannels*, *samplerate*) indicating the parameter values that were actually set by the device driver (i.e., the same as the return values of *setfmt()*, *channels()*, and *speed()*).

For example,

```
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

is equivalent to

```
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
rate = dsp.rate(rate)
```

oss_audio_device.**bufsize**()
Returns the size of the hardware buffer, in samples.

oss_audio_device.**obufcount**()
Returns the number of samples that are in the hardware buffer yet to be played.

oss_audio_device.**obuffree**()
Returns the number of samples that could be queued into the hardware buffer to be played without blocking.

Audio device objects also support several read-only attributes:

oss_audio_device.**closed**
Boolean indicating whether the device has been closed.

oss_audio_device.**name**
String containing the name of the device file.

oss_audio_device.**mode**
The I/O mode for the file, either "r", "rw", or "w".

21.10.2 Mixer Device Objects

The mixer object provides two file-like methods:

oss_mixer_device.**close**()
This method closes the open mixer device file. Any further attempts to use the mixer after this file is closed will raise an *IOError*.

oss_mixer_device.**fileno**()
Returns the file handle number of the open mixer device file.

The remaining methods are specific to audio mixing:

oss_mixer_device.**controls**()
This method returns a bitmask specifying the available mixer controls ("Control" being a specific mixable "channel", such as SOUND_MIXER_PCM or SOUND_MIXER_SYNTH). This bitmask indicates a subset of all available mixer controls—the SOUND_MIXER_* constants defined at module level. To determine if, for example, the current mixer object supports a PCM mixer, use the following Python code:

```
mixer=ossaudiodev.openmixer()
if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
    ... code ...
```

For most purposes, the SOUND_MIXER_VOLUME (master volume) and SOUND_MIXER_PCM controls should suffice—but code that uses the mixer should be flexible when it comes to choosing mixer controls. On the Gravis Ultrasound, for example, SOUND_MIXER_VOLUME does not exist.

oss_mixer_device.**stereocontrols**()
Returns a bitmask indicating stereo mixer controls. If a bit is set, the corresponding control is stereo; if it is unset, the control is either monophonic or not supported by the mixer (use in combination with *controls()* to determine which).

See the code example for the *controls()* function for an example of getting data from a bitmask.

oss_mixer_device.**reccontrols**()
Returns a bitmask specifying the mixer controls that may be used to record. See the code example for *controls()* for an example of reading from a bitmask.

oss_mixer_device.**get**(*control*)
Returns the volume of a given mixer control. The returned volume is a 2-tuple (left_volume, right_volume). Volumes are specified as numbers from 0 (silent) to 100 (full volume). If the control is monophonic, a 2-tuple is still returned, but both volumes are the same.

Raises *OSSAudioError* if an invalid control was is specified, or *IOError* if an unsupported control is specified.

oss_mixer_device.**set**(*control*, *(left*, *right)*)
Sets the volume for a given mixer control to (left,right). left and right must be ints and between 0 (silent) and 100 (full volume). On success, the new volume is returned as a 2-tuple. Note that this may not be exactly the same as the volume specified, because of the limited resolution of some soundcard's mixers.

Raises *OSSAudioError* if an invalid mixer control was specified, or if the specified volumes were out-of-range.

oss_mixer_device.**get_recsrc**()
This method returns a bitmask indicating which control(s) are currently being used as a recording source.

oss_mixer_device.**set_recsrc**(*bitmask*)
Call this function to specify a recording source. Returns a bitmask indicating the new recording source (or sources) if successful; raises *IOError* if an invalid source was specified. To set the current recording source to the microphone input:

```
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```

CHAPTER 22

国际化

本章中介绍的模块通过提供选择要在程序信息中使用的语言的机制或通过定制输出以匹配本地约定来帮助你编写不依赖于语言和区域设置的软件。

本章中描述的模块列表是：

22.1 gettext —多语种国际化服务

源代码： Lib/gettext.py

The *gettext* module provides internationalization (I18N) and localization (L10N) services for your Python modules and applications. It supports both the GNU `gettext` message catalog API and a higher level, class-based API that may be more appropriate for Python files. The interface described below allows you to write your module and application messages in one natural language, and provide a catalog of translated messages for running under different natural languages.

同时还给出一些本地化 Python 模块及应用程序的小技巧。

22.1.1 GNU gettext API

模块*gettext* 定义了下列 API，这与 **gettext** API 类似。如果你使用该 API，将会对整个应用程序产生全局的影响。如果你的应用程序支持多语种，而语言选择取决于用户的区域设置，这通常正是你所想要的。而如果你正在本地化某个 Python 模块，或者你的应用程序需要在运行时切换语言，相反你或许想用基于类的 API。

`gettext.`**`bindtextdomain`**(*domain*[, *localedir*])

Bind the *domain* to the locale directory *localedir*. More concretely, *gettext* will look for binary `.mo` files for the given domain using the path (on Unix): `localedir/language/LC_MESSAGES/domain.mo`, where *languages* is searched for in the environment variables `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG` respectively.

如果遗漏了 *localedir* 或者设置为 None，那么将返回当前 *domain* 所绑定的值[1]

gettext.**bind_textdomain_codeset**(*domain*[, *codeset*])
Bind the *domain* to *codeset*, changing the encoding of strings returned by the *gettext()* family of functions. If *codeset* is omitted, then the current binding is returned.

2.4 新版功能.

gettext.**textdomain**([*domain*])
Change or query the current global domain. If *domain* is None, then the current global domain is returned, otherwise the global domain is set to *domain*, which is returned.

gettext.**gettext**(*message*)
Return the localized translation of *message*, based on the current global domain, language, and locale directory. This function is usually aliased as _() in the local namespace (see examples below).

gettext.**lgettext**(*message*)
Equivalent to *gettext()*, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with *bind_textdomain_codeset()*.

2.4 新版功能.

gettext.**dgettext**(*domain*, *message*)
Like *gettext()*, but look the message up in the specified *domain*.

gettext.**ldgettext**(*domain*, *message*)
Equivalent to *dgettext()*, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with *bind_textdomain_codeset()*.

2.4 新版功能.

gettext.**ngettext**(*singular*, *plural*, *n*)
Like *gettext()*, but consider plural forms. If a translation is found, apply the plural formula to *n*, and return the resulting message (some languages have more than two plural forms). If no translation is found, return *singular* if *n* is 1; return *plural* otherwise.

The Plural formula is taken from the catalog header. It is a C or Python expression that has a free variable *n*; the expression evaluates to the index of the plural in the catalog. See the GNU gettext documentation for the precise syntax to be used in .po files and the formulas for a variety of languages.

2.3 新版功能.

gettext.**lngettext**(*singular*, *plural*, *n*)
Equivalent to *ngettext()*, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with *bind_textdomain_codeset()*.

2.4 新版功能.

gettext.**dngettext**(*domain*, *singular*, *plural*, *n*)
Like *ngettext()*, but look the message up in the specified *domain*.

2.3 新版功能.

gettext.**ldngettext**(*domain*, *singular*, *plural*, *n*)
Equivalent to *dngettext()*, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with *bind_textdomain_codeset()*.

2.4 新版功能.

[1] The default locale directory is system dependent; for example, on RedHat Linux it is /usr/share/locale, but on Solaris it is /usr/lib/locale. The *gettext* module does not try to support these system dependent defaults; instead its default is sys.prefix/share/locale. For this reason, it is always best to call *bindtextdomain()* with an explicit absolute path at the start of your application.

Note that GNU **gettext** also defines a `dcgettext()` method, but this was deemed not useful and so it is currently unimplemented.

Here's an example of typical usage for this API:

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print _('This is a translatable string.')
```

22.1.2 Class-based API

The class-based API of the *gettext* module gives you more flexibility and greater convenience than the GNU **gettext** API. It is the recommended way of localizing your Python applications and modules. `gettext` defines a "translations" class which implements the parsing of GNU `.mo` format files, and has methods for returning either standard 8-bit strings or Unicode strings. Instances of this "translations" class can also install themselves in the built-in namespace as the function `_()`.

`gettext.`**find**(*domain*[, *localedir*[, *languages*[, *all*]]])

This function implements the standard `.mo` file search algorithm. It takes a *domain*, identical to what *textdomain()* takes. Optional *localedir* is as in *bindtextdomain()* Optional *languages* is a list of strings, where each string is a language code.

If *localedir* is not given, then the default system locale directory is used.[2] If *languages* is not given, then the following environment variables are searched: `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG`. The first one returning a non-empty value is used for the *languages* variable. The environment variables should contain a colon separated list of languages, which will be split on the colon to produce the expected list of language code strings.

find() then expands and normalizes the languages, and then iterates through them, searching for an existing file built of these components:

`localedir/language/LC_MESSAGES/domain.mo`

The first such file name that exists is returned by *find()*. If no such file is found, then `None` is returned. If *all* is given, it returns a list of all file names, in the order in which they appear in the languages list or the environment variables.

`gettext.`**translation**(*domain*[, *localedir*[, *languages*[, *class_*[, *fallback*[, *codeset*]]]]])

Return a `Translations` instance based on the *domain*, *localedir*, and *languages*, which are first passed to *find()* to get a list of the associated `.mo` file paths. Instances with identical `.mo` file names are cached. The actual class instantiated is either *class_* if provided, otherwise `GNUTranslations`. The class's constructor must take a single file object argument. If provided, *codeset* will change the charset used to encode translated strings.

If multiple files are found, later files are used as fallbacks for earlier ones. To allow setting the fallback, *copy.copy()* is used to clone each translation object from the cache; the actual instance data is still shared with the cache.

If no `.mo` file is found, this function raises *IOError* if *fallback* is false (which is the default), and returns a *NullTranslations* instance if *fallback* is true.

在 2.4 版更改: Added the *codeset* parameter.

`gettext.`**install**(*domain*[, *localedir*[, *unicode*[, *codeset*[, *names*]]]])

This installs the function `_()` in Python's builtins namespace, based on *domain*, *localedir*, and *codeset* which

[2] See the footnote for *bindtextdomain()* above.

are passed to the function *translation()*. The *unicode* flag is passed to the resulting translation object's *install()* method.

For the *names* parameter, please see the description of the translation object's *install()* method.

As seen below, you usually mark the strings in your application that are candidates for translation, by wrapping them in a call to the `_()` function, like this:

```
print _('This string will be translated.')
```

For convenience, you want the `_()` function to be installed in Python's builtins namespace, so it is easily accessible in all modules of your application.

在 2.4 版更改: Added the *codeset* parameter.

在 2.5 版更改: Added the *names* parameter.

The `NullTranslations` class

Translation classes are what actually implement the translation of original source file message strings to translated message strings. The base class used by all translation classes is *NullTranslations*; this provides the basic interface you can use to write your own specialized translation classes. Here are the methods of `NullTranslations`:

class gettext.**NullTranslations**([*fp*])

Takes an optional file object *fp*, which is ignored by the base class. Initializes "protected" instance variables *_info* and *_charset* which are set by derived classes, as well as *_fallback*, which is set through *add_fallback()*. It then calls `self._parse(fp)` if *fp* is not `None`.

_parse(*fp*)

No-op'd in the base class, this method takes file object *fp*, and reads the data from the file, initializing its message catalog. If you have an unsupported message catalog file format, you should override this method to parse your format.

add_fallback(*fallback*)

Add *fallback* as the fallback object for the current translation object. A translation object should consult the fallback if it cannot provide a translation for a given message.

gettext(*message*)

If a fallback has been set, forward `gettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

lgettext(*message*)

If a fallback has been set, forward `lgettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

2.4 新版功能.

ugettext(*message*)

If a fallback has been set, forward `ugettext()` to the fallback. Otherwise, return the translated message as a Unicode string. Overridden in derived classes.

ngettext(*singular*, *plural*, *n*)

If a fallback has been set, forward `ngettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

2.3 新版功能.

lngettext(*singular*, *plural*, *n*)

If a fallback has been set, forward `lngettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

2.4 新版功能.

ungettext(*singular*, *plural*, *n*)
If a fallback has been set, forward `ungettext()` to the fallback. Otherwise, return the translated message as a Unicode string. Overridden in derived classes.

2.3 新版功能.

info()
Return the "protected" `_info` variable.

charset()
Return the "protected" `_charset` variable.

output_charset()
Return the "protected" `_output_charset` variable, which defines the encoding used to return translated messages.

2.4 新版功能.

set_output_charset(*charset*)
Change the "protected" `_output_charset` variable, which defines the encoding used to return translated messages.

2.4 新版功能.

install(*unicode*[, *names*])
If the *unicode* flag is false, this method installs `self.gettext()` into the built-in namespace, binding it to _. If *unicode* is true, it binds `self.ugettext()` instead. By default, *unicode* is false.

If the *names* parameter is given, it must be a sequence containing the names of functions you want to install in the builtins namespace in addition to `_()`. Supported names are `'gettext'` (bound to `self.gettext()` or `self.ugettext()` according to the *unicode* flag), `'ngettext'` (bound to `self.ngettext()` or `self.ungettext()` according to the *unicode* flag), `'lgettext'` and `'lngettext'`.

Note that this is only one way, albeit the most convenient way, to make the `_()` function available to your application. Because it affects the entire application globally, and specifically the built-in namespace, localized modules should never install `_()`. Instead, they should use this code to make `_()` available to their module:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

This puts `_()` only in the module's global namespace and so only affects calls within this module.

在 2.5 版更改: Added the *names* parameter.

The GNUTranslations class

The *gettext* module provides one additional class derived from *NullTranslations*: `GNUTranslations`. This class overrides `_parse()` to enable reading GNU **gettext** format `.mo` files in both big-endian and little-endian format. It also coerces both message ids and message strings to Unicode.

`GNUTranslations` parses optional meta-data out of the translation catalog. It is convention with GNU **gettext** to include meta-data as the translation for the empty string. This meta-data is in **RFC 822**-style `key: value` pairs, and should contain the `Project-Id-Version` key. If the key `Content-Type` is found, then the `charset` property is used to initialize the "protected" `_charset` instance variable, defaulting to `None` if not found. If the charset encoding is specified, then all message ids and message strings read from the catalog are converted to Unicode using this encoding. The `ugettext()` method always returns a Unicode, while the *gettext()* returns an encoded 8-bit

string. For the message id arguments of both methods, either Unicode strings or 8-bit strings containing only US-ASCII characters are acceptable. Note that the Unicode version of the methods (i.e. `ugettext()` and `ungettext()`) are the recommended interface to use for internationalized Python programs.

The entire set of key/value pairs are placed into a dictionary and set as the "protected" `_info` instance variable.

If the `.mo` file's magic number is invalid, or if other problems occur while reading the file, instantiating a `GNUTranslations` class can raise *IOError*.

The following methods are overridden from the base class implementation:

`GNUTranslations.`**`gettext`**(*message*)
Look up the *message* id in the catalog and return the corresponding message string, as an 8-bit string encoded with the catalog's charset encoding, if known. If there is no entry in the catalog for the *message* id, and a fallback has been set, the look up is forwarded to the fallback's *gettext()* method. Otherwise, the *message* id is returned.

`GNUTranslations.`**`lgettext`**(*message*)
Equivalent to *gettext()*, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with `set_output_charset()`.

2.4 新版功能.

`GNUTranslations.`**`ugettext`**(*message*)
Look up the *message* id in the catalog and return the corresponding message string, as a Unicode string. If there is no entry in the catalog for the *message* id, and a fallback has been set, the look up is forwarded to the fallback's *ugettext()* method. Otherwise, the *message* id is returned.

`GNUTranslations.`**`ngettext`**(*singular*, *plural*, *n*)
Do a plural-forms lookup of a message id. *singular* is used as the message id for purposes of lookup in the catalog, while *n* is used to determine which plural form to use. The returned message string is an 8-bit string encoded with the catalog's charset encoding, if known.

If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's *ngettext()* method. Otherwise, when *n* is 1 *singular* is returned, and *plural* is returned in all other cases.

2.3 新版功能.

`GNUTranslations.`**`lngettext`**(*singular*, *plural*, *n*)
Equivalent to *gettext()*, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with `set_output_charset()`.

2.4 新版功能.

`GNUTranslations.`**`ungettext`**(*singular*, *plural*, *n*)
Do a plural-forms lookup of a message id. *singular* is used as the message id for purposes of lookup in the catalog, while *n* is used to determine which plural form to use. The returned message string is a Unicode string.

If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's *ungettext()* method. Otherwise, when *n* is 1 *singular* is returned, and *plural* is returned in all other cases.

Here is an example:

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ungettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

2.3 新版功能.

Solaris message catalog support

The Solaris operating system defines its own binary `.mo` file format, but since no documentation can be found on this format, it is not supported at this time.

The Catalog constructor

GNOME uses a version of the *gettext* module by James Henstridge, but this version has a slightly different API. Its documented usage was:

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print _('hello world')
```

For compatibility with this older module, the function `Catalog()` is an alias for the *translation()* function described above.

One difference between this module and Henstridge's: his catalog objects supported access through a mapping API, but this appears to be unused and so is not currently supported.

22.1.3 Internationalizing your programs and modules

Internationalization (I18N) refers to the operation by which a program is made aware of multiple languages. Localization (L10N) refers to the adaptation of your program, once internationalized, to the local language and cultural habits. In order to provide multilingual messages for your Python programs, you need to take the following steps:

1. prepare your program or module by specially marking translatable strings
2. run a suite of tools over your marked files to generate raw messages catalogs
3. create language specific translations of the message catalogs
4. use the *gettext* module so that message strings are properly translated

In order to prepare your code for I18N, you need to look at all the strings in your files. Any string that needs to be translated should be marked by wrapping it in `_('...')` —that is, a call to the function `_()`. For example:

```
filename = 'mylog.txt'
message = _('writing a log message')
fp = open(filename, 'w')
fp.write(message)
fp.close()
```

In this example, the string `'writing a log message'` is marked as a candidate for translation, while the strings `'mylog.txt'` and `'w'` are not.

The Python distribution comes with two tools which help you generate the message catalogs once you've prepared your source code. These may or may not be available from a binary distribution, but they can be found in a source distribution, in the `Tools/i18n` directory.

The **pygettext**[3] program scans all your Python source code looking for the strings you previously marked as translatable. It is similar to the GNU **gettext** program except that it understands all the intricacies of Python source code, but knows nothing about C or C++ source code. You don't need GNU `gettext` unless you're also going to be translating C code (such as C extension modules).

[3] François Pinard has written a program called **xpot** which does a similar job. It is available as part of his po-utils package.

pygettext generates textual Uniforum-style human readable message catalog `.pot` files, essentially structured human readable files which contain every marked string in the source code, along with a placeholder for the translation strings. **pygettext** is a command line script that supports a similar command line interface as **xgettext**; for details on its use, run:

```
pygettext.py --help
```

Copies of these `.pot` files are then handed over to the individual human translators who write language-specific versions for every supported natural language. They send you back the filled in language-specific versions as a `.po` file. Using the **msgfmt.py**[4] program (in the `Tools/i18n` directory), you take the `.po` files from your translators and generate the machine-readable `.mo` binary catalog files. The `.mo` files are what the *gettext* module uses for the actual translation processing during run-time.

How you use the *gettext* module in your code depends on whether you are internationalizing a single module or your entire application. The next two sections will discuss each case.

Localizing your module

If you are localizing your module, you must take care not to make global changes, e.g. to the built-in namespace. You should not use the GNU `gettext` API but instead the class-based API.

Let's say your module is called "spam" and the module's various natural language translation `.mo` files reside in `/usr/share/locale` in GNU **gettext** format. Here's what you would put at the top of your module:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.lgettext
```

If your translators were providing you with Unicode strings in their `.po` files, you'd instead do:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.ugettext
```

Localizing your application

If you are localizing your application, you can install the `_()` function globally into the built-in namespace, usually in the main driver file of your application. This will let all your application-specific files just use `_('...')` without having to explicitly install it in each file.

In the simple case then, you need only add the following bit of code to the main driver file of your application:

```
import gettext
gettext.install('myapplication')
```

If you need to set the locale directory or the *unicode* flag, you can pass these into the *install()* function:

```
import gettext
gettext.install('myapplication', '/usr/share/locale', unicode=1)
```

[4] **msgfmt.py** is binary compatible with GNU **msgfmt** except that it provides a simpler, all-Python implementation. With this and **pygettext.py**, you generally won't need to install the GNU **gettext** package to internationalize your Python applications.

Changing languages on the fly

If your program needs to support many languages at the same time, you may want to create multiple translation instances and then switch between them explicitly, like so:

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

Deferred translations

In most coding situations, strings are translated where they are coded. Occasionally however, you need to mark strings for translation, but defer actual translation until later. A classic example is:

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]
# ...
for a in animals:
    print a
```

Here, you want to mark the strings in the `animals` list as being translatable, but you don't actually want to translate them until they are printed.

Here is one way you can handle this situation:

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print _(a)
```

This works because the dummy definition of `_()` simply returns the string unchanged. And this dummy definition will temporarily override any definition of `_()` in the built-in namespace (until the `del` command). Take care, though if you have a previous definition of `_()` in the local namespace.

Note that the second use of `_()` will not identify "a" as being translatable to the **pygettext** program, since it is not a string.

Another way to handle this is with the following example:

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print _(a)
```

In this case, you are marking translatable strings with the function `N_()`,[5] which won't conflict with any definition of `_()`. However, you will need to teach your message extraction program to look for translatable strings marked with `N_()`. **pygettext** and **xpot** both support this through the use of command line switches.

gettext() vs. lgettext()

In Python 2.4 the *lgettext()* family of functions were introduced. The intention of these functions is to provide an alternative which is more compliant with the current implementation of GNU gettext. Unlike *gettext()*, which returns strings encoded with the same codeset used in the translation file, *lgettext()* will return strings encoded with the preferred system encoding, as returned by *locale.getpreferredencoding()*. Also notice that Python 2.4 introduces new functions to explicitly choose the codeset used in translated strings. If a codeset is explicitly set, even *lgettext()* will return translated strings in the requested codeset, as would be expected in the GNU gettext implementation.

22.1.4 致谢

以下人员为创建此模块贡献了代码、反馈、设计建议、早期实现和宝贵的经验:

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer

[5] The choice of `N_()` here is totally arbitrary; it could have just as easily been `MarkThisStringForTranslation()`.

备注

22.2 locale —国际化服务

The *locale* module opens access to the POSIX locale database and functionality. The POSIX locale mechanism allows programmers to deal with certain cultural issues in an application, without requiring the programmer to know all the specifics of each country where the software is executed.

The *locale* module is implemented on top of the _locale module, which in turn uses an ANSI C locale implementation if available.

The *locale* module defines the following exception and functions:

exception locale.**Error**
Exception raised when the locale passed to *setlocale()* is not recognized.

locale.**setlocale**(*category*[, *locale*])
If *locale* is given and not None, *setlocale()* modifies the locale setting for the *category*. The available categories are listed in the data description below. *locale* may be a string, or an iterable of two strings (language code and encoding). If it's an iterable, it's converted to a locale name using the locale aliasing engine. An empty string specifies the user's default settings. If the modification of the locale fails, the exception *Error* is raised. If successful, the new locale setting is returned.

If *locale* is omitted or None, the current setting for *category* is returned.

setlocale() is not thread-safe on most systems. Applications typically start with a call of

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

This sets the locale for all categories to the user's default setting (typically specified in the LANG environment variable). If the locale is not changed thereafter, using multithreading should not cause problems.

在 2.0 版更改: Added support for iterable values of the *locale* parameter.

locale.**localeconv**()
以字典的形式返回本地约定的数据库。此字典具有以下字符串作为键:

类别	（Windows 注册表的）键	含义
LC_NUMERIC	'decimal_point'	小数点字符。
	'grouping'	Sequence of numbers specifying which relative positions the 'thousands_sep' is expected. If the sequence is terminated with *CHAR_MAX*, no further grouping is performed. If the sequence terminates with a 0, the last group size is repeatedly used.
	'thousands_sep'	组之间使用的字符。
LC_MONETARY	'int_curr_symbol'	国际货币符号。
	'currency_symbol'	当地货币符号。
	'p_cs_precedes/n_cs_precedes'	货币符号是否在值之前（对于正值或负值）。
	'p_sep_by_space/n_sep_by_space'	货币符号是否通过空格与值分隔（对于正值或负值）。
	'mon_decimal_point'	用于货币金额的小数点。
	'frac_digits'	货币值的本地格式中使用的小数位数。
	'int_frac_digits'	货币价值的国际格式中使用的小数位数。
	'mon_thousands_sep'	用于货币值的组分隔符。
	'mon_grouping'	相当于 'grouping'，用于货币价值。
	'positive_sign'	用于标注正货币价值的符号。
	'negative_sign'	用于注释负货币价值的符号。
	'p_sign_posn/n_sign_posn'	符号的位置（对于正值或负值），见下文。

可以将所有数值设置为*CHAR_MAX*，以指示此语言环境中未指定任何值。

下面给出了 'p_sign_posn' 和 'n_sign_posn' 的可能值。

值	解释
0	被括号括起来的货币和金额。
1	该标志应位于值和货币符号之前。
2	该标志应位于值和货币符号之后。
3	标志应该紧跟在值之前。
4	标志应该紧跟值项。
CHAR_MAX	此语言环境中未指定任何内容。

locale.**nl_langinfo**(*option*)

Return some locale-specific information as a string. This function is not available on all systems, and the set of possible options might also vary across platforms. The possible argument values are numbers, for which symbolic constants are available in the locale module.

The *nl_langinfo()* function accepts one of the following keys. Most descriptions are taken from the corresponding description in the GNU C library.

locale.**CODESET**

Get a string with the name of the character encoding used in the selected locale.

locale.**D_T_FMT**
Get a string that can be used as a format string for *time.strftime()* to represent date and time in a locale-specific way.

locale.**D_FMT**
Get a string that can be used as a format string for *time.strftime()* to represent a date in a locale-specific way.

locale.**T_FMT**
Get a string that can be used as a format string for *time.strftime()* to represent a time in a locale-specific way.

locale.**T_FMT_AMPM**
Get a format string for *time.strftime()* to represent time in the am/pm format.

DAY_1 ... DAY_7
Get the name of the n-th day of the week.

注解： This follows the US convention of DAY_1 being Sunday, not the international convention (ISO 8601) that Monday is the first day of the week.

ABDAY_1 ... ABDAY_7
Get the abbreviated name of the n-th day of the week.

MON_1 ... MON_12
Get the name of the n-th month.

ABMON_1 ... ABMON_12
Get the abbreviated name of the n-th month.

locale.**RADIXCHAR**
Get the radix character (decimal dot, decimal comma, etc.).

locale.**THOUSEP**
Get the separator character for thousands (groups of three digits).

locale.**YESEXPR**
Get a regular expression that can be used with the regex function to recognize a positive response to a yes/no question.

注解： The expression is in the syntax suitable for the regex() function from the C library, which might differ from the syntax used in *re*.

---

locale.**NOEXPR**
Get a regular expression that can be used with the regex(3) function to recognize a negative response to a yes/no question.

locale.**CRNCYSTR**
Get the currency symbol, preceded by "-" if the symbol should appear before the value, "+" if the symbol should appear after the value, or "." if the symbol should replace the radix character.

locale.**ERA**
Get a string that represents the era used in the current locale.

Most locales do not define this value. An example of a locale which does define this value is the Japanese one. In Japan, the traditional representation of dates includes the name of the era corresponding to the then-emperor's reign.

Normally it should not be necessary to use this value directly. Specifying the `E` modifier in their format strings causes the *time.strftime()* function to use this information. The format of the returned string is not specified, and therefore you should not assume knowledge of it on different systems.

`locale.`**ERA_D_T_FMT**
Get a format string for *time.strftime()* to represent date and time in a locale-specific era-based way.

`locale.`**ERA_D_FMT**
Get a format string for *time.strftime()* to represent a date in a locale-specific era-based way.

`locale.`**ERA_T_FMT**
Get a format string for *time.strftime()* to represent a time in a locale-specific era-based way.

`locale.`**ALT_DIGITS**
Get a representation of up to 100 values used to represent the values 0 to 99.

`locale.`**getdefaultlocale**([*envvars*])
Tries to determine the default locale settings and returns them as a tuple of the form `(language code, encoding)`.

According to POSIX, a program which has not called `setlocale(LC_ALL, '')` runs using the portable `'C'` locale. Calling `setlocale(LC_ALL, '')` lets it use the default locale as defined by the `LANG` variable. Since we do not want to interfere with the current locale setting we thus emulate the behavior in the way described above.

To maintain compatibility with other platforms, not only the `LANG` variable is tested, but a list of variables given as envvars parameter. The first found to be defined will be used. *envvars* defaults to the search path used in GNU gettext; it must always contain the variable name `LANG`. The GNU gettext search path contains `'LANGUAGE'`, `'LC_ALL'`, `'LC_CTYPE'`, and `'LANG'`, in that order.

Except for the code `'C'`, the language code corresponds to **RFC 1766**. *language code* and *encoding* may be `None` if their values cannot be determined.

2.0 新版功能.

`locale.`**getlocale**([*category*])
Returns the current setting for the given locale category as sequence containing *language code*, *encoding*. *category* may be one of the `LC_*` values except *LC_ALL*. It defaults to *LC_CTYPE*.

Except for the code `'C'`, the language code corresponds to **RFC 1766**. *language code* and *encoding* may be `None` if their values cannot be determined.

2.0 新版功能.

`locale.`**getpreferredencoding**([*do_setlocale*])
Return the encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess.

On some systems, it is necessary to invoke *setlocale()* to obtain the user preferences, so this function is not thread-safe. If invoking setlocale is not necessary or desired, *do_setlocale* should be set to `False`.

2.3 新版功能.

`locale.`**normalize**(*localename*)
Returns a normalized locale code for the given locale name. The returned locale code is formatted for use with *setlocale()*. If normalization fails, the original name is returned unchanged.

If the given encoding is not known, the function defaults to the default encoding for the locale code just like *setlocale()*.

2.0 新版功能.

locale.**resetlocale**([*category*])
Sets the locale for *category* to the default setting.

The default setting is determined by calling *getdefaultlocale()*. *category* defaults to *LC_ALL*.

2.0 新版功能.

locale.**strcoll**(*string1*, *string2*)
Compares two strings according to the current *LC_COLLATE* setting. As any other compare function, returns a negative, or a positive value, or 0, depending on whether *string1* collates before or after *string2* or is equal to it.

locale.**strxfrm**(*string*)
Transforms a string to one that can be used for the built-in function *cmp()*, and still returns locale-aware results. This function can be used when the same string is compared repeatedly, e.g. when collating a sequence of strings.

locale.**format**(*format*, *val*[, *grouping*[, *monetary*]])
Formats a number *val* according to the current *LC_NUMERIC* setting. The format follows the conventions of the % operator. For floating point values, the decimal point is modified if appropriate. If *grouping* is true, also takes the grouping into account.

If *monetary* is true, the conversion uses monetary thousands separator and grouping strings.

Please note that this function will only work for exactly one %char specifier. For whole format strings, use *format_string()*.

在 2.5 版更改: Added the *monetary* parameter.

locale.**format_string**(*format*, *val*[, *grouping*])
Processes formatting specifiers as in `format % val`, but takes the current locale settings into account.

2.5 新版功能.

locale.**currency**(*val*[, *symbol*[, *grouping*[, *international*]]])
Formats a number *val* according to the current *LC_MONETARY* settings.

The returned string includes the currency symbol if *symbol* is true, which is the default. If *grouping* is true (which is not the default), grouping is done with the value. If *international* is true (which is not the default), the international currency symbol is used.

Note that this function will not work with the 'C' locale, so you have to set a locale via *setlocale()* first.

2.5 新版功能.

locale.**str**(*float*)
Formats a floating point number using the same format as the built-in function `str(float)`, but takes the decimal point into account.

locale.**atof**(*string*)
Converts a string to a floating point number, following the *LC_NUMERIC* settings.

locale.**atoi**(*string*)
Converts a string to an integer, following the *LC_NUMERIC* conventions.

locale.**LC_CTYPE**
Locale category for the character type functions. Depending on the settings of this category, the functions of module *string* dealing with case change their behaviour.

locale.**LC_COLLATE**
Locale category for sorting strings. The functions *strcoll()* and *strxfrm()* of the *locale* module are affected.

locale.**LC_TIME**
Locale category for the formatting of time. The function *time.strftime()* follows these conventions.

locale.**LC_MONETARY**
Locale category for formatting of monetary values. The available options are available from the *localeconv()* function.

locale.**LC_MESSAGES**
Locale category for message display. Python currently does not support application specific locale-aware messages. Messages displayed by the operating system, like those returned by *os.strerror()* might be affected by this category.

locale.**LC_NUMERIC**
Locale category for formatting numbers. The functions *format()*, *atoi()*, *atof()* and *str()* of the *locale* module are affected by that category. All other numeric formatting operations are not affected.

locale.**LC_ALL**
Combination of all locale settings. If this flag is used when the locale is changed, setting the locale for all categories is attempted. If that fails for any category, no category is changed at all. When the locale is retrieved using this flag, a string indicating the setting for all categories is returned. This string can be later used to restore the settings.

locale.**CHAR_MAX**
This is a symbolic constant used for different values returned by *localeconv()*.

示例:

```
>>> import locale
>>> loc = locale.getlocale()  # get current locale
# use German locale; name might vary with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xe4n', 'foo')  # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '')    # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C')   # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc)   # restore saved locale
```

22.2.1 Background, details, hints, tips and caveats

The C standard defines the locale as a program-wide property that may be relatively expensive to change. On top of that, some implementation are broken in such a way that frequent locale changes may cause core dumps. This makes the locale somewhat painful to use correctly.

Initially, when a program is started, the locale is the `C` locale, no matter what the user's preferred locale is. The program must explicitly say that it wants the user's preferred locale settings by calling `setlocale(LC_ALL, '')`.

It is generally a bad idea to call *setlocale()* in some library routine, since as a side effect it affects the entire program. Saving and restoring it is almost as bad: it is expensive and affects other threads that happen to run before the settings have been restored.

If, when coding a module for general use, you need a locale independent version of an operation that is affected by the locale (such as *string.lower()*, or certain formats used with *time.strftime()*), you will have to find a way to do it without using the standard library routine. Even better is convincing yourself that using locale settings is okay. Only as a last resort should you document that your module is not compatible with non-`C` locale settings.

The case conversion functions in the *string* module are affected by the locale settings. When a call to the *setlocale()* function changes the *LC_CTYPE* settings, the variables `string.lowercase`, `string.uppercase` and `string.letters` are recalculated. Note that code that uses these variable through '`from ... import ...`', e.g. `from string import letters`, is not affected by subsequent *setlocale()* calls.

The only way to perform numeric operations according to the locale is to use the special functions defined by this module: *atof()*, *atoi()*, *format()*, *str()*.

22.2.2 For extension writers and programs that embed Python

Extension modules should never call *setlocale()*, except to find out what the current locale is. But since the return value can only be used portably to restore it, that is not very useful (except perhaps to find out whether or not the locale is C).

When Python code uses the *locale* module to change the locale, this also affects the embedding application. If the embedding application doesn't want this to happen, it should remove the _locale extension module (which does all the work) from the table of built-in modules in the config.c file, and make sure that the _locale module is not accessible as a shared library.

22.2.3 Access to message catalogs

locale.**gettext**(*msg*)

locale.**dgettext**(*domain*, *msg*)

locale.**dcgettext**(*domain*, *msg*, *category*)

locale.**textdomain**(*domain*)

locale.**bindtextdomain**(*domain*, *dir*)

The locale module exposes the C library's gettext interface on systems that provide this interface. It consists of the functions gettext(), dgettext(), dcgettext(), textdomain(), bindtextdomain(), and bind_textdomain_codeset(). These are similar to the same functions in the *gettext* module, but use the C library's binary format for message catalogs, and the C library's search algorithms for locating message catalogs.

Python applications should normally find no need to invoke these functions, and should use *gettext* instead. A known exception to this rule are applications that link with additional C libraries which internally invoke gettext() or dcgettext(). For these applications, it may be necessary to bind the text domain, so that the libraries can properly locate their message catalogs.

CHAPTER 23

程序框架

本章中描述的模块是很大程度上决定程序结构的框架。目前，这里描述的模块都面向编写命令行接口。

本章描述的完整模块列表如下：

23.1 cmd —支持面向行的命令解释器

源代码: Lib/cmd.py

Cmd 类提供简单框架用于编写面向行的命令解释器。这些通常对测试工具，管理工具和原型有用，这些工具随后将被包含在更复杂的接口中。

class cmd.**Cmd**([*completekey*[, *stdin*[, *stdout*]]])

一个*Cmd* 实例或子类实例是面向行的解释器框架结构。实例化*Cmd* 本身是没有充分理由的，它作为自定义解释器类的超类是非常有用的为了继承*Cmd* 的方法并且封装动作方法。

可选参数 *completekey* 是完成键的*readline* 名称；默认是 Tab 。如果 *completekey* 不是*None* 并且*readline* 是可用的，命令完成会自动完成。

可选参数 *stdin* 和 *stdout* 指定了 Cmd 实例或子类实例将用于输入和输出的输入和输出文件对象。如果没有指定，他们将默认为*sys.stdin* 和*sys.stdout* 。

如果你想要使用一个给定的 *stdin* ，确保将实例的*use_rawinput* 属性设置为 False ，否则 *stdin* 将被忽略。

在 2.3 版更改: The *stdin* and *stdout* parameters were added.

23.1.1 Cmd 对象

Cmd 实例有下列方法：

Cmd.**cmdloop**([*intro*])
反复发出提示，接受输入，从收到的输入中解析出一个初始前缀，并分派给操作方法，将其余的行作为参数传递给它们。

可选参数是在第一个提示之前发布的横幅或介绍字符串（这将覆盖*intro*类属性）。

如果*readline*继承模块被加载，输入将自动继承类似 **bash** 的历史列表编辑（例如，Control-P 滚动回到最后一个命令，Control-N 转到下一个命令，以 Control-F 非破坏性的方式向右 Control-B 移动光标，破坏性地等）。

输入的文件结束符被作为字符串传回 `'EOF'` 。

解释器实例将会识别命令名称 `foo` 当且仅当它有方法 `do_foo()` 。有一个特殊情况，分派始于字符 `'?'` 的行到方法 `do_help()` 。另一种特殊情况，分派始于字符 `'!'` 的行到方法 `do_shell()` （如果定义了这个方法）

这个方法将返回当*postcmd()*方法返回一个真值。参数 *stop* 到*postcmd()*是命令对应的返回值 `do_*()` 的方法。

如果激活了完成，全部命令将会自动完成，并且通过调用 `complete_foo()` 参数 *text* , *line*, *begidx* , 和 *endidx* 完成全部命令参数。*text* 是我们试图匹配的字符串前缀，所有返回的匹配项必须以它为开头。*line* 是删除了前导空格的当前的输入行，*begidx* 和 *endidx* 是前缀文本的开始和结束索引。，可以用于根据参数位置提供不同的完成。

所有*Cmd* 的子类继承一个预定义 `do_help()` 。这个方法使用参数 `'bar'` 调用，调用对应的方法 `help_bar()` ，如果不存在，打印 `do_bar()` 的文档字符串，如果可用。没有参数的情况下，`do_help()` 方法会列出所有可用的帮助主题（即所有具有相应的 `help_*()` 方法或命令的文档字符串），也会列举所有未被记录的命令。

Cmd.**onecmd**(*str*)
解释该参数，就好像它是为响应提示而键入的一样。这可能会被覆盖，但通常不应该被覆盖; 请参阅: *precmd()* 和*postcmd()* 方法，用于执行有用的挂钩。返回值是一个标志，指示解释器对命令的解释是否应该停止。如果命令 *str* 有一个 `do_*()` 方法，则返回该方法的返回值，否则返回*default()* 方法的返回值。

Cmd.**emptyline**()
在响应提示输入空行时调用的方法。如果此方法未被覆盖，则重复输入的最后一个非空命令。

Cmd.**default**(*line*)
当命令前缀不能被识别的时候在输入行调用的方法。如果此方法未被覆盖，它将输出一个错误信息并返回。

Cmd.**completedefault**(*text*, *line*, *begidx*, *endidx*)
当没有特定于命令的 `complete_*()` 方法可用时，调用此方法完成输入行。默认情况下，它返回一个空列表。

Cmd.**precmd**(*line*)
钩方法在命令行 *line* 被解释之前执行，但是在输入提示被生成和发出后。这个方法是一个在*Cmd* 中的存根；它的存在是为了被子类覆盖。返回值被用作*onecmd()* 方法执行的命令；*precmd()* 的实现或许会重写命令或者简单的返回 *line* 不变。

Cmd.**postcmd**(*stop*, *line*)
钩方法只在命令调度完成后执行。这个方法是一个在*Cmd* 中的存根；它的存在是为了子类被覆盖。*line* 是被执行的命令行，*stop* 是一个表示在调用*postcmd()* 之后是否终止执行的标志；这将作为*onecmd()* 方法的返回值。这个方法的返回值被用作与 *stop* 相关联的内部标志的新值；返回 false 将导致解释继续。

Cmd.**preloop**()
钩方法当*cmdloop()* 被调用时执行一次。方法是一个在*Cmd* 中的存根；它的存在是为了被子类覆盖。

Cmd.**postloop**()
钩方法在*cmdloop()* 即将返回时执行一次。这个方法是一个在*Cmd* 中的存根；塔顶存在是为了被子类覆盖。

Instances of *Cmd* subclasses have some public instance variables:

Cmd.**prompt**
发出提示以请求输入。

Cmd.**identchars**
接受命令前缀的字符串。

Cmd.**lastcmd**
看到最后一个非空命令前缀。

Cmd.**cmdqueue**
排队的输入行列表。当需要新的输入时，在*cmdloop()* 中检查 cmdqueue 列表；如果它不是空的，它的元素将被按顺序处理，就像在提示符处输入一样。

Cmd.**intro**
要作为简介或横幅发出的字符串。可以通过给*cmdloop()* 方法一个参数来覆盖它。

Cmd.**doc_header**
如果帮助输出具有记录命令的段落，则发出头文件。

Cmd.**misc_header**
如果帮助输出其他帮助主题的部分（即与 `do_*()` 方法没有关联的 `help_*()` 方法），则发出头文件。

Cmd.**undoc_header**
如果帮助输出未被记录命令的部分（即与 `help_*()` 方法没有关联的 `do_*()` 方法），则发出头文件。

Cmd.**ruler**
用于在帮助信息标题的下方绘制分隔符的字符，如果为空，则不绘制标尺线。这个字符默认是 `'='` 。

Cmd.**use_rawinput**
A flag, defaulting to true. If true, *cmdloop()* uses *raw_input()* to display a prompt and read the next command; if false, `sys.stdout.write()` and `sys.stdin.readline()` are used. (This means that by importing *readline*, on systems that support it, the interpreter will automatically support **Emacs**-like line editing and command-history keystrokes.)

23.2 `shlex` —Simple lexical analysis

1.5.2 新版功能.

Source code: Lib/shlex.py

The *shlex* class makes it easy to write lexical analyzers for simple syntaxes resembling that of the Unix shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

Prior to Python 2.7.3, this module did not support Unicode input.

The *shlex* module defines the following functions:

shlex.**split**(*s*[, *comments*[, *posix*]])
Split the string *s* using shell-like syntax. If *comments* is `False` (the default), the parsing of comments in the given string will be disabled (setting the `commenters` attribute of the `shlex` instance to the empty string). This function operates in POSIX mode by default, but uses non-POSIX mode if the *posix* argument is false.

2.3 新版功能.

在 2.6 版更改: Added the *posix* parameter.

注解: Since the `split()` function instantiates a `shlex` instance, passing `None` for *s* will read the string to split from standard input.

The `shlex` module defines the following class:

class shlex.**shlex**([*instream*[, *infile*[, *posix*]]])
A `shlex` instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with `read()` and `readline()` methods, or a string (strings are accepted since Python 2.3). If no argument is given, input will be taken from `sys.stdin`. The second optional argument is a filename string, which sets the initial value of the `infile` attribute. If the *instream* argument is omitted or equal to `sys.stdin`, this second argument defaults to "stdin". The *posix* argument was introduced in Python 2.3, and defines the operational mode. When *posix* is not true (default), the `shlex` instance will operate in compatibility mode. When operating in POSIX mode, `shlex` will try to be as close as possible to the POSIX shell parsing rules.

参见:

Module `ConfigParser` Parser for configuration files similar to the Windows `.ini` files.

23.2.1 shlex Objects

A `shlex` instance has the following methods:

shlex.**get_token**()
Return a token. If tokens have been stacked using `push_token()`, pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, `eof` is returned (the empty string (`''`) in non-POSIX mode, and `None` in POSIX mode).

shlex.**push_token**(*str*)
Push the argument onto the token stack.

shlex.**read_token**()
Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

shlex.**sourcehook**(*filename*)
When `shlex` detects a source request (see `source` below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as `sys.stdin`), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles `#include "file.h"`).

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with `open()` called on it to yield the second component. (Note: this is the reverse of the order of arguments in instance initialization!)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding 'close' hook, but a shlex instance will call the *close()* method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the *push_source()* and *pop_source()* methods.

shlex.**push_source**(*stream*[, *filename*])
Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the *sourcehook()* method.

2.1 新版功能.

shlex.**pop_source**()
Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream.

2.1 新版功能.

shlex.**error_leader**([*file*[, *line*]])
This method generates an error message leader in the format of a Unix C compiler error label; the format is `'"%s", line %d: '`, where the `%s` is replaced with the name of the current source file and the `%d` with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage *shlex* users to generate error messages in the standard, parseable format understood by Emacs and other Unix tools.

Instances of *shlex* subclasses have some public instance variables which either control lexical analysis or can be used for debugging:

shlex.**commenters**
The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just `'#'` by default.

shlex.**wordchars**
The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore.

shlex.**whitespace**
Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

shlex.**escape**
Characters that will be considered as escape. This will be only used in POSIX mode, and includes just `'\'` by default.

2.3 新版功能.

shlex.**quotes**
Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

shlex.**escapedquotes**
Characters in *quotes* that will interpret escape characters defined in *escape*. This is only used in POSIX mode, and includes just `'"'` by default.

2.3 新版功能.

shlex.**whitespace_split**
If `True`, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with *shlex*, getting tokens in a similar way to shell arguments.

2.3 新版功能.

shlex.**infile**
The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

shlex.**instream**
The input stream from which this *shlex* instance is reading characters.

shlex.**source**
This attribute is None by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the source keyword in various shells. That is, the immediately following token will be opened as a filename and input will be taken from that stream until EOF, at which point the *close()* method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

shlex.**debug**
If this attribute is numeric and 1 or more, a *shlex* instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

shlex.**lineno**
Source line number (count of newlines seen so far plus one).

shlex.**token**
The token buffer. It may be useful to examine this when catching exceptions.

shlex.**eof**
Token used to determine end of file. This will be set to the empty string (''), in non-POSIX mode, and to None in POSIX mode.

2.3 新版功能.

23.2.2 Parsing Rules

When operating in non-POSIX mode, *shlex* will try to obey to the following rules.

- Quote characters are not recognized within words (Do"Not"Separate is parsed as the single word Do"Not"Separate);
- Escape characters are not recognized;
- Enclosing characters in quotes preserve the literal value of all characters within the quotes;
- Closing quotes separate words ("Do"Separate is parsed as "Do" and Separate);
- If *whitespace_split* is False, any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token. If it is True, *shlex* will only split words in whitespaces;
- EOF is signaled with an empty string ('');
- It's not possible to parse empty strings, even if quoted.

When operating in POSIX mode, *shlex* will try to obey to the following parsing rules.

- Quotes are stripped out, and do not separate words ("Do"Not"Separate" is parsed as the single word DoNotSeparate);
- Non-quoted escape characters (e.g. '\') preserve the literal value of the next character that follows;
- Enclosing characters in quotes which are not part of *escapedquotes* (e.g. "'") preserve the literal value of all characters within the quotes;
- Enclosing characters in quotes which are part of *escapedquotes* (e.g. '"') preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in *escape*. The escape characters

retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.

- EOF is signaled with a *None* value;
- Quoted empty strings (`''`) are allowed;

CHAPTER 24

Tk 图形用户界面 (GUI)

Tk/Tcl has long been an integral part of Python. It provides a robust and platform independent windowing toolkit, that is available to Python programmers using the *Tkinter* module, and its extensions, the *Tix* and the *ttk* modules.

The *Tkinter* module is a thin object-oriented layer on top of Tcl/Tk. To use *Tkinter*, you don't need to write Tcl code, but you will need to consult the Tk documentation, and occasionally the Tcl documentation. *Tkinter* is a set of wrappers that implement the Tk widgets as Python classes. In addition, the internal module `_tkinter` provides a threadsafe mechanism which allows Python and Tcl to interact.

Tkinter's chief virtues are that it is fast, and that it usually comes bundled with Python. Although its standard documentation is weak, good material is available, which includes: references, tutorials, a book and others. *Tkinter* is also famous for having an outdated look and feel, which has been vastly improved in Tk 8.5. Nevertheless, there are many other GUI libraries that you could be interested in. For more information about alternatives, see the 其他图形用户界面（*GUI*）包 section.

24.1 `Tkinter` —Python interface to Tcl/Tk

The *Tkinter* module ("Tk interface") is the standard Python interface to the Tk GUI toolkit. Both Tk and *Tkinter* are available on most Unix platforms, as well as on Windows systems. (Tk itself is not part of Python; it is maintained at ActiveState.)

Running `python -m Tkinter` from the command line should open a window demonstrating a simple Tk interface, letting you know that *Tkinter* is properly installed on your system, and also showing what version of Tcl/Tk is installed, so you can read the Tcl/Tk documentation specific to that version.

注解： *Tkinter* has been renamed to `tkinter` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

参见：

Tkinter 文档：

Python Tkinter 资源 The Python Tkinter Topic Guide 提供了在 Python 中使用 Tk 的很多信息，同时包含了 Tk 其他信息的链接。

TKDocs 大量的教程，部分可视化组件的介绍说明。

Tkinter 8.5 reference: a GUI for Python 在线参考资料。

Tkinter docs from effbot effbot.org 提供的 tkinter 在线参考资料。

使用 Python 编程 由 Mark Lutz 所著的书籍，对 Tkinter 进行了完美的介绍。

为繁忙的 Python 开发者所准备的现代 Tkinter 由 Mark Rozerman 所著的关于如何使用 Python 和 Tkinter 来搭建有吸引力的和现代化的图形用户界面的书籍

Python 和 Tkinter 编程 作者：John Grayson (ISBN 1-884777-81-3).

Tcl/Tk 文档:

Tk 命令 Most commands are available as *Tkinter* or `Tkinter.ttk` classes. Change '8.6' to match the version of your Tcl/Tk installation.

Tcl/Tk 最新手册页面 www.tcl.tk 上面最新的 Tcl/Tk 手册。

ActiveState Tcl Home Page Tk/Tcl 的多数开发工作发生在 ActiveState 。

Tcl 及 Tk 工具集 由 Tcl 发明者 John Ousterhout 所著的书籍。

'Tcl 和 Tk 编程实战 <http://www.beedub.com/book/>'_ Brent Welch 所著的百科全局式书籍。

24.1.1 Tkinter 模块

Most of the time, the *Tkinter* module is all you really need, but a number of additional modules are available as well. The Tk interface is located in a binary module named `_tkinter`. This module contains the low-level interface to Tk, and should never be used directly by application programmers. It is usually a shared library (or DLL), but might in some cases be statically linked with the Python interpreter.

In addition to the Tk interface module, *Tkinter* includes a number of Python modules. The two most important modules are the *Tkinter* module itself, and a module called `Tkconstants`. The former automatically imports the latter, so to use Tkinter, all you need to do is to import one module:

```
import Tkinter
```

或者更常用的:

```
from Tkinter import *
```

class `Tkinter.`**Tk**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=1*)

The *Tk* class is instantiated without arguments. This creates a toplevel widget of Tk which usually is the main window of an application. Each instance has its own associated Tcl interpreter.

在 2.4 版更改: The *useTk* parameter was added.

`Tkinter.`**Tcl**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=0*)

The *Tcl()* function is a factory function which creates an object much like that created by the *Tk* class, except that it does not initialize the Tk subsystem. This is most often useful when driving the Tcl interpreter in an environment where one doesn't want to create extraneous toplevel windows, or where one cannot (such as Unix/Linux systems without an X server). An object created by the *Tcl()* object can have a Toplevel window created (and the Tk subsystem initialized) by calling its `loadtk()` method.

2.4 新版功能.

Other modules that provide Tk support include:

ScrolledText Text widget with a vertical scroll bar built in.

tkColorChooser Dialog to let the user choose a color.

tkCommonDialog Base class for the dialogs defined in the other modules listed here.

tkFileDialog Common dialogs to allow the user to specify a file to open or save.

tkFont Utilities to help work with fonts.

tkMessageBox Access to standard Tk dialog boxes.

tkSimpleDialog Basic dialogs and convenience functions.

Tkdnd Drag-and-drop support for *Tkinter*. This is experimental and should become deprecated when it is replaced with the Tk DND.

turtle Turtle graphics in a Tk window.

These have been renamed as well in Python 3; they were all made submodules of the new `tkinter` package.

24.1.2 Tkinter Life Preserver

This section is not designed to be an exhaustive tutorial on either Tk or Tkinter. Rather, it is intended as a stop gap, providing some introductory orientation on the system.

Credits:

- Tkinter was written by Steen Lumholt and Guido van Rossum.
- Tk was written by John Ousterhout while at Berkeley.
- This Life Preserver was written by Matt Conway at the University of Virginia.
- The html rendering, and some liberal editing, was produced from a FrameMaker version by Ken Manheimer.
- Fredrik Lundh elaborated and revised the class interface descriptions, to get them current with Tk 4.2.
- Mike Clarkson converted the documentation to LaTeX, and compiled the User Interface chapter of the reference manual.

How To Use This Section

This section is designed in two parts: the first half (roughly) covers background material, while the second half can be taken to the keyboard as a handy reference.

When trying to answer questions of the form "how do I do blah", it is often best to find out how to do "blah" in straight Tk, and then convert this back into the corresponding *Tkinter* call. Python programmers can often guess at the correct Python command by looking at the Tk documentation. This means that in order to use Tkinter, you will have to know a little bit about Tk. This document can't fulfill that role, so the best we can do is point you to the best documentation that exists. Here are some hints:

- The authors strongly suggest getting a copy of the Tk man pages. Specifically, the man pages in the `mann` directory are most useful. The `man3` man pages describe the C interface to the Tk library and thus are not especially helpful for script writers.
- Addison-Wesley publishes a book called Tcl and the Tk Toolkit by John Ousterhout (ISBN 0-201-63337-X) which is a good introduction to Tcl and Tk for the novice. The book is not exhaustive, and for many details it defers to the man pages.
- `Tkinter.py` is a last resort for most, but can be a good place to go when nothing else makes sense.

A Simple Hello World Program

```
from Tkinter import *

class Application(Frame):
    def say_hi(self):
        print "hi there, everyone!"

    def createWidgets(self):
        self.QUIT = Button(self)
        self.QUIT["text"] = "QUIT"
        self.QUIT["fg"]   = "red"
        self.QUIT["command"] =  self.quit

        self.QUIT.pack({"side": "left"})

        self.hi_there = Button(self)
        self.hi_there["text"] = "Hello",
        self.hi_there["command"] = self.say_hi

        self.hi_there.pack({"side": "left"})

    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

root = Tk()
app = Application(master=root)
app.mainloop()
root.destroy()
```

24.1.3 A (Very) Quick Look at Tcl/Tk

The class hierarchy looks complicated, but in actual practice, application programmers almost always refer to the classes at the very bottom of the hierarchy.

注释:

- These classes are provided for the purposes of organizing certain functions under one namespace. They aren't meant to be instantiated independently.
- The *Tk* class is meant to be instantiated only once in an application. Application programmers need not instantiate one explicitly, the system creates one whenever any of the other classes are instantiated.
- The `Widget` class is not meant to be instantiated, it is meant only for subclassing to make "real" widgets (in C++, this is called an 'abstract class').

To make use of this reference material, there will be times when you will need to know how to read short passages of Tk and how to identify the various parts of a Tk command. (See section *Mapping Basic Tk into Tkinter* for the *Tkinter* equivalents of what's below.)

Tk scripts are Tcl programs. Like all Tcl programs, Tk scripts are just lists of tokens separated by spaces. A Tk widget is just its *class*, the *options* that help configure it, and the *actions* that make it do useful things.

To make a widget in Tk, the command is always of the form:

```
classCommand newPathname options
```

classCommand denotes which kind of widget to make (a button, a label, a menu…)

newPathname is the new name for this widget. All names in Tk must be unique. To help enforce this, widgets in Tk are named with *pathnames*, just like files in a file system. The top level widget, the *root*, is called `.` (period) and children are delimited by more periods. For example, `.myApp.controlPanel.okButton` might be the name of a widget.

options configure the widget's appearance and in some cases, its behavior. The options come in the form of a list of flags and values. Flags are preceded by a '-', like Unix shell command flags, and values are put in quotes if they are more than one word.

例如

```
button   .fred   -fg red -text "hi there"
   ^       ^     \_____________________/
   |       |                |
 class    new            options
command  widget  (-opt val -opt val ...)
```

Once created, the pathname to the widget becomes a new command. This new *widget command* is the programmer's handle for getting the new widget to perform some *action*. In C, you'd express this as someAction(fred, someOptions), in C++, you would express this as fred.someAction(someOptions), and in Tk, you say:

```
.fred someAction someOptions
```

Note that the object name, `.fred`, starts with a dot.

As you'd expect, the legal values for *someAction* will depend on the widget's class: `.fred disable` works if fred is a button (fred gets greyed out), but does not work if fred is a label (disabling of labels is not supported in Tk).

The legal values of *someOptions* is action dependent. Some actions, like `disable`, require no arguments, others, like a text-entry box's `delete` command, would need arguments to specify what range of text to delete.

24.1.4 Mapping Basic Tk into Tkinter

Class commands in Tk correspond to class constructors in Tkinter.

```
button .fred                =====>  fred = Button()
```

The master of an object is implicit in the new name given to it at creation time. In Tkinter, masters are specified explicitly.

```
button .panel.fred          =====>  fred = Button(panel)
```

The configuration options in Tk are given in lists of hyphened tags followed by values. In Tkinter, options are specified as keyword-arguments in the instance constructor, and keyword-args for configure calls or as instance indices, in dictionary style, for established instances. See section *Setting Options* on setting options.

```
button .fred -fg red        =====>  fred = Button(panel, fg = "red")
.fred configure -fg red     =====>  fred["fg"] = red
                            OR ==>  fred.config(fg = "red")
```

In Tk, to perform an action on a widget, use the widget name as a command, and follow it with an action name, possibly with arguments (options). In Tkinter, you call methods on the class instance to invoke actions on the widget. The actions (methods) that a given widget can perform are listed in the Tkinter.py module.

```
.fred invoke                  =====>  fred.invoke()
```

To give a widget to the packer (geometry manager), you call pack with optional arguments. In Tkinter, the Pack class holds all this functionality, and the various forms of the pack command are implemented as methods. All widgets in *Tkinter* are subclassed from the Packer, and so inherit all the packing methods. See the *Tix* module documentation for additional information on the Form geometry manager.

```
pack .fred -side left         =====>  fred.pack(side = "left")
```

24.1.5 How Tk and Tkinter are Related

From the top down:

Your App Here (Python) A Python application makes a *Tkinter* call.

Tkinter (Python Module) This call (say, for example, creating a button widget), is implemented in the *Tkinter* module, which is written in Python. This Python function will parse the commands and the arguments and convert them into a form that makes them look as if they had come from a Tk script instead of a Python script.

tkinter (C) These commands and their arguments will be passed to a C function in the *tkinter* - note the lowercase - extension module.

Tk Widgets (C and Tcl) This C function is able to make calls into other C modules, including the C functions that make up the Tk library. Tk is implemented in C and some Tcl. The Tcl part of the Tk widgets is used to bind certain default behaviors to widgets, and is executed once at the point where the Python *Tkinter* module is imported. (The user never sees this stage).

Tk (C) The Tk part of the Tk Widgets implement the final mapping to ···

Xlib (C) the Xlib library to draw graphics on the screen.

24.1.6 Handy Reference

Setting Options

Options control things like the color and border width of a widget. Options can be set in three ways:

At object creation time, using keyword arguments

```
fred = Button(self, fg = "red", bg = "blue")
```

After object creation, treating the option name like a dictionary index

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

Use the config() method to update multiple attrs subsequent to object creation

```
fred.config(fg = "red", bg = "blue")
```

For a complete explanation of a given option and its behavior, see the Tk man pages for the widget in question.

Note that the man pages list "STANDARD OPTIONS" and "WIDGET SPECIFIC OPTIONS" for each widget. The former is a list of options that are common to many widgets, the latter are the options that are idiosyncratic to that particular widget. The Standard Options are documented on the *options(3)* man page.

No distinction between standard and widget-specific options is made in this document. Some options don't apply to some kinds of widgets. Whether a given widget responds to a particular option depends on the class of the widget; buttons have a `command` option, labels do not.

The options supported by a given widget are listed in that widget's man page, or can be queried at runtime by calling the `config()` method without arguments, or by calling the `keys()` method on that widget. The return value of these calls is a dictionary whose key is the name of the option as a string (for example, `'relief'`) and whose values are 5-tuples.

Some options, like `bg` are synonyms for common options with long names (`bg` is shorthand for "background"). Passing the `config()` method the name of a shorthand option will return a 2-tuple, not 5-tuple. The 2-tuple passed back will contain the name of the synonym and the "real" option (such as `('bg', 'background')`).

索引	含义	示例
0	选项名称	`'relief'`
1	数据库查找的选项名称	`'relief'`
2	数据库查找的选项类	`'Relief'`
3	默认值	`'raised'`
4	当前值	`'groove'`

示例:

```
>>> print fred.config()
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

Of course, the dictionary printed will include all the options available and their values. This is meant only as an example.

The Packer

The packer is one of Tk's geometry-management mechanisms. Geometry managers are used to specify the relative positioning of the positioning of widgets within their container - their mutual *master*. In contrast to the more cumbersome *placer* (which is used less commonly, and we do not cover here), the packer takes qualitative relationship specification - *above*, *to the left of*, *filling*, etc - and works everything out to determine the exact placement coordinates for you.

The size of any *master* widget is determined by the size of the "slave widgets" inside. The packer is used to control where slave widgets appear inside the master into which they are packed. You can pack widgets into frames, and frames into other frames, in order to achieve the kind of layout you desire. Additionally, the arrangement is dynamically adjusted to accommodate incremental changes to the configuration, once it is packed.

Note that widgets do not appear until they have had their geometry specified with a geometry manager. It's a common early mistake to leave out the geometry specification, and then be surprised when the widget is created but nothing appears. A widget will appear only after it has had, for example, the packer's `pack()` method applied to it.

The pack() method can be called with keyword-option/value pairs that control where the widget is to appear within its container, and how it is to behave when the main application window is resized. Here are some examples:

```
fred.pack()                     # defaults to side = "top"
fred.pack(side = "left")
fred.pack(expand = 1)
```

Packer Options

For more extensive information on the packer and the options that it can take, see the man pages and page 183 of John Ousterhout's book.

anchor Anchor type. Denotes where the packer is to place each slave in its parcel.

expand Boolean, `0` or `1`.

fill Legal values: `'x'`, `'y'`, `'both'`, `'none'`.

ipadx and ipady A distance - designating internal padding on each side of the slave widget.

padx and pady A distance - designating external padding on each side of the slave widget.

side Legal values are: `'left'`, `'right'`, `'top'`, `'bottom'`.

Coupling Widget Variables

The current-value setting of some widgets (like text entry widgets) can be connected directly to application variables by using special options. These options are `variable`, `textvariable`, `onvalue`, `offvalue`, and `value`. This connection works both ways: if the variable changes for any reason, the widget it's connected to will be updated to reflect the new value.

Unfortunately, in the current implementation of *Tkinter* it is not possible to hand over an arbitrary Python variable to a widget through a `variable` or `textvariable` option. The only kinds of variables for which this works are variables that are subclassed from a class called Variable, defined in the *Tkinter* module.

There are many useful subclasses of Variable already defined: `StringVar`, `IntVar`, `DoubleVar`, and `BooleanVar`. To read the current value of such a variable, call the `get()` method on it, and to change its value you call the `set()` method. If you follow this protocol, the widget will always track the value of the variable, with no further intervention on your part.

例如

```
class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.pack()

        # here is the application variable
        self.contents = StringVar()
        # set it to some value
        self.contents.set("this is a variable")
        # tell the entry widget to watch this variable
        self.entrythingy["textvariable"] = self.contents

        # and here we get a callback when the user hits return.
        # we will have the program print out the value of the
        # application variable when the user hits return
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print "hi. contents of entry is now ---->", \
              self.contents.get()
```

The Window Manager

In Tk, there is a utility command, `wm`, for interacting with the window manager. Options to the `wm` command allow you to control things like titles, placement, icon bitmaps, and the like. In *Tkinter*, these commands have been implemented as methods on the `Wm` class. Toplevel widgets are subclassed from the `Wm` class, and so can call the `Wm` methods directly.

To get at the toplevel window that contains a given widget, you can often just refer to the widget's master. Of course if the widget has been packed inside of a frame, the master won't represent a toplevel window. To get at the toplevel window that contains an arbitrary widget, you can call the `_root()` method. This method begins with an underscore to denote the fact that this function is part of the implementation, and not an interface to Tk functionality.

Here are some examples of typical usage:

```
from Tkinter import *
class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()


# create the application
myapp = App()

#
# here are method calls to the window manager class
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# start the program
myapp.mainloop()
```

Tk Option Data Types

anchor Legal values are points of the compass: `"n"`, `"ne"`, `"e"`, `"se"`, `"s"`, `"sw"`, `"w"`, `"nw"`, and also `"center"`.

bitmap There are eight built-in, named bitmaps: `'error'`, `'gray25'`, `'gray50'`, `'hourglass'`, `'info'`, `'questhead'`, `'question'`, `'warning'`. To specify an X bitmap filename, give the full path to the file, preceded with an `@`, as in `"@/usr/contrib/bitmap/gumby.bit"`.

boolean You can pass integers 0 or 1 or the strings `"yes"` or `"no"`.

callback This is any Python function that takes no arguments. For example:

```
def print_it():
        print "hi there"
fred["command"] = print_it
```

color Colors can be given as the names of X colors in the rgb.txt file, or as strings representing RGB values in 4 bit: `"#RGB"`, 8 bit: `"#RRGGBB"`, 12 bit" `"#RRRGGGBBB"`, or 16 bit `"#RRRRGGGGBBBB"` ranges, where R,G,B here represent any legal hex digit. See page 160 of Ousterhout's book for details.

cursor The standard X cursor names from `cursorfont.h` can be used, without the `XC_` prefix. For example to get a hand cursor (`XC_hand2`), use the string `"hand2"`. You can also specify a bitmap and mask file of your own. See page 179 of Ousterhout's book.

distance Screen distances can be specified in either pixels or absolute distances. Pixels are given as numbers and absolute distances as strings, with the trailing character denoting units: `c` for centimetres, `i` for inches, `m` for millimetres, `p` for printer's points. For example, 3.5 inches is expressed as `"3.5i"`.

font Tk uses a list font name format, such as `{courier 10 bold}`. Font sizes with positive numbers are measured in points; sizes with negative numbers are measured in pixels.

geometry This is a string of the form `widthxheight`, where width and height are measured in pixels for most widgets (in characters for widgets displaying text). For example: `fred["geometry"] = "200x100"`.

justify Legal values are the strings: `"left"`, `"center"`, `"right"`, and `"fill"`.

region This is a string with four space-delimited elements, each of which is a legal distance (see above). For example: `"2 3 4 5"` and `"3i 2i 4.5i 2i"` and `"3c 2c 4c 10.43c"` are all legal regions.

relief Determines what the border style of a widget will be. Legal values are: `"raised"`, `"sunken"`, `"flat"`, `"groove"`, and `"ridge"`.

scrollcommand This is almost always the `set()` method of some scrollbar widget, but can be any widget method that takes a single argument. Refer to the file `Demo/tkinter/matt/canvas-with-scrollbars.py` in the Python source distribution for an example.

wrap: Must be one of: `"none"`, `"char"`, or `"word"`.

Bindings and Events

The bind method from the widget command allows you to watch for certain events and to have a callback function trigger when that event type occurs. The form of the bind method is:

```
def bind(self, sequence, func, add=''):
```

where:

sequence –序列 is a string that denotes the target kind of event. (See the bind man page and page 201 of John Ousterhout's book for details).

func is a Python function, taking one argument, to be invoked when the event occurs. An Event instance will be passed as the argument. (Functions deployed this way are commonly known as *callbacks*.)

add is optional, either `''` or `'+'`. Passing an empty string denotes that this binding is to replace any other bindings that this event is associated with. Passing a `'+'` means that this function is to be added to the list of functions bound to this event type.

例如

```
def turnRed(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turnRed)
```

Notice how the widget field of the event is being accessed in the `turnRed()` callback. This field contains the widget that caught the X event. The following table lists the other event fields you can access, and how they are denoted in Tk, which can be useful when referring to the Tk man pages.

```
Tk      Tkinter Event Field             Tk      Tkinter Event Field
--      -------------------             --      -------------------
%f      focus                           %A      char
%h      height                          %E      send_event
%k      keycode                         %K      keysym
```

(下页继续)

(续上页)

%s	state	%N	keysym_num
%t	time	%T	type
%w	width	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

The index Parameter

A number of widgets require" index" parameters to be passed. These are used to point at a specific place in a Text widget, or to particular characters in an Entry widget, or to particular menu items in a Menu widget.

Entry widget indexes (index, view index, etc.) Entry widgets have options that refer to character positions in the text being displayed. You can use these *Tkinter* functions to access these special points in text widgets:

AtEnd() refers to the last position in the text

AtInsert() refers to the point where the text cursor is

AtSelFirst() indicates the beginning point of the selected text

AtSelLast() denotes the last point of the selected text and finally

At(x[, y]) refers to the character at pixel location *x*, *y* (with *y* not used in the case of a text entry widget, which contains a single line of text).

Text widget indexes The index notation for Text widgets is very rich and is best described in the Tk man pages.

Menu indexes (menu.invoke(), menu.entryconfig(), etc.) Some options and methods for menus manipulate specific menu entries. Anytime a menu index is needed for an option or a parameter, you may pass in:

- an integer which refers to the numeric position of the entry in the widget, counted from the top, starting with 0;
- the string `'active'`, which refers to the menu position that is currently under the cursor;
- the string `"last"` which refers to the last menu item;
- An integer preceded by `@`, as in `@6`, where the integer is interpreted as a y pixel coordinate in the menu's coordinate system;
- the string `"none"`, which indicates no menu entry at all, most often used with menu.activate() to deactivate all entries, and finally,
- a text string that is pattern matched against the label of the menu entry, as scanned from the top of the menu to the bottom. Note that this index type is considered after all the others, which means that matches for menu items labelled `last`, `active`, or `none` may be interpreted as the above literals, instead.

Images

Images of different formats can be created through the corresponding subclass of `Tkinter.Image`:

- `BitmapImage` for images in XBM format.
- `PhotoImage` for images in PGM, PPM, GIF and PNG formats. The latter is supported starting with Tk 8.6.

Either type of image is created through either the `file` or the `data` option (other options are available as well).

The image object can then be used wherever an `image` option is supported by some widget (e.g. labels, buttons, menus). In these cases, Tk will not keep a reference to the image. When the last Python reference to the image object is deleted, the image data is deleted as well, and Tk will display an empty box wherever the image was used.

参见:

The Pillow package adds support for formats such as BMP, JPEG, TIFF, and WebP, among others.

24.1.7 File Handlers

Tk allows you to register and unregister a callback function which will be called from the Tk mainloop when I/O is possible on a file descriptor. Only one handler may be registered per file descriptor. Example code:

```
import Tkinter
widget = Tkinter.Tk()
mask = Tkinter.READABLE | Tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

This feature is not available on Windows.

Since you don't know how many bytes are available for reading, you may not want to use the *BufferedIOBase* or *TextIOBase* *read()* or *readline()* methods, since these will insist on reading a predefined number of bytes. For sockets, the *recv()* or *recvfrom()* methods will work fine; for other files, use raw reads or `os.read(file.fileno(), maxbytecount)`.

`Widget.tk.`**`createfilehandler`**(*file*, *mask*, *func*)
Registers the file handler callback function *func*. The *file* argument may either be an object with a *fileno()* method (such as a file or socket object), or an integer file descriptor. The *mask* argument is an ORed combination of any of the three constants below. The callback is called as follows:

```
callback(file, mask)
```

`Widget.tk.`**`deletefilehandler`**(*file*)
Unregisters a file handler.

`Tkinter.`**`READABLE`**
`Tkinter.`**`WRITABLE`**
`Tkinter.`**`EXCEPTION`**
Constants used in the *mask* arguments.

24.2 `ttk` —Tk themed widgets

The *ttk* module provides access to the Tk themed widget set, which has been introduced in Tk 8.5. If Python is not compiled against Tk 8.5 code may still use this module as long as Tile is installed. However, some features provided by the new Tk, like anti-aliased font rendering under X11, window transparency (on X11 you will need a composition window manager) will be missing.

The basic idea of *ttk* is to separate, to the extent possible, the code implementing a widget's behavior from the code implementing its appearance.

参见:

Tk Widget Styling Support The document which brought up theming support for Tk

24.2.1 Using Ttk

To start using Ttk, import its module:

```
import ttk
```

But code like this:

```
from Tkinter import *
```

may optionally want to use this:

```
from Tkinter import *
from ttk import *
```

And then several *ttk* widgets (`Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale` and `Scrollbar`) will automatically substitute for the Tk widgets.

This has the direct benefit of using the new widgets, giving better look & feel across platforms, but be aware that they are not totally compatible. The main difference is that widget options such as "fg", "bg" and others related to widget styling are no longer present in Ttk widgets. Use *ttk.Style* to achieve the same (or better) styling.

参见:

Converting existing applications to use the Tile widgets A text which talks in Tcl terms about differences typically found when converting applications to use the new widgets.

24.2.2 Ttk Widgets

Ttk comes with 17 widgets, 11 of which already exist in Tkinter: `Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale` and `Scrollbar`. The 6 new widget classes are: *Combobox*, *Notebook*, *Progressbar*, `Separator`, `Sizegrip` and *Treeview*. All of these classes are subclasses of *Widget*.

As said previously, you will notice changes in look-and-feel as well in the styling code. To demonstrate the latter, a very simple example is shown below.

Tk code:

```
l1 = Tkinter.Label(text="Test", fg="black", bg="white")
l2 = Tkinter.Label(text="Test", fg="black", bg="white")
```

Corresponding Ttk code:

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

For more information about *TtkStyling* read the *Style* class documentation.

24.2.3 Widget

ttk.Widget defines standard options and methods supported by Tk themed widgets and is not supposed to be directly instantiated.

Standard Options

All the *ttk* widgets accept the following options:

Option	Description
class	Specifies the window class. The class is used when querying the option database for the window's other options, to determine the default bindtags for the window, and to select the widget's default layout and style. This is a read-only option which may only be specified when the window is created.
cursor	Specifies the mouse cursor to be used for the widget. If set to the empty string (the default), the cursor is inherited from the parent widget.
takefocus	Determines whether the window accepts the focus during keyboard traversal. 0, 1 or an empty string is returned. If 0, the window should be skipped entirely during keyboard traversal. If 1, the window should receive the input focus as long as it is viewable. An empty string means that the traversal scripts make the decision about whether or not to focus on the window.
style	May be used to specify a custom widget style.

Scrollable Widget Options

The following options are supported by widgets that are controlled by a scrollbar.

option	description
xscrollcommand	Used to communicate with horizontal scrollbars. When the view in the widget's window changes, the widget will generate a Tcl command based on the scrollcommand. Usually this option consists of the `Scrollbar.set()` method of some scrollbar. This will cause the scrollbar to be updated whenever the view in the window changes.
yscrollcommand	Used to communicate with vertical scrollbars. For more information, see above.

Label Options

The following options are supported by labels, buttons and other button-like widgets.

option	description
text	Specifies a text string to be displayed inside the widget.
textvariable	Specifies a name whose value will be used in place of the text option resource.
underline	If set, specifies the index (0-based) of a character to underline in the text string. The underline character is used for mnemonic activation.
image	Specifies an image to display. This is a list of 1 or more elements. The first element is the default image name. The rest of the list is a sequence of statespec/value pairs as defined by *Style.map()*, specifying different images to use when the widget is in a particular state or a combination of states. All images in the list should have the same size.
compound	Specifies how to display the image relative to the text, in the case both text and image options are present. Valid values are: • text: display text only • image: display image only • top, bottom, left, right: display image above, below, left of, or right of the text, respectively. • none: the default. display the image if present, otherwise the text.
width	If greater than zero, specifies how much space, in character widths, to allocate for the text label; if less than zero, specifies a minimum width. If zero or unspecified, the natural width of the text label is used.

Compatibility Options

option	description
state	May be set to "normal" or "disabled" to control the "disabled" state bit. This is a write-only option: setting it changes the widget state, but the *Widget.state()* method does not affect this option.

Widget States

The widget state is a bitmap of independent state flags.

flag	description
active	The mouse cursor is over the widget and pressing a mouse button will cause some action to occur.
disabled	Widget is disabled under program control.
focus	Widget has keyboard focus.
pressed	Widget is being pressed.
selected	“On”, “true”, or “current” for things like Checkbuttons and radiobuttons.
background	Windows and Mac have a notion of an “active” or foreground window. The *background* state is set for widgets in a background window, and cleared for those in the foreground window.
readonly	Widget should not allow user modification.
alternate	A widget-specific alternate display format.
invalid	The widget’s value is invalid.

A state specification is a sequence of state names, optionally prefixed with an exclamation point indicating that the bit is off.

ttk.Widget

Besides the methods described below, the *ttk.Widget* class supports the `Tkinter.Widget.cget()` and `Tkinter.Widget.configure()` methods.

class ttk.**Widget**

identify(*x*, *y*)
Returns the name of the element at position *x* *y*, or the empty string if the point does not lie within any element.

x and *y* are pixel coordinates relative to the widget.

instate(*statespec*, *callback=None*, **args*, ***kw*)
Test the widget’s state. If a callback is not specified, returns `True` if the widget state matches *statespec* and `False` otherwise. If callback is specified then it is called with *args* if widget state matches *statespec*.

state(*[statespec=None]*)
Modify or read widget state. If *statespec* is specified, sets the widget state accordingly and returns a new *statespec* indicating which flags were changed. If *statespec* is not specified, returns the currently-enabled state flags.

statespec will usually be a list or a tuple.

24.2.4 Combobox

The *ttk.Combobox* widget combines a text field with a pop-down list of values. This widget is a subclass of `Entry`.

Besides the methods inherited from *Widget* (`Widget.cget()`, `Widget.configure()`, *Widget.identify()*, *Widget.instate()* and *Widget.state()*) and those inherited from `Entry` (`Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.selection()`, `Entry.xview()`), this class has some other methods, described at *ttk.Combobox*.

Options

This widget accepts the following options:

option	description
exportselection	Boolean value. If set, the widget selection is linked to the Window Manager selection (which can be returned by invoking `Misc.selection_get()`, for example).
justify	Specifies how the text is aligned within the widget. One of "left", "center", or "right".
height	Specifies the height of the pop-down listbox, in rows.
postcommand	A script (possibly registered with `Misc.register()`) that is called immediately before displaying the values. It may specify which values to display.
state	One of "normal", "readonly", or "disabled". In the "readonly" state, the value may not be edited directly, and the user can only select one of the values from the dropdown list. In the "normal" state, the text field is directly editable. In the "disabled" state, no interaction is possible.
textvariable	Specifies a name whose value is linked to the widget value. Whenever the value associated with that name changes, the widget value is updated, and vice versa. See `Tkinter.StringVar`.
values	Specifies the list of values to display in the drop-down listbox.
width	Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget's font.

Virtual events

The combobox widget generates a **«ComboboxSelected»** virtual event when the user selects an element from the list of values.

ttk.Combobox

class ttk.**Combobox**

current(*[newindex=None]*)
If *newindex* is specified, sets the combobox value to the element position *newindex*. Otherwise, returns the index of the current value or -1 if the current value is not in the values list.

get()
Returns the current value of the combobox.

set(*value*)
Sets the value of the combobox to *value*.

24.2.5 Notebook

The Ttk Notebook widget manages a collection of windows and displays a single one at a time. Each child window is associated with a tab, which the user may select to change the currently-displayed window.

Options

This widget accepts the following specific options:

option	description
height	If present and greater than zero, specifies the desired height of the pane area (not including internal padding or tabs). Otherwise, the maximum height of all panes is used.
padding	Specifies the amount of extra space to add around the outside of the notebook. The padding is a list of up to four length specifications: left top right bottom. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left.
width	If present and greater than zero, specifies the desired width of the pane area (not including internal padding). Otherwise, the maximum width of all panes is used.

Tab Options

There are also specific options for tabs:

option	description
state	Either “normal”, “disabled” or “hidden”. If “disabled”, then the tab is not selectable. If “hidden”, then the tab is not shown.
sticky	Specifies how the child window is positioned within the pane area. Value is a string containing zero or more of the characters “n”, “s”, “e” or “w”. Each letter refers to a side (north, south, east or west) that the child window will stick to, as per the `grid()` geometry manager.
padding	Specifies the amount of extra space to add between the notebook and this pane. Syntax is the same as for the option padding used by this widget.
text	Specifies a text to be displayed in the tab.
image	Specifies an image to display in the tab. See the option image described in *`Widget`*.
compound	Specifies how to display the image relative to the text, in the case both text and image options are present. See *Label Options* for legal values.
underline	Specifies the index (0-based) of a character to underline in the text string. The underlined character is used for mnemonic activation if *`Notebook.enable_traversal()`* is called.

Tab Identifiers

The *tab_id* present in several methods of *`ttk.Notebook`* may take any of the following forms:

- An integer between zero and the number of tabs.
- The name of a child window.
- A positional specification of the form “@x,y”, which identifies the tab.
- The literal string “current”, which identifies the currently-selected tab.
- The literal string “end”, which returns the number of tabs (only valid for *`Notebook.index()`*).

Virtual Events

This widget generates a **«NotebookTabChanged»** virtual event after a new tab is selected.

ttk.Notebook

class `ttk.`**Notebook**

add(*child*, ***kw*)
Adds a new tab to the notebook.

If window is currently managed by the notebook but hidden, it is restored to its previous position.

See *Tab Options* for the list of available options.

forget(*tab_id*)
Removes the tab specified by *tab_id*, unmaps and unmanages the associated window.

hide(*tab_id*)
Hides the tab specified by *tab_id*.

The tab will not be displayed, but the associated window remains managed by the notebook and its configuration remembered. Hidden tabs may be restored with the *`add()`* command.

identify (*x*, *y*)
Returns the name of the tab element at position *x*, *y*, or the empty string if none.

index (*tab_id*)
Returns the numeric index of the tab specified by *tab_id*, or the total number of tabs if *tab_id* is the string "end".

insert (*pos*, *child*, ***kw*)
Inserts a pane at the specified position.

pos is either the string "end", an integer index, or the name of a managed child. If *child* is already managed by the notebook, moves it to the specified position.

See *Tab Options* for the list of available options.

select ([*tab_id*])
Selects the specified *tab_id*.

The associated child window will be displayed, and the previously-selected window (if different) is unmapped. If *tab_id* is omitted, returns the widget name of the currently selected pane.

tab (*tab_id*, *option=None*, ***kw*)
Query or modify the options of the specific *tab_id*.

If *kw* is not given, returns a dictionary of the tab option values. If *option* is specified, returns the value of that *option*. Otherwise, sets the options to the corresponding values.

tabs ()
Returns a list of windows managed by the notebook.

enable_traversal ()
Enable keyboard traversal for a toplevel window containing this notebook.

This will extend the bindings for the toplevel window containing the notebook as follows:

- `Control-Tab`: selects the tab following the currently selected one.
- `Shift-Control-Tab`: selects the tab preceding the currently selected one.
- `Alt-K`: where *K* is the mnemonic (underlined) character of any tab, will select that tab.

Multiple notebooks in a single toplevel may be enabled for traversal, including nested notebooks. However, notebook traversal only works properly if all panes have the notebook they are in as master.

24.2.6 Progressbar

The *ttk.Progressbar* widget shows the status of a long-running operation. It can operate in two modes: determinate mode shows the amount completed relative to the total amount of work to be done, and indeterminate mode provides an animated display to let the user know that something is happening.

Options

This widget accepts the following specific options:

option	description
orient	One of "horizontal" or "vertical". Specifies the orientation of the progress bar.
length	Specifies the length of the long axis of the progress bar (width if horizontal, height if vertical).
mode	One of "determinate" or "indeterminate".
maximum	A number specifying the maximum value. Defaults to 100.
value	The current value of the progress bar. In "determinate" mode, this represents the amount of work completed. In "indeterminate" mode, it is interpreted as modulo *maximum*; that is, the progress bar completes one "cycle" when its value increases by *maximum*.
variable	A name which is linked to the option value. If specified, the value of the progress bar is automatically set to the value of this name whenever the latter is modified.
phase	Read-only option. The widget periodically increments the value of this option whenever its value is greater than 0 and, in determinate mode, less than maximum. This option may be used by the current theme to provide additional animation effects.

ttk.Progressbar

class ttk.**Progressbar**

start([*interval*])
Begin autoincrement mode: schedules a recurring timer event that calls *Progressbar.step()* every *interval* milliseconds. If omitted, *interval* defaults to 50 milliseconds.

step([*amount*])
Increments the progress bar's value by *amount*.

amount defaults to 1.0 if omitted.

stop()
Stop autoincrement mode: cancels any recurring timer event initiated by *Progressbar.start()* for this progress bar.

24.2.7 Separator

The ttk.Separator widget displays a horizontal or vertical separator bar.

It has no other methods besides the ones inherited from *ttk.Widget*.

Options

This widget accepts the following specific option:

option	description
orient	One of "horizontal" or "vertical". Specifies the orientation of the separator.

24.2.8 Sizegrip

The `ttk.Sizegrip` widget (also known as a grow box) allows the user to resize the containing toplevel window by pressing and dragging the grip.

This widget has neither specific options nor specific methods, besides the ones inherited from *`ttk.Widget`*.

Platform-specific notes

- On Mac OS X, toplevel windows automatically include a built-in size grip by default. Adding a `Sizegrip` is harmless, since the built-in grip will just mask the widget.

Bugs

- If the containing toplevel's position was specified relative to the right or bottom of the screen (e.g. ...), the `Sizegrip` widget will not resize the window.
- This widget supports only "southeast" resizing.

24.2.9 Treeview

The *`ttk.Treeview`* widget displays a hierarchical collection of items. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label.

The order in which data values are displayed may be controlled by setting the widget option `displaycolumns`. The tree widget can also display column headings. Columns may be accessed by number or symbolic names listed in the widget option columns. See *Column Identifiers*.

Each item is identified by a unique name. The widget will generate item IDs if they are not supplied by the caller. There is a distinguished root item, named `{}`. The root item itself is not displayed; its children appear at the top level of the hierarchy.

Each item also has a list of tags, which can be used to associate event bindings with individual items and control the appearance of the item.

The Treeview widget supports horizontal and vertical scrolling, according to the options described in *Scrollable Widget Options* and the methods *`Treeview.xview()`* and *`Treeview.yview()`*.

Options

This widget accepts the following specific options:

option	description
columns	A list of column identifiers, specifying the number of columns and their names.
displaycolumns	A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the string "#all".
height	Specifies the number of rows which should be visible. Note: the requested width is determined from the sum of the column widths.
padding	Specifies the internal padding for the widget. The padding is a list of up to four length specifications.
selectmode	Controls how the built-in class bindings manage the selection. One of "extended", "browse" or "none". If set to "extended" (the default), multiple items may be selected. If "browse", only a single item will be selected at a time. If "none", the selection will not be changed. Note that the application code and tag bindings can set the selection however they wish, regardless of the value of this option.
show	A list containing zero or more of the following values, specifying which elements of the tree to display. • tree: display tree labels in column #0. • headings: display the heading row. The default is "tree headings", i.e., show all elements. **Note**: Column #0 always refers to the tree column, even if show="tree" is not specified.

Item Options

The following item options may be specified for items in the insert and item widget commands.

option	description
text	The textual label to display for the item.
image	A Tk Image, displayed to the left of the label.
values	The list of values associated with the item. Each item should have the same number of values as the widget option columns. If there are fewer values than columns, the remaining values are assumed empty. If there are more values than columns, the extra values are ignored.
open	True/False value indicating whether the item's children should be displayed or hidden.
tags	A list of tags associated with this item.

Tag Options

The following options may be specified on tags:

option	description
foreground	Specifies the text foreground color.
background	Specifies the cell or item background color.
font	Specifies the font to use when drawing text.
image	Specifies the item image, in case the item's image option is empty.

Column Identifiers

Column identifiers take any of the following forms:

- A symbolic name from the list of columns option.
- An integer n, specifying the nth data column.
- A string of the form #n, where n is an integer, specifying the nth display column.

Notes:

- Item's option values may be displayed in a different order than the order in which they are stored.
- Column #0 always refers to the tree column, even if show="tree" is not specified.

A data column number is an index into an item's option values list; a display column number is the column number in the tree where the values are displayed. Tree labels are displayed in column #0. If option displaycolumns is not set, then data column n is displayed in column #n+1. Again, **column #0 always refers to the tree column**.

Virtual Events

The Treeview widget generates the following virtual events.

event	description
«TreeviewSelect»	Generated whenever the selection changes.
«TreeviewOpen»	Generated just before settings the focus item to open=True.
«TreeviewClose»	Generated just after setting the focus item to open=False.

The `Treeview.focus()` and `Treeview.selection()` methods can be used to determine the affected item or items.

ttk.Treeview

class ttk.**Treeview**

bbox(*item*, *column=None*)

Returns the bounding box (relative to the treeview widget's window) of the specified *item* in the form (x, y, width, height).

If *column* is specified, returns the bounding box of that cell. If the *item* is not visible (i.e., if it is a descendant of a closed item or is scrolled offscreen), returns an empty string.

get_children(*[item]*)
Returns the list of children belonging to *item*.

If *item* is not specified, returns root children.

set_children(*item*, **newchildren*)
Replaces *item*'s child with *newchildren*.

Children present in *item* that are not present in *newchildren* are detached from the tree. No items in *newchildren* may be an ancestor of *item*. Note that not specifying *newchildren* results in detaching *item*'s children.

column(*column*, *option=None*, ***kw*)
Query or modify the options for the specified *column*.

If *kw* is not given, returns a dict of the column option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- **id** Returns the column name. This is a read-only option.
- **anchor: One of the standard Tk anchor values.** Specifies how the text in this column should be aligned with respect to the cell.
- **minwidth: width** The minimum width of the column in pixels. The treeview widget will not make the column any smaller than specified by this option when the widget is resized or the user drags a column.
- **stretch: True/False** Specifies whether the column's width should be adjusted when the widget is resized.
- **width: width** The width of the column in pixels.

To configure the tree column, call this with column = "#0"

delete(**items*)
Delete all specified *items* and all their descendants.

The root item may not be deleted.

detach(**items*)
Unlinks all of the specified *items* from the tree.

The items and all of their descendants are still present, and may be reinserted at another point in the tree, but will not be displayed.

The root item may not be detached.

exists(*item*)
Returns `True` if the specified *item* is present in the tree.

focus(*[item=None]*)
If *item* is specified, sets the focus item to *item*. Otherwise, returns the current focus item, or '' if there is none.

heading(*column*, *option=None*, ***kw*)
Query or modify the heading options for the specified *column*.

If *kw* is not given, returns a dict of the heading option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- **text: text** The text to display in the column heading.
- **image: imageName** Specifies an image to display to the right of the column heading.

- **anchor: anchor** Specifies how the heading text should be aligned. One of the standard Tk anchor values.
- **command: callback** A callback to be invoked when the heading label is pressed.

To configure the tree column heading, call this with column = "#0".

identify (*component*, *x*, *y*)
Returns a description of the specified *component* under the point given by *x* and *y*, or the empty string if no such *component* is present at that position.

identify_row (*y*)
Returns the item ID of the item at position *y*.

identify_column (*x*)
Returns the data column identifier of the cell at position *x*.

The tree column has ID #0.

identify_region (*x*, *y*)
Returns one of:

region	meaning
heading	Tree heading area.
separator	Space between two columns headings.
tree	The tree area.
cell	A data cell.

Availability: Tk 8.6.

identify_element (*x*, *y*)
Returns the element at position *x*, *y*.

Availability: Tk 8.6.

index (*item*)
Returns the integer index of *item* within its parent's list of children.

insert (*parent*, *index*, *iid=None*, ***kw*)
Creates a new item and returns the item identifier of the newly created item.

parent is the item ID of the parent item, or the empty string to create a new top-level item. *index* is an integer, or the value "end", specifying where in the list of parent's children to insert the new item. If *index* is less than or equal to zero, the new node is inserted at the beginning; if *index* is greater than or equal to the current number of children, it is inserted at the end. If *iid* is specified, it is used as the item identifier; *iid* must not already exist in the tree. Otherwise, a new unique identifier is generated.

See *Item Options* for the list of available points.

item (*item*[, *option*[, ***kw*]])
Query or modify the options for the specified *item*.

If no options are given, a dict with options/values for the item is returned. If *option* is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values as given by *kw*.

move (*item*, *parent*, *index*)
Moves *item* to position *index* in *parent*'s list of children.

It is illegal to move an item under one of its descendants. If *index* is less than or equal to zero, *item* is moved to the beginning; if greater than or equal to the number of children, it is moved to the end. If *item* was detached it is reattached.

next (*item*)
Returns the identifier of *item*'s next sibling, or '' if *item* is the last child of its parent.

parent (*item*)
Returns the ID of the parent of *item*, or '' if *item* is at the top level of the hierarchy.

prev (*item*)
Returns the identifier of *item*'s previous sibling, or '' if *item* is the first child of its parent.

reattach (*item*, *parent*, *index*)
An alias for `Treeview.move()`.

see (*item*)
Ensure that *item* is visible.

Sets all of *item*'s ancestors open option to `True`, and scrolls the widget if necessary so that *item* is within the visible portion of the tree.

selection ([*selop=None*[, *items=None*]])
If *selop* is not specified, returns selected items. Otherwise, it will act according to the following selection methods.

selection_set (*items*)
items becomes the new selection.

selection_add (*items*)
Add *items* to the selection.

selection_remove (*items*)
Remove *items* from the selection.

selection_toggle (*items*)
Toggle the selection state of each item in *items*.

set (*item*, *column=None*, *value=None*)
With one argument, returns a dictionary of column/value pairs for the specified *item*. With two arguments, returns the current value of the specified *column*. With three arguments, sets the value of given *column* in given *item* to the specified *value*.

tag_bind (*tagname*, *sequence=None*, *callback=None*)
Bind a callback for the given event *sequence* to the tag *tagname*. When an event is delivered to an item, the callbacks for each of the item's tags option are called.

tag_configure (*tagname*, *option=None*, ***kw*)
Query or modify the options for the specified *tagname*.

If *kw* is not given, returns a dict of the option settings for *tagname*. If *option* is specified, returns the value for that *option* for the specified *tagname*. Otherwise, sets the options to the corresponding values for the given *tagname*.

tag_has (*tagname*[, *item*])
If *item* is specified, returns 1 or 0 depending on whether the specified *item* has the given *tagname*. Otherwise, returns a list of all items that have the specified tag.

Availability: Tk 8.6

xview (**args*)
Query or modify horizontal position of the treeview.

yview (**args*)
Query or modify vertical position of the treeview.

24.2.10 Ttk Styling

Each widget in *ttk* is assigned a style, which specifies the set of elements making up the widget and how they are arranged, along with dynamic and default settings for element options. By default the style name is the same as the widget's class name, but it may be overridden by the widget's style option. If the class name of a widget is unknown, use the method `Misc.winfo_class()` (somewidget.winfo_class()).

参见:

Tcl'2004 conference presentation This document explains how the theme engine works

class `ttk.Style`
This class is used to manipulate the style database.

configure(*style*, *query_opt=None*, ***kw*)
Query or set the default value of the specified option(s) in *style*.

Each key in *kw* is an option and each value is a string identifying the value for that option.

For example, to change every default button to be a flat button with some padding and a different background color do:

```
import ttk
import Tkinter

root = Tkinter.Tk()

ttk.Style().configure("TButton", padding=6, relief="flat",
   background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map(*style*, *query_opt=None*, ***kw*)
Query or sets dynamic values of the specified option(s) in *style*.

Each key in *kw* is an option and each value should be a list or a tuple (usually) containing statespecs grouped in tuples, lists, or something else of your preference. A statespec is a compound of one or more states and then a value.

An example:

```
import Tkinter
import ttk

root = Tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
    )

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()
```

Note that the order of the (states, value) sequences for an option matters. In the previous example, if you change the order to `[('active', 'blue'), ('pressed', 'red')]` in the foreground option, for example, you would get a blue foreground when the widget is in the active or pressed states.

lookup (*style*, *option*, *state=None*, *default=None*)
Returns the value specified for *option* in *style*.

If *state* is specified, it is expected to be a sequence of one or more states. If the *default* argument is set, it is used as a fallback value in case no specification for option is found.

To check what font a Button uses by default, do:

```
import ttk

print ttk.Style().lookup("TButton", "font")
```

layout (*style*, *layoutspec=None*)
Define the widget layout for given *style*. If *layoutspec* is omitted, return the layout specification for given style.

layoutspec, if specified, is expected to be a list or some other sequence type (excluding strings), where each item should be a tuple and the first item is the layout name and the second item should have the format described in *Layouts*.

To understand the format, see the following example (it is not intended to do anything useful):

```
import ttk
import Tkinter

root = Tkinter.Tk()

style = ttk.Style()
style.layout("TMenubutton", [
    ("Menubutton.background", None),
    ("Menubutton.button", {"children":
        [("Menubutton.focus", {"children":
            [("Menubutton.padding", {"children":
                [("Menubutton.label", {"side": "left", "expand": 1})]
            })]
        })]
    }),
])

mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

element_create (*elementname*, *etype*, **args*, ***kw*)
Create a new element in the current theme, of the given *etype* which is expected to be either "image", "from" or "vsapi". The latter is only available in Tk 8.6a for Windows XP and Vista and is not described here.

If "image" is used, *args* should contain the default image name followed by statespec/value pairs (this is the imagespec), and *kw* may have the following options:

- **border=padding** padding is a list of up to four integers, specifying the left, top, right, and bottom borders, respectively.
- **height=height** Specifies a minimum height for the element. If less than zero, the base image's height is used as a default.
- **padding=padding** Specifies the element's interior padding. Defaults to border's value if not specified.

- **sticky=spec** Specifies how the image is placed within the final parcel. spec contains zero or more characters "n", "s", "w", or "e".
- **width=width** Specifies a minimum width for the element. If less than zero, the base image's width is used as a default.

If "from" is used as the value of *etype*, `element_create()` will clone an existing element. *args* is expected to contain a themename, from which the element will be cloned, and optionally an element to clone from. If this element to clone from is not specified, an empty element will be used. *kw* is discarded.

element_names()
Returns the list of elements defined in the current theme.

element_options(*elementname*)
Returns the list of *elementname*'s options.

theme_create(*themename*, *parent=None*, *settings=None*)
Create a new theme.

It is an error if *themename* already exists. If *parent* is specified, the new theme will inherit styles, elements and layouts from the parent theme. If *settings* are present they are expected to have the same syntax used for `theme_settings()`.

theme_settings(*themename*, *settings*)
Temporarily sets the current theme to *themename*, apply specified *settings* and then restore the previous theme.

Each key in *settings* is a style and each value may contain the keys 'configure', 'map', 'layout' and 'element create' and they are expected to have the same format as specified by the methods `Style.configure()`, `Style.map()`, `Style.layout()` and `Style.element_create()` respectively.

As an example, let's change the Combobox for the default theme a bit:

```
import ttk
import Tkinter

root = Tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
   "TCombobox": {
       "configure": {"padding": 5},
       "map": {
           "background": [("active", "green2"),
                          ("!disabled", "green4")],
           "fieldbackground": [("!disabled", "green3")],
           "foreground": [("focus", "OliveDrab1"),
                          ("!disabled", "OliveDrab2")]
       }
   }
})

combo = ttk.Combobox().pack()

root.mainloop()
```

theme_names()
Returns a list of all known themes.

theme_use([*themename*])
If *themename* is not given, returns the theme in use. Otherwise, sets the current theme to *themename*, refreshes all widgets and emits a «ThemeChanged» event.

Layouts

A layout can be just `None`, if it takes no options, or a dict of options specifying how to arrange the element. The layout mechanism uses a simplified version of the pack geometry manager: given an initial cavity, each element is allocated a parcel. Valid options/values are:

- **side: whichside** Specifies which side of the cavity to place the element; one of top, right, bottom or left. If omitted, the element occupies the entire cavity.
- **sticky: nswe** Specifies where the element is placed inside its allocated parcel.
- **unit: 0 or 1** If set to 1, causes the element and all of its descendants to be treated as a single element for the purposes of *`Widget.identify()`* et al. It's used for things like scrollbar thumbs with grips.
- **children: [sublayout…]** Specifies a list of elements to place inside the element. Each element is a tuple (or other sequence type) where the first item is the layout name, and the other is a *Layout*.

24.3 `Tix` —Extension widgets for Tk

The *`Tix`* (Tk Interface Extension) module provides an additional rich set of widgets. Although the standard Tk library has many useful widgets, they are far from complete. The *`Tix`* library provides most of the commonly needed widgets that are missing from standard Tk: *`HList`*, *`ComboBox`*, *`Control`* (a.k.a. SpinBox) and an assortment of scrollable widgets. *`Tix`* also includes many more widgets that are generally useful in a wide range of applications: *`NoteBook`*, *`FileEntry`*, *`PanedWindow`*, etc; there are more than 40 of them.

With all these new widgets, you can introduce new interaction techniques into applications, creating more useful and more intuitive user interfaces. You can design your application by choosing the most appropriate widgets to match the special needs of your application and users.

注解: *`Tix`* has been renamed to `tkinter.tix` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

参见:

Tix Homepage The home page for *`Tix`*. This includes links to additional documentation and downloads.

Tix Man Pages On-line version of the man pages and reference material.

Tix Programming Guide On-line version of the programmer's reference material.

Tix Development Applications Tix applications for development of Tix and Tkinter programs. Tide applications work under Tk or Tkinter, and include **`TixInspect`**, an inspector to remotely modify and debug Tix/Tk/Tkinter applications.

24.3.1 Using Tix

class `Tix.`**`Tix`**(*screenName*[, *baseName*[, *className*]])

Toplevel widget of Tix which represents mostly the main window of an application. It has an associated Tcl interpreter.

Classes in the *`Tix`* module subclasses the classes in the *`Tkinter`* module. The former imports the latter, so to use *`Tix`* with Tkinter, all you need to do is to import one module. In general, you can just import *`Tix`*, and replace the toplevel call to *`Tkinter.Tk`* with `Tix.Tk`:

```
import Tix
from Tkconstants import *
root = Tix.Tk()
```

To use *Tix*, you must have the *Tix* widgets installed, usually alongside your installation of the Tk widgets. To test your installation, try the following:

```
import Tix
root = Tix.Tk()
root.tk.eval('package require Tix')
```

If this fails, you have a Tk installation problem which must be resolved before proceeding. Use the environment variable `TIX_LIBRARY` to point to the installed *Tix* library directory, and make sure you have the dynamic object library (`tix8183.dll` or `libtix8183.so`) in the same directory that contains your Tk dynamic object library (`tk8183.dll` or `libtk8183.so`). The directory with the dynamic object library should also have a file called `pkgIndex.tcl` (case sensitive), which contains the line:

```
package ifneeded Tix 8.1 [list load "[file join $dir tix8183.dll]" Tix]
```

24.3.2 Tix Widgets

Tix introduces over 40 widget classes to the *Tkinter* repertoire. There is a demo of all the *Tix* widgets in the `Demo/tix` directory of the standard distribution.

Basic Widgets

class `Tix.`**Balloon**
A Balloon that pops up over a widget to provide help. When the user moves the cursor inside a widget to which a Balloon widget has been bound, a small pop-up window with a descriptive message will be shown on the screen.

class `Tix.`**ButtonBox**
The ButtonBox widget creates a box of buttons, such as is commonly used for `Ok Cancel`.

class `Tix.`**ComboBox**
The ComboBox widget is similar to the combo box control in MS Windows. The user can select a choice by either typing in the entry subwidget or selecting from the listbox subwidget.

class `Tix.`**Control**
The Control widget is also known as the `SpinBox` widget. The user can adjust the value by pressing the two arrow buttons or by entering the value directly into the entry. The new value will be checked against the user-defined upper and lower limits.

class `Tix.`**LabelEntry**
The LabelEntry widget packages an entry widget and a label into one mega widget. It can be used to simplify the creation of "entry-form" type of interface.

class `Tix.`**LabelFrame**
The LabelFrame widget packages a frame widget and a label into one mega widget. To create widgets inside a LabelFrame widget, one creates the new widgets relative to the `frame` subwidget and manage them inside the `frame` subwidget.

class `Tix.`**Meter**
The Meter widget can be used to show the progress of a background job which may take a long time to execute.

class `Tix.`**OptionMenu**
The OptionMenu creates a menu button of options.

class Tix.**PopupMenu**
The PopupMenu widget can be used as a replacement of the `tk_popup` command. The advantage of the *Tix PopupMenu* widget is it requires less application code to manipulate.

class Tix.**Select**
The Select widget is a container of button subwidgets. It can be used to provide radio-box or check-box style of selection options for the user.

class Tix.**StdButtonBox**
The StdButtonBox widget is a group of standard buttons for Motif-like dialog boxes.

File Selectors

class Tix.**DirList**
The DirList widget displays a list view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class Tix.**DirTree**
The DirTree widget displays a tree view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class Tix.**DirSelectDialog**
The DirSelectDialog widget presents the directories in the file system in a dialog window. The user can use this dialog window to navigate through the file system to select the desired directory.

class Tix.**DirSelectBox**
The *DirSelectBox* is similar to the standard Motif(TM) directory-selection box. It is generally used for the user to choose a directory. DirSelectBox stores the directories mostly recently selected into a ComboBox widget so that they can be quickly selected again.

class Tix.**ExFileSelectBox**
The ExFileSelectBox widget is usually embedded in a tixExFileSelectDialog widget. It provides a convenient method for the user to select files. The style of the *ExFileSelectBox* widget is very similar to the standard file dialog on MS Windows 3.1.

class Tix.**FileSelectBox**
The FileSelectBox is similar to the standard Motif(TM) file-selection box. It is generally used for the user to choose a file. FileSelectBox stores the files mostly recently selected into a *ComboBox* widget so that they can be quickly selected again.

class Tix.**FileEntry**
The FileEntry widget can be used to input a filename. The user can type in the filename manually. Alternatively, the user can press the button widget that sits next to the entry, which will bring up a file selection dialog.

Hierarchical ListBox

class Tix.**HList**
The HList widget can be used to display any data that have a hierarchical structure, for example, file system directory trees. The list entries are indented and connected by branch lines according to their places in the hierarchy.

class Tix.**CheckList**
The CheckList widget displays a list of items to be selected by the user. CheckList acts similarly to the Tk check-button or radiobutton widgets, except it is capable of handling many more items than checkbuttons or radiobuttons.

class Tix.**Tree**
The Tree widget can be used to display hierarchical data in a tree form. The user can adjust the view of the tree by opening or closing parts of the tree.

Tabular ListBox

class Tix.**TList**
The TList widget can be used to display data in a tabular format. The list entries of a *TList* widget are similar to the entries in the Tk listbox widget. The main differences are (1) the *TList* widget can display the list entries in a two dimensional format and (2) you can use graphical images as well as multiple colors and fonts for the list entries.

Manager Widgets

class Tix.**PanedWindow**
The PanedWindow widget allows the user to interactively manipulate the sizes of several panes. The panes can be arranged either vertically or horizontally. The user changes the sizes of the panes by dragging the resize handle between two panes.

class Tix.**ListNoteBook**
The ListNoteBook widget is very similar to the `TixNoteBook` widget: it can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages (windows). At one time only one of these pages can be shown. The user can navigate through these pages by choosing the name of the desired page in the `hlist` subwidget.

class Tix.**NoteBook**
The NoteBook widget can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages. At one time only one of these pages can be shown. The user can navigate through these pages by choosing the visual "tabs" at the top of the NoteBook widget.

Image Types

The *Tix* module adds:

- pixmap capabilities to all *Tix* and *Tkinter* widgets to create color images from XPM files.
- Compound image types can be used to create images that consists of multiple horizontal lines; each line is composed of a series of items (texts, bitmaps, images or spaces) arranged from left to right. For example, a compound image can be used to display a bitmap and a text string simultaneously in a Tk `Button` widget.

Miscellaneous Widgets

class Tix.**InputOnly**
The InputOnly widgets are to accept inputs from the user, which can be done with the `bind` command (Unix only).

Form Geometry Manager

In addition, *Tix* augments *Tkinter* by providing:

class Tix.**Form**
The Form geometry manager based on attachment rules for all Tk widgets.

24.3.3 Tix Commands

class Tix.**tixCommand**

The tix commands provide access to miscellaneous elements of *Tix*' s internal state and the *Tix* application context. Most of the information manipulated by these methods pertains to the application as a whole, or to a screen or display, rather than to a particular window.

To view the current settings, the common usage is:

```
import Tix
root = Tix.Tk()
print root.tix_configure()
```

tixCommand.**tix_configure**(*cnf=None* ***kw*)

Query or modify the configuration options of the Tix application context. If no option is specified, returns a dictionary all of the available options. If option is specified with no value, then the method returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the method modifies the given option(s) to have the given value(s); in this case the method returns an empty string. Option may be any of the configuration options.

tixCommand.**tix_cget**(*option*)

Returns the current value of the configuration option given by *option*. Option may be any of the configuration options.

tixCommand.**tix_getbitmap**(*name*)

Locates a bitmap file of the name `name.xpm` or `name` in one of the bitmap directories (see the *tix_addbitmapdir()* method). By using *tix_getbitmap()*, you can avoid hard coding the pathnames of the bitmap files in your application. When successful, it returns the complete pathname of the bitmap file, prefixed with the character `@`. The returned value can be used to configure the `bitmap` option of the Tk and Tix widgets.

tixCommand.**tix_addbitmapdir**(*directory*)

Tix maintains a list of directories under which the *tix_getimage()* and *tix_getbitmap()* methods will search for image files. The standard bitmap directory is `$TIX_LIBRARY/bitmaps`. The *tix_addbitmapdir()* method adds *directory* into this list. By using this method, the image files of an applications can also be located using the *tix_getimage()* or *tix_getbitmap()* method.

tixCommand.**tix_filedialog**([*dlgclass*])

Returns the file selection dialog that may be shared among different calls from this application. This method will create a file selection dialog widget when it is called the first time. This dialog will be returned by all subsequent calls to *tix_filedialog()*. An optional dlgclass parameter can be passed as a string to specified what type of file selection dialog widget is desired. Possible options are `tix`, `FileSelectDialog` or `tixExFileSelectDialog`.

tixCommand.**tix_getimage**(*self*, *name*)

Locates an image file of the name `name.xpm`, `name.xbm` or `name.ppm` in one of the bitmap directories (see the *tix_addbitmapdir()* method above). If more than one file with the same name (but different extensions) exist, then the image type is chosen according to the depth of the X display: xbm images are chosen on monochrome displays and color images are chosen on color displays. By using *tix_getimage()*, you can avoid hard coding the pathnames of the image files in your application. When successful, this method returns the name of the newly created image, which can be used to configure the `image` option of the Tk and Tix widgets.

tixCommand.**tix_option_get**(*name*)

Gets the options maintained by the Tix scheme mechanism.

tixCommand.**tix_resetoptions**(*newScheme*, *newFontSet*[, *newScmPrio*])

Resets the scheme and fontset of the Tix application to *newScheme* and *newFontSet*, respectively. This affects only

those widgets created after this call. Therefore, it is best to call the resetoptions method before the creation of any widgets in a Tix application.

The optional parameter *newScmPrio* can be given to reset the priority level of the Tk options set by the Tix schemes.

Because of the way Tk handles the X option database, after Tix has been has imported and inited, it is not possible to reset the color schemes and font sets using the `tix_config()` method. Instead, the *`tix_resetoptions()`* method must be used.

24.4 `ScrolledText` —Scrolled Text Widget

The *`ScrolledText`* module provides a class of the same name which implements a basic text widget which has a vertical scroll bar configured to do the "right thing." Using the `ScrolledText` class is a lot easier than setting up a text widget and scroll bar directly. The constructor is the same as that of the `Tkinter.Text` class.

注解: *`ScrolledText`* has been renamed to `tkinter.scrolledtext` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

The text widget and scrollbar are packed together in a `Frame`, and the methods of the `Grid` and `Pack` geometry managers are acquired from the `Frame` object. This allows the `ScrolledText` widget to be used directly to achieve most normal geometry management behavior.

Should more specific control be necessary, the following attributes are available:

`ScrolledText.`**`frame`**
The frame which surrounds the text and scroll bar widgets.

`ScrolledText.`**`vbar`**
The scroll bar widget.

24.5 `turtle` —Turtle graphics for Tk

24.5.1 概述

Turtle graphics is a popular way for introducing programming to kids. It was part of the original Logo programming language developed by Wally Feurzig and Seymour Papert in 1966.

请想象绘图区有一只机器海龟，起始位置在 x-y 平面的 (0, 0) 点。先执行 `import turtle`，再执行 `turtle.forward(15)`，它将 (在屏幕上) 朝所面对的 x 轴正方向前进 15 像素，随着它的移动画出一条线段。再执行 `turtle.right(25)`，它将原地右转 25 度。

通过组合使用此类命令，可以轻松地绘制出精美的形状和图案。

`turtle` 模块是基于 Python 标准发行版 2.5 以来的同名模块重新编写并进行了功能扩展。

新模块尽量保持了原模块的特点，并且 (几乎)100% 与其兼容。这就意味着初学编程者能够以交互方式使用模块的所有命令、类和方法——运行 IDLE 时注意加 `-n` 参数。

The turtle module provides turtle graphics primitives, in both object-oriented and procedure-oriented ways. Because it uses *`Tkinter`* for the underlying graphics, it needs a version of Python installed with Tk support.

面向对象的接口主要使用 "2+2" 个类：

1. The *TurtleScreen* class defines graphics windows as a playground for the drawing turtles. Its constructor needs a `Tkinter.Canvas` or a *ScrolledCanvas* as argument. It should be used when *turtle* is used as part of some application.

   Screen() 函数返回一个*TurtleScreen* 子类的单例对象。此函数应在*turtle* 作为独立绘图工具时使用。作为一个单例对象，其所属的类是不可被继承的。

   TurtleScreen/Screen 的所有方法还存在对应的函数，即作为面向过程的接口组成部分。

2. *RawTurtle* (别名: *RawPen*) 类定义海龟对象在*TurtleScreen* 上绘图。它的构造器需要一个 Canvas, ScrolledCanvas 或 TurtleScreen 作为参数，以指定 RawTurtle 对象在哪里绘图。

   Derived from RawTurtle is the subclass *Turtle* (alias: `Pen`), which draws on "the" *Screen* - instance which is automatically created, if not already present.

   RawTurtle/Turtle 的所有方法也存在对应的函数，即作为面向过程的接口组成部分。

过程式接口提供与*Screen* 和*Turtle* 类的方法相对应的函数。函数名与对应的方法名相同。当 Screen 类的方法对应函数被调用时会自动创建一个 Screen 对象。当 Turtle 类的方法对应函数被调用时会自动创建一个(匿名的) Turtle 对象。

To use multiple turtles an a screen one has to use the object-oriented interface.

注解： 以下文档给出了函数的参数列表。对于方法来说当然还有额外的第一个参数 *self*，这里省略了。

24.5.2 Overview over available Turtle and Screen methods

Turtle 方法

海龟动作

移动和绘制

forward() | *fd()* 前进
backward() | *bk()* | *back()* 后退
right() | *rt()* 右转
left() | *lt()* 左转
goto() | *setpos()* | *setposition()* 前往/定位
setx() 设置 x 坐标
sety() 设置 y 坐标
setheading() | *seth()* 设置朝向
home() 返回原点
circle() 画圆
dot() 画点
stamp() 印章
clearstamp() 清除印章
clearstamps() 清除多个印章
undo() 撤消
speed() 速度

获取海龟的状态

position() | *pos()* 位置
towards() 目标方向

xcor() x 坐标
ycor() y 坐标
heading() 朝向
distance() 距离

设置与度量单位

degrees() 角度
radians() 弧度

画笔控制

绘图状态

pendown() | *pd()* | *down()* 画笔落下
penup() | *pu()* | *up()* 画笔抬起
pensize() | *width()* 画笔粗细
pen() 画笔
isdown() 画笔是否落下

颜色控制

color() 颜色
pencolor() 画笔颜色
fillcolor() 填充颜色

填充

fill()
begin_fill() 开始填充
end_fill() 结束填充

更多绘图控制

reset() 重置
clear() 清空
write() 书写

海龟状态

可见性

showturtle() | *st()* 显示海龟
hideturtle() | *ht()* 隐藏海龟
isvisible() 是否可见

外观

shape() 形状
resizemode() 大小调整模式
shapesize() | *turtlesize()* 形状大小
settiltangle() 设置倾角
tiltangle() 倾角
tilt() 倾斜

使用事件

onclick() 当鼠标点击
onrelease() 当鼠标释放

ondrag() 当鼠标拖动
mainloop() | *done()* 主循环

特殊海龟方法

begin_poly() 开始记录多边形
end_poly() 结束记录多边形
get_poly() 获取多边形
clone() 克隆
getturtle() | *getpen()* 获取海龟画笔
getscreen() 获取屏幕
setundobuffer() 设置撤消缓冲区
undobufferentries() 撤消缓冲区条目数
tracer() 追踪
window_width() 窗口宽度
window_height() 窗口高度

TurtleScreen/Screen 方法

窗口控制

bgcolor() 背景颜色
bgpic() 背景图片
clear() | *clearscreen()* 清屏
reset() | *resetscreen()* 重置
screensize() 屏幕大小
setworldcoordinates() 设置世界坐标系

动画控制

delay() 延迟
tracer() 追踪
update() 更新

使用屏幕事件

listen() 监听
onkey()
onclick() | *onscreenclick()* 当点击屏幕
ontimer() 当达到定时

设置与特殊方法

mode()
colormode() 颜色模式
getcanvas() 获取画布
getshapes() 获取形状
register_shape() | *addshape()* 添加形状
turtles() 所有海龟
window_height() 窗口高度
window_width() 窗口宽度

Screen 专有方法

bye() 退出
exitonclick() 当点击时退出
setup() 设置
title() 标题

24.5.3 RawTurtle/Turtle 方法和对应函数

本节中的大部分示例都使用 Turtle 类的一个实例，命名为 `turtle`。

海龟动作

`turtle.`**`forward`**(*distance*)
`turtle.`**`fd`**(*distance*)

参数 distance —一个数值 (整型或浮点型)

海龟前进 *distance* 指定的距离，方向为海龟的朝向。

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

`turtle.`**`back`**(*distance*)
`turtle.`**`bk`**(*distance*)
`turtle.`**`backward`**(*distance*)

参数 distance —一个数值

海龟后退 *distance* 指定的距离，方向与海龟的朝向相反。不改变海龟的朝向。

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

`turtle.`**`right`**(*angle*)
`turtle.`**`rt`**(*angle*)

参数 angle —一个数值 (整型或浮点型)

海龟右转 *angle* 个单位。(单位默认为角度，但可通过*degrees()* 和*radians()* 函数改变设置。) 角度的正负由海龟模式确定，参见*mode()*。

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

`turtle.`**`left`**(*angle*)
`turtle.`**`lt`**(*angle*)

参数 angle –一个数值 (整型或浮点型)

海龟左转 *angle* 个单位。(单位默认为角度，但可通过*degrees()* 和*radians()* 函数改变设置。) 角度的正负由海龟模式确定，参见*mode()*。

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

turtle.**goto**(*x*, *y=None*)
turtle.**setpos**(*x*, *y=None*)
turtle.**setposition**(*x*, *y=None*)

参数

- **x** –一个数值或数值对/向量
- **y** –一个数值或 None

如果 *y* 为 None，*x* 应为一个表示坐标的数值对或*Vec2D* 类对象 (例如*pos()* 返回的对象).

海龟移动到一个绝对坐标。如果画笔已落下将会画线。不改变海龟的朝向。

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

turtle.**setx**(*x*)

参数 x –一个数值 (整型或浮点型)

设置海龟的横坐标为 *x*，纵坐标保持不变。

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

turtle.**sety**(*y*)

参数 y –一个数值 (整型或浮点型)

设置海龟的纵坐标为 *y*，横坐标保持不变。

```
>>> turtle.position()
(0.00,40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

turtle.**setheading**(*to_angle*)
turtle.**seth**(*to_angle*)

参数 to_angle —一个数值 (整型或浮点型)

设置海龟的朝向为 *to_angle*。以下是以角度表示的几个常用方向：

标准模式	logo 模式
0 - 东	0 - 北
90 - 北	90 - 东
180 - 西	180 - 南
270 - 南	270 - 西

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

turtle.**home**()

海龟移至初始坐标 (0,0)，并设置朝向为初始方向 (由海龟模式确定，参见*mode()*)。

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**circle**(*radius*, *extent=None*, *steps=None*)

参数

- **radius** —一个数值
- **extent** —一个数值 (或 None)
- **steps** —一个整型数 (或 None)

绘制一个 *radius* 指定半径的圆。圆心在海龟左边 *radius* 个单位；*extent* 为一个夹角，用来决定绘制圆的一部分。如未指定 *extent** 则绘制整个圆。如果 **extent* 不是完整圆周，则以当前画笔位置为一个端点绘制圆弧。如果 *radius* 为正值则朝逆时针方向绘制圆弧，否则朝顺时针方向。最终海龟的朝向会依据 *extent* 的值而改变。

圆实际是以其内切正多边形来近似表示的，其边的数量由 *steps* 指定。如果未指定边数则会自动确定。此方法也可用来绘制正多边形。

```
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
```

(下页继续)

(续上页)

```
>>> turtle.circle(120, 180)  # draw a semicircle
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
```

turtle.**dot**(*size=None*, **color*)

参数

- **size** —一个整型数 >= 1 (如果指定)
- **color** —一个颜色字符串或颜色数值元组

绘制一个直径为 *size*，颜色为 *color* 的圆点。如果 *size* 未指定，则直径取 pensize+4 和 2*pensize 中的较大值。

```
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
```

turtle.**stamp**()

在海龟当前位置印制一个海龟形状。返回该印章的 stamp_id，印章可以通过调用 `clearstamp(stamp_id)` 来删除。

```
>>> turtle.color("blue")
>>> turtle.stamp()
11
>>> turtle.fd(50)
```

turtle.**clearstamp**(*stampid*)

参数 stampid —一个整型数，必须是之前*stamp()* 调用的返回值

删除 *stampid* 指定的印章。

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

turtle.**clearstamps**(*n=None*)

参数 n —一个整型数 (或 None)

删除全部或前/后 *n* 个海龟印章。如果 *n* 为 None 则删除全部印章，如果 $n > 0$ 则删除前 *n* 个印章，否则如果 $n < 0$ 则删除后 *n* 个印章。

```
>>> for i in range(8):
...     turtle.stamp(); turtle.fd(30)
13
14
15
16
17
18
19
20
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

turtle.**undo**()

撤消 (或连续撤消) 最近的一个 (或多个) 海龟动作。可撤消的次数由撤消缓冲区的大小决定。

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

turtle.**speed**(*speed=None*)

参数 speed – 一个 0..10 范围内的整型数或速度字符串 (见下)

设置海龟移动的速度为 0..10 表示的整型数值。如未指定参数则返回当前速度。

如果输入数值大于 10 或小于 0.5 则速度设为 0。速度字符串与速度值的对应关系如下:

- “fastest” : 0 最快
- “fast” : 10 快
- “normal” : 6 正常
- “slow” : 3 慢
- “slowest” : 1 最慢

速度值从 1 到 10，画线和海龟转向的动画效果逐级加快。

注意: *speed* = 0 表示 没有动画效果。forward/back 将使海龟向前/向后跳跃，同样的 left/right 将使海龟立即改变朝向。

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

获取海龟的状态

turtle.**position**()
turtle.**pos**()
返回海龟当前的坐标 (x,y) (为*Vec2D* 矢量类对象)。

```
>>> turtle.pos()
(440.00,-0.00)
```

turtle.**towards**(*x*, *y=None*)

参数

- **x** –一个数值或数值对/矢量，或一个海龟实例
- **y** –一个数值——如果 *x* 是一个数值，否则为 None

从海龟位置到由 (x,y)，矢量或另一海龟对应位置的连线的夹角。此数值依赖于海龟初始朝向 - 由“standard”/”world”或“logo”模式设置所决定)。

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

turtle.**xcor**()
返回海龟的 x 坐标。

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28,76.60)
>>> print turtle.xcor()
64.2787609687
```

turtle.**ycor**()
返回海龟的 y 坐标。

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print turtle.pos()
(50.00,86.60)
>>> print turtle.ycor()
86.6025403784
```

turtle.**heading**()
返回海龟当前的朝向 (数值依赖于海龟模式参见*mode()*)。

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

turtle.**distance**(*x*, *y=None*)

参数

- **x** –一个数值或数值对/矢量，或一个海龟实例

- **y** —一个数值——如果 *x* 是一个数值，否则为 None

返回从海龟位置到由 (x,y)，适量或另一海龟对应位置的单位距离。

```
>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

度量单位设置

turtle.**degrees**(*fullcircle=360.0*)

参数 fullcircle —一个数值

设置角度的度量单位，即设置一个圆周为多少“度”。默认值为 360 度。

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

Change angle measurement unit to grad (also known as gon,
grade, or gradian and equals 1/100-th of the right angle.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

turtle.**radians**()

设置角度的度量单位为弧度。其值等于 degrees(2*math.pi)。

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

画笔控制

绘图状态

turtle.**pendown**()
turtle.**pd**()
turtle.**down**()
画笔落下–移动时将画线。

turtle.**penup**()
turtle.**pu**()
turtle.**up**()
画笔抬起–移动时不画线。

turtle.**pensize**(*width=None*)
turtle.**width**(*width=None*)

参数 width –一个正数值

设置线条的粗细为 *width* 或返回该值。如果 resizemode 设为“auto”并且 turtleshape 为多边形，该多边形也以同样组细的线条绘制。如未指定参数，则返回当前的 pensize。

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)   # from here on lines of width 10 are drawn
```

turtle.**pen**(*pen=None*, ***pendict*)

参数

- **pen** –一个包含部分或全部下列键的字典
- **pendict** –一个或多个以下列键为关键字的关键字参数

返回或设置画笔的属性，以一个包含以下键值对的“画笔字典”表示:

- “shown”：True/False
- “pendown”：True/False
- “pencolor”：颜色字符串或颜色元组
- “fillcolor”：颜色字符串或颜色元组
- “pensize”：正数值
- “speed”：0..10 范围内的数值
- “resizemode”：“auto”或“user”或“noresize”
- “stretchfactor”：(正数值, 正数值)
- “outline”：正数值
- “tilt”：数值

此字典可作为后续调用*pen()*时的参数，以恢复之前的画笔状态。另外还可将这些属性作为关键词参数提交。使用此方式可以用一条语句设置画笔的多个属性。

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
```

(下页继续)

(续上页)

```
 ('shown', True), ('speed', 9), ('stretchfactor', (1, 1)), ('tilt', 0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow'),
 ('pendown', False), ('pensize', 10), ('resizemode', 'noresize'),
 ('shown', True), ('speed', 9), ('stretchfactor', (1, 1)), ('tilt', 0)]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shown', True), ('speed', 9), ('stretchfactor', (1, 1)), ('tilt', 0)]
```

turtle.**isdown**()

如果画笔落下返回 True，如果画笔抬起返回 False。

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

颜色控制

turtle.**pencolor**(**args*)

返回或设置画笔颜色。

允许以下四种输入格式:

pencolor() 返回以颜色描述字符串或元组 (见示例) 表示的当前画笔颜色。可用作其他 color/pencolor/fillcolor 调用的输入。

pencolor(colorstring) 设置画笔颜色为 *colorstring* 指定的 Tk 颜色描述字符串，例如 "red"、"yellow" 或 "#33cc8c"。

pencolor((r, g, b)) 设置画笔颜色为以 *r*, *g*, *b* 元组表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode，colormode 的值为 1.0 或 255 (参见*colormode()*)。

pencolor(r, g, b)

设置画笔颜色为以 *r*, *g*, *b* 表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode。

如果 turtleshape 为多边形，该多边形轮廓也以新设置的画笔颜色绘制。

```
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
```

(下页继续)

(续上页)

```
>>> colormode(255)
>>> turtle.pencolor()
(51, 204, 140)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50, 193, 143)
```

turtle.**fillcolor**(**args*)

返回或设置填充颜色。

允许以下四种输入格式:

fillcolor() 返回以颜色描述字符串或元组 (见示例) 表示的当前填充颜色。可用作其他 color/pencolor/fillcolor 调用的输入。

fillcolor(colorstring) 设置填充颜色为 *colorstring* 指定的 Tk 颜色描述字符串，例如 `"red"`、`"yellow"` 或 `"#33cc8c"`。

fillcolor((r, g, b)) 设置填充颜色为以 *r*, *g*, *b* 元组表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode，colormode 的值为 1.0 或 255 (参见*colormode()*)。

fillcolor(r, g, b)

设置填充颜色为 *r*, *g*, *b* 表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode。

如果 turtleshape 为多边形，该多边形内部也以新设置的填充颜色填充。

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> col = turtle.pencolor()
>>> col
(50, 193, 143)
>>> turtle.fillcolor(col)
>>> turtle.fillcolor()
(50, 193, 143)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255, 255, 255)
```

turtle.**color**(**args*)

返回或设置画笔颜色和填充颜色。

允许多种输入格式。使用如下 0 至 3 个参数:

color() 返回以一对颜色描述字符串或元组表示的当前画笔颜色和填充颜色，两者可分别由*pencolor()* 和*fillcolor()* 返回。

color(colorstring),color((r,g,b)),color(r,g,b) 输入格式与*pencolor()* 相同，同时设置填充颜色和画笔颜色为指定的值。

color(colorstring1, colorstring2),color((r1,g1,b1), (r2,g2,b2))

相当于 `pencolor(colorstring1)` 加 `fillcolor(colorstring2)`，使用其他输入格式的方法也与之类似。

如果 turtleshape 为多边形，该多边形轮廓与填充也使用新设置的颜色。

```
>>> turtle.color("red", "green")
>>> turtle.color()
```

(下页继续)

(续上页)

```
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40, 80, 120), (160, 200, 240))
```

另参见: Screen 方法*colormode()*。

填充

turtle.**fill**(*flag*)

参数 **flag** –True/False (or 1/0 respectively)

Call `fill(True)` before drawing the shape you want to fill, and `fill(False)` when done. When used without argument: return fillstate (`True` if filling, `False` else).

```
>>> turtle.fill(True)
>>> for _ in range(3):
...     turtle.forward(100)
...     turtle.left(120)
...
>>> turtle.fill(False)
```

turtle.**begin_fill**()
Call just before drawing a shape to be filled. Equivalent to `fill(True)`.

turtle.**end_fill**()
Fill the shape drawn after the last call to *begin_fill()*. Equivalent to `fill(False)`.

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

更多绘图控制

turtle.**reset**()
从屏幕中删除海龟的绘图，海龟回到原点并设置所有变量为默认值。

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
(0.00,-22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**clear**()
从屏幕中删除指定海龟的绘图。不移动海龟。海龟的状态和位置以及其他海龟的绘图不受影响。

turtle.**write**(*arg*, *move=False*, *align="left"*, *font=("Arial", 8, "normal")*)

参数

- **arg** –要书写到 TurtleScreen 的对象
- **move** –True/False
- **align** –字符串“left”，“center”或“right”
- **font** –一个三元组 (fontname, fontsize, fonttype)

书写文本 - *arg* 指定的字符串 - 到当前海龟位置，*align* 指定对齐方式（“left”，“center”或 right”），font 指定字体。如果 *move* 为 True，画笔会移动到文本的右下角。默认 *move* 为 `False`。

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

海龟状态

可见性

`turtle.`**`hideturtle`**`()`
`turtle.`**`ht`**`()`
使海龟不可见。当你绘制复杂图形时这是个好主意，因为隐藏海龟可显著加快绘制速度。

```
>>> turtle.hideturtle()
```

`turtle.`**`showturtle`**`()`
`turtle.`**`st`**`()`
使海龟可见。

```
>>> turtle.showturtle()
```

`turtle.`**`isvisible`**`()`
如果海龟显示返回 `True`，如果海龟隐藏返回 `False`。

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

外观

`turtle.`**`shape`**`(`*name=None*`)`

参数 name –一个有效的形状名字符串

设置海龟形状为 *name* 指定的形状名，如未指定形状名则返回当前的形状名。*name* 指定的形状名应存在于 TurtleScreen 的 shape 字典中。多边形的形状初始时有以下几种: “arrow”，“turtle”，“circle”，“square”，“triangle”，“classic”。要了解如何处理形状请参看 Screen 方法*`register_shape()`*。

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
```

(下页继续)

(续上页)

```
>>> turtle.shape()
'turtle'
```

turtle.**resizemode**(*rmode=None*)

参数 **rmode** –字符串“auto”，“user”，“noresize”其中之一

设置大小调整模式为以下值之一：“auto”，“user”，“noresize”。如未指定 *rmode* 则返回当前的大小调整模式。不同的大小调整模式的效果如下:

- “auto”：根据画笔粗细值调整海龟的外观。
- “user”：根据拉伸因子和轮廓宽度 (outline) 值调整海龟的外观，两者是由*shapesize()* 设置的。
- “noresize”：不调整海龟的外观大小。

大小调整模式（“user”）会在*shapesize()* 带参数调用时生效。

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

turtle.**shapesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)
turtle.**turtlesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)

参数

- **stretch_wid** –正数值
- **stretch_len** –正数值
- **outline** –正数值

返回或设置画笔的属性 x/y-拉伸因子和/或轮廓。设置大小调整模式为“user”。当且仅当大小调整模式设为“user”时海龟会基于其拉伸因子调整外观: *stretch_wid* 为垂直于其朝向的宽度拉伸因子，*stretch_len* 为平等于其朝向的长度拉伸因子，决定形状轮廓线的粗细。

```
>>> turtle.shapesize()
(1, 1, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

turtle.**tilt**(*angle*)

参数 **angle** –一个数值

海龟形状自其当前的倾角转动 *angle* 指定的角度，但 不改变海龟的朝向 (移动方向)。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

(下页继续)

(续上页)

```
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

turtle.**settiltangle**(*angle*)

参数 **angle** –一个数值

旋转海龟形状使其指向 *angle* 指定的方向，忽略其当前的倾角，不改变海龟的朝向 (移动方向)。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.settiltangle(45)
>>> turtle.fd(50)
>>> turtle.settiltangle(-45)
>>> turtle.fd(50)
```

turtle.**tiltangle**()

Return the current tilt-angle, i.e. the angle between the orientation of the turtleshape and the heading of the turtle (its direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

使用事件

turtle.**onclick**(*fun*, *btn=1*, *add=None*)

参数

- **fun** –一个函数，调用时将传入两个参数表示在画布上点击的坐标。
- **btn** –鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** –`True` 或 `False` –如为 `True` 则将添加一个新绑定，否则将取代先前的绑定

将 *fun* 指定的函数绑定到鼠标点击此海龟事件。如果 *fun* 值为 `None`，则移除现有的绑定。以下为使用匿名海龟即过程式的示例:

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)  # Now clicking into the turtle will turn it.
>>> onclick(None)  # event-binding will be removed
```

turtle.**onrelease**(*fun*, *btn=1*, *add=None*)

参数

- **fun** –一个函数，调用时将传入两个参数表示在画布上点击的坐标。
- **btn** –鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** –`True` 或 `False` –如为 `True` 则将添加一个新绑定，否则将取代先前的绑定

将 *fun* 指定的函数绑定到在此海龟上释放鼠标按键事件。如果 *fun* 值为 `None`，则移除现有的绑定。

```
>>> class MyTurtle(Turtle):
...     def glow(self,x,y):
...         self.fillcolor("red")
...     def unglow(self,x,y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)      # clicking on turtle turns fillcolor red,
>>> turtle.onrelease(turtle.unglow) # releasing turns it to transparent.
```

turtle.**ondrag**(*fun*, *btn=1*, *add=None*)

参数

- **fun** –一个函数，调用时将传入两个参数表示在画布上点击的坐标。
- **btn** –鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** –`True` 或 `False` –如为 `True` 则将添加一个新绑定，否则将取代先前的绑定

将 *fun* 指定的函数绑定到在此海龟上移动鼠标事件。如果 *fun* 值为 `None`，则移除现有的绑定。

注: 在海龟上移动鼠标事件之前应先发生在此海龟上点击鼠标事件。

```
>>> turtle.ondrag(turtle.goto)
```

在此之后点击并拖动海龟可在屏幕上手绘线条 (如果画笔为落下)。

turtle.**mainloop**()
turtle.**done**()
Starts event loop - calling Tkinter’s mainloop function. Must be the last statement in a turtle graphics program.

```
>>> turtle.mainloop()
```

特殊海龟方法

turtle.**begin_poly**()
开始记录多边形的顶点。当前海龟位置为多边形的第一个顶点。

turtle.**end_poly**()
停止记录多边形的顶点。当前海龟位置为多边形的最后一个顶点。它将连线到第一个顶点。

turtle.**get_poly**()
返回最新记录的多边形。

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

turtle.**clone**()
创建并返回海龟的克隆体，具有相同的位置、朝向和海龟属性。

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

turtle.**getturtle**()
turtle.**getpen**()
返回海龟对象自身。唯一合理的用法: 作为一个函数来返回“匿名海龟”：

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

turtle.**getscreen**()
返回作为海龟绘图场所的*TurtleScreen*类对象。该对象将可调用 TurtleScreen 方法。

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

turtle.**setundobuffer**(*size*)

参数 size —一个整型数值或 None

设置或禁用撤消缓冲区。如果 *size* 为一个整型数则将开辟一个指定大小的空缓冲区。*size* 表示可使用*undo()*方法/函数撤消的海龟命令的次数上限。如果 *size* 为 None 则禁用撤消缓冲区。

```
>>> turtle.setundobuffer(42)
```

turtle.**undobufferentries**()
返回撤销缓冲区里的条目数。

```
>>> while undobufferentries():
...     undo()
```

turtle.**tracer**(*flag=None*, *delay=None*)
A replica of the corresponding TurtleScreen method.

2.6 版后已移除.

turtle.**window_width**()
turtle.**window_height**()
Both are replicas of the corresponding TurtleScreen methods.

2.6 版后已移除.

Excursus about the use of compound shapes

要使用由多个不同颜色多边形构成的复合海龟形状，你必须明确地使用辅助类*Shape*，具体步骤如下:

1. 创建一个空 Shape 对象，类型为“compound”。
2. 按照需要使用 addcomponent() 方法向此对象添加多个部件。

   例如:

```
>>> s = Shape("compound")
>>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s.addcomponent(poly1, "red", "blue")
>>> poly2 = ((0,0),(10,-5),(-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```

3. 接下来将 Shape 对象添加到 Screen 对象的形状列表并使用它:

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

注解: *Shape* 类在*register_shape()* 方法的内部以多种方式使用。应用程序编写者 只有在使用上述的复合形状时才需要处理 Shape 类。

24.5.4 TurtleScreen/Screen 方法及对应函数

本节中的大部分示例都使用 TurtleScreen 类的一个实例，命名为 `screen`。

窗口控制

turtle.**bgcolor**(**args*)

> **参数 args** –一个颜色字符串或三个取值范围 0..colormode 内的数值或一个取值范围相同的数值 3 元组

设置或返回 TurtleScreen 的背景颜色。

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128, 0, 128)
```

turtle.**bgpic**(*picname=None*)

> **参数 picname** –一个字符串, gif-文件名, `"nopic"`, 或 `None`

设置背景图片或返回当前背景图片名称。如果 *picname* 为一个文件名，则将相应图片设为背景。如果 *picname* 为 `"nopic"`，则删除当前背景图片。如果 *picname* 为 `None`，则返回当前背景图片文件名。:

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

turtle.**clear**()
turtle.**clearscreen**()

从中删除所有海龟的全部绘图。将已清空的 TurtleScreen 重置为初始状态: 白色背景，无背景片，无事件绑定并启用追踪。

注解: This TurtleScreen method is available as a global function only under the name `clearscreen`. The global function `clear` is another one derived from the Turtle method `clear`.

turtle.**reset**()
turtle.**resetscreen**()
重置屏幕上的所有海龟为其初始状态。

注解: 此 TurtleScreen 方法作为全局函数时只有一个名字 `resetscreen`。全局函数 `reset` 所对应的是 Turtle 方法 `reset`。

turtle.**screensize**(*canvwidth=None*, *canvheight=None*, *bg=None*)

参数

- **canvwidth** –正整型数，以像素表示画布的新宽度值
- **canvheight** –正整型数，以像素表示画面的新高度值
- **bg** –颜色字符串或颜色元组，新的背景颜色

如未指定任何参数，则返回当前的 (canvaswidth, canvasheight)。否则改变作为海龟绘图场所的画布大小。不改变绘图窗口。要观察画布的隐藏区域，可以使用滚动条。通过此方法可以令之前绘制于画布之外的图形变为可见。

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

也可以用来寻找意外逃走的海龟;-)

turtle.**setworldcoordinates**(*llx*, *lly*, *urx*, *ury*)

参数

- **llx** –一个数值, 画布左下角的 x-坐标
- **lly** –一个数值, 画布左下角的 y-坐标
- **urx** –一个数值, 画面右上角的 x-坐标
- **ury** –一个数值, 画布右上角的 y-坐标

设置用户自定义坐标系并在必要时切换模式为“world”。这会执行一次 `screen.reset()`。如果“world”模式已激活，则所有图形将根据新的坐标系重绘。

注意: 在用户自定义坐标系中，角度可能显得扭曲。

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
>>> for _ in range(8):
...     left(45); fd(2)   # a regular octagon
```

动画控制

turtle.**delay**(*delay=None*)

参数 **delay** – 正整型数

设置或返回以毫秒数表示的延迟值 *delay*。(这约等于连续两次画布刷新的间隔时间。) 绘图延迟越长，动画速度越慢。

可选参数:

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

turtle.**tracer**(*n=None*, *delay=None*)

参数

- **n** – 非负整型数
- **delay** – 非负整型数

Turn turtle animation on/off and set delay for update drawings. If *n* is given, only each n-th regular screen update is really performed. (Can be used to accelerate the drawing of complex graphics.) Second argument sets delay value (see *delay()*).

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

turtle.**update**()

执行一次 TurtleScreen 刷新。在禁用追踪时使用。

另参见 RawTurtle/Turtle 方法*speed()*。

使用屏幕事件

turtle.**listen**(*xdummy=None*, *ydummy=None*)

设置焦点到 TurtleScreen (以便接收按键事件)。使用两个 Dummy 参数以便能够传递*listen()* 给 onclick 方法。

turtle.**onkey**(*fun*, *key*)

参数

- **fun** – 一个无参数的函数或 None
- **key** – 一个字符串: 键 (例如“a”) 或键标 (例如“space”)

绑定 *fun* 指定的函数到按键释放事件。如果 *fun* 值为 None，则移除事件绑定。注: 为了能够注册按键事件，TurtleScreen 必须得到焦点。(参见 method *listen()* 方法。)

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onclick**(*fun*, *btn=1*, *add=None*)
turtle.**onscreenclick**(*fun*, *btn=1*, *add=None*)

参数

- **fun** –一个函数，调用时将传入两个参数表示在画布上点击的坐标。
- **btn** –鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** –`True` 或 `False` –如为 `True` 则将添加一个新绑定，否则将取代先前的绑定

绑定 *fun* 指定的函数到鼠标点击屏幕事件。如果 *fun* 值为 `None`，则移除现有的绑定。

以下示例使用一个 TurtleScreen 实例 `screen` 和一个 Turtle 实例 turtle:

```
>>> screen.onclick(turtle.goto) # Subsequently clicking into the TurtleScreen will
>>>                             # make the turtle move to the clicked point.
>>> screen.onclick(None)        # remove event binding again
```

注解: 此 TurtleScreen 方法作为全局函数时只有一个名字 `onscreenclick`。全局函数 `onclick` 所对应的是 Turtle 方法 `onclick`。

---

turtle.**ontimer**(*fun*, *t=0*)

参数

- **fun** –一个无参数的函数
- **t** –一个数值 >= 0

安装一个计时器，在 *t* 毫秒后调用 *fun* 函数。

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()   ### makes the turtle march around
>>> running = False
```

设置与特殊方法

turtle.**mode**(*mode=None*)

参数 mode –字符串“standard”，“logo”或“world”其中之一

设置海龟模式（“standard”，“logo”或“world”）并执行重置。如未指定模式则返回当前的模式。

“standard”模式与旧的*turtle*兼容。”logo”模式与大部分 Logo 海龟绘图兼容。”world”模式使用用户自定义的“世界坐标系”。**注意**: 在此模式下，如果 x/y 单位比率不等于 1 则角度会显得扭曲。

模式	初始海龟朝向	正数角度
“standard”	朝右 (东)	逆时针
“logo”	朝上 (北)	顺时针

```
>>> mode("logo")   # resets turtle heading to north
>>> mode()
'logo'
```

turtle.**colormode**(*cmode=None*)

参数 cmode –数值 1.0 或 255 其中之一

返回颜色模式或将其设为 1.0 或 255。构成颜色三元组的 *r*, *g*, *b* 数值必须在 0..*cmode* 范围之内。

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
     ...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

turtle.**getcanvas**()

返回此 TurtleScreen 的 Canvas 对象。供了解 Tkinter 的 Canvas 对象内部机理的人士使用。

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas instance at 0x...>
```

turtle.**getshapes**()

返回所有当前可用海龟形状的列表。

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

turtle.**register_shape**(*name*, *shape=None*)
turtle.**addshape**(*name*, *shape=None*)

调用此函数有三种不同方式:

(1) *name* 为一个 gif 文件的文件名，*shape* 为 None: 安装相应的图像形状。:

```
>>> screen.register_shape("turtle.gif")
```

注解: 当海龟转向时图像形状 不会转动，因此无法显示海龟的朝向!

(2) *name* 为指定的字符串，*shape* 为由坐标值对构成的元组: 安装相应的多边形形状。

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
```

(3) *name* 为指定的字符串，为一个 (复合) *Shape* 类对象: 安装相应的复合形状。

将一个海龟形状加入 TurtleScreen 的形状列表。只有这样注册过的形状才能通过执行 `shape(shapename)` 命令来使用。

`turtle.`**`turtles`**`()`
返回屏幕上的海龟列表。

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

`turtle.`**`window_height`**`()`
返回海龟窗口的高度。:

```
>>> screen.window_height()
480
```

`turtle.`**`window_width`**`()`
返回海龟窗口的宽度。:

```
>>> screen.window_width()
640
```

Screen 专有方法, 而非继承自 TurtleScreen

`turtle.`**`bye`**`()`
关闭海龟绘图窗口。

`turtle.`**`exitonclick`**`()`
将 bye() 方法绑定到 Screen 上的鼠标点击事件。

如果配置字典中 “using_IDLE” 的值为 `False` (默认值) 则同时进入主事件循环。注: 如果启动 IDLE 时使用了 `-n` 开关 (无子进程)，`turtle.cfg` 中此数值应设为 `True`。在此情况下 IDLE 本身的主事件循环同样会作用于客户脚本。

`turtle.`**`setup`**`(`*width=_CFG["width"], height=_CFG["height"], startx=_CFG["leftright"], starty=_CFG["topbottom"]*`)`
设置主窗口的大小和位置。默认参数值保存在配置字典中，可通过 `turtle.cfg` 文件进行修改。

参数

- **`width`** –如为一个整型数值，表示大小为多少像素，如为一个浮点数值，则表示屏幕的占比；默认为屏幕的 50%
- **`height`** –如为一个整型数值，表示高度为多少像素，如为一个浮点数值，则表示屏幕的占比；默认为屏幕的 75%
- **`startx`** –如为正值，表示初始位置距离屏幕左边缘多少像素，负值表示距离右边缘，`None` 表示窗口水平居中
- **`starty`** –如为正值，表示初始位置距离屏幕上边缘多少像素，负值表示距离下边缘，`None` 表示窗口垂直居中

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>              # sets window to 200x200 pixels, in upper left of screen
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>              # sets window to 75% of screen by 50% of screen and centers
```

turtle.**title**(*titlestring*)

参数 titlestring –一个字符串，显示为海龟绘图窗口的标题栏文本

设置海龟窗口标题为 *titlestring* 指定的文本。

```
>>> screen.title("Welcome to the turtle zoo!")
```

24.5.5 The public classes of the module turtle

class turtle.**RawTurtle**(*canvas*)
class turtle.**RawPen**(*canvas*)

参数 canvas –a Tkinter.Canvas, a *ScrolledCanvas* or a *TurtleScreen*

创建一个海龟。海龟对象具有 “Turtle/RawTurtle 方法” 一节所述的全部方法。

class turtle.**Turtle**
RawTurtle 的子类，具有相同的接口，但其绘图场所为默认的*Screen* 类对象，在首次使用时自动创建。

class turtle.**TurtleScreen**(*cv*)

参数 cv –a Tkinter.Canvas

提供面向屏幕的方法例如 setbg() 等。说明见上文。

class turtle.**Screen**
TurtleScreen 的子类，*增加了四个方法*.

class turtle.**ScrolledCanvas**(*master*)

参数 master –可容纳 ScrolledCanvas 的 Tkinter 部件，即添加了滚动条的 Tkinter-canvas

由 Screen 类使用，使其能够自动提供一个 ScrolledCanvas 作为海龟的绘图场所。

class turtle.**Shape**(*type_*, *data*)

参数 type_ –字符串 “polygon”，“image”，“compound” 其中之一

实现形状的数据结构。(type_, data) 必须遵循以下定义:

*type_*	*data*
“polygon”	一个多边形元组，即由坐标值对构成的元组
“image”	一个图片 (此形式仅限内部使用!)
“compound”	None (复合形状必须使用*addcomponent()* 方法来构建)

addcomponent(*poly*, *fill*, *outline=None*)

参数

- **poly** –一个多边形，即由数值对构成的元组
- **fill** –一种颜色，将用来填充 *poly* 指定的多边形
- **outline** –一种颜色，用于多边形的轮廓 (如有指定)

示例:

```
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

参见*Excursus about the use of compound shapes*。

class turtle.**Vec2D**(*x*, *y*)

一个二维矢量类，用来作为实现海龟绘图的辅助类。也可能在海龟绘图程序中使用。派生自元组，因此矢量也属于元组!

提供的运算 (*a*, *b* 为矢量, *k* 为数值):

- `a + b` 矢量加法
- `a - b` 矢量减法
- `a * b` 内积
- `k * a` 和 `a * k` 与标量相乘
- `abs(a)` a 的绝对值
- `a.rotate(angle)` 旋转

24.5.6 帮助与配置

如何使用帮助

Screen 和 Turtle 类的公用方法以文档字符串提供了详细的文档。因此可以利用 Python 帮助工具获取这些在线帮助信息:

- 当使用 IDLE 时，输入函数/方法调用将弹出工具提示显示其签名和文档字符串的头几行。
- 对文法或函数调用*help()* 将显示其文档字符串:

```
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.


      >>> screen.bgcolor("orange")
      >>> screen.bgcolor()
      "orange"
      >>> screen.bgcolor(0.5,0,0.5)
      >>> screen.bgcolor()
      "#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.
```

(下页继续)

(续上页)

```
    Aliases: penup | pu | up

    No argument

    >>> turtle.penup()
```

- 方法对应函数的文档字符串的形式会有一些修改:

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.

    Example::

      >>> bgcolor("orange")
      >>> bgcolor()
      "orange"
      >>> bgcolor(0.5,0,0.5)
      >>> bgcolor()
      "#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()
```

这些修改版文档字符串是在导入时与方法对应函数的定义一起自动生成的。

文档字符串翻译为不同的语言

可使用工具创建一个字典，键为方法名，值为 Screen 和 Turtle 类公共方法的文档字符串。

turtle.**write_docstringdict**(*filename="turtle_docstringdict"*)

> **参数 filename** —一个字符串，表示文件名
>
> 创建文档字符串字典并将其写入 filename 指定的 Python 脚本文件。此函数必须显示地调用 (海龟绘图类并不使用此函数)。文档字符串字典将被写入到 Python 脚本文件 *filename*.py。该文件可作为模板用来将文档字符串翻译为不同语言。

如果你 (或你的学生) 想使用本国语言版本的*turtle* 在线帮助，你必须翻译文档字符串并保存结果文件，例如 turtle_docstringdict_german.py.

如果你在 turtle.cfg 文件中加入了相应的条目，此字典将在导入模块时被读取并替代原有的英文版文档字符串。

在撰写本文档时已经有了德语和意大利语版的文档字符串字典。(更多需求请联系 glingl@aon.at)

如何配置 Screen 和 Turtle

内置的默认配置是模仿旧 turtle 模块的外观和行为，以便尽可能地与其保持兼容。

如果你想使用不同的配置，以便更好地反映此模块的特性或是更适合你的需求，例如在课堂中使用，你可以准备一个配置文件 `turtle.cfg`，该文件将在导入模块时被读取并根据其中的设定修改模块配置。

内置的配置对应以下的 turtle.cfg:

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

选定条目的简短说明:

- 开头的四行对应 `Screen.setup()` 方法的参数。
- 第 5 和 6 行对应 `Screen.screensize()` 方法的参数。
- *shape* 可以是任何内置形状，即: arrow, turtle 等。更多信息可用 `help(shape)` 查看。
- 如果你想使用无填充色 (即令海龟变透明)，你必须写 `fillcolor = ""` (但 cfg 文件中所有非空字符串都不可加引号)。
- 如果你想令海龟反映其状态，你必须使用 `resizemode = auto`。
- 如果你设置语言例如 `language = italian` 则文档字符串字典 `turtle_docstringdict_italian.py` 将在导入模块时被加载 (如果导入路径即*turtle* 的目录中存在此文件。
- *exampleturtle* 和 *examplescreen* 条目定义了相应对象在文档字符串中显示的名称。方法文档字符串转换为函数文档字符串时将从文档字符串中删去这些名称。
- *using_IDLE*: 如果你经常使用 IDLE 并启用其 -n 开关 (“无子进程”) 则应将此项设为 `True`，这将阻止*exitonclick()* 进入主事件循环。

`turtle.cfg` 文件可以保存于*turtle* 所在目录，当前工作目录也可以有一个同名文件。后者会重载覆盖前者的设置。

The `Demo/turtle` directory contains a `turtle.cfg` file. You can study it as an example and see its effects when running the demos (preferably not from within the demo-viewer).

24.5.7 Demo scripts

There is a set of demo scripts in the turtledemo directory located in the `Demo/turtle` directory in the source distribution.

It contains:

- a set of 15 demo scripts demonstrating different features of the new module *turtle*
- a demo viewer `turtleDemo.py` which can be used to view the sourcecode of the scripts and run them at the same time. 14 of the examples can be accessed via the Examples menu; all of them can also be run standalone.
- The example `turtledemo_two_canvases.py` demonstrates the simultaneous use of two canvases with the turtle module. Therefore it only can be run standalone.
- There is a `turtle.cfg` file in this directory, which also serves as an example for how to write and use such files.

The demoscripts are:

名称	描述	相关特性
bytedesign	complex classical turtlegraphics pattern	*tracer()*, delay, *update()*
chaos	绘制 Verhulst 动态模型，演示通过计算机的运算可能会生成令人惊叹的结果	世界坐标系
clock	绘制模拟时钟显示本机的当前时间	海龟作为表针, ontimer
colormixer	试验 r, g, b 颜色模式	*ondrag()* 当鼠标拖动
fractalcurves	绘制 Hilbert & Koch 曲线	递归
lindenmayer	文化数学 (印度装饰艺术)	L-系统
minimal_hanoi	汉诺塔	矩形海龟作为汉诺盘 (shape, shapesize)
paint	超极简主义绘画程序	*onclick()* 当鼠标点击
peace	初级技巧	海龟: 外观与动画
penrose	非周期性地使用风筝和飞镖形状铺满平面	*stamp()* 印章
planet_and_moon	模拟引力系统	复合开关, *Vec2D* 类
tree	一棵 (图形化的) 广度优先树 (使用生成器)	*clone()* 克隆
wikipedia	一个来自介绍海龟绘图的维基百科文章的图案	*clone()*, *undo()*
yinyang	另一个初级示例	*circle()* 画圆

祝你玩得开心!

24.6 IDLE

IDLE 是 Python 所内置的开发与学习环境。

IDLE 具有以下特性:

- 编码于 100% 纯正的 Python，使用名为 `tkinter` 的图形用户界面工具
- cross-platform: works mostly the same on Windows, Unix, and Mac OS X
- 提供输入输出高亮和错误信息的 Python 命令行窗口（交互解释器）
- 提供多次撤销操作、Python 语法高亮、智能缩进、函数调用提示、自动补全等功能的多窗口文本编辑器
- 在多个窗口中检索，在编辑器中替换文本，以及在多个文件中检索（通过 grep）
- 提供持久保存的断点调试、单步调试、查看本地和全局命名空间功能的调试器

- 配置、浏览以及其它对话框

24.6.1 目录

IDLE has two main window types, the Shell window and the Editor window. It is possible to have multiple editor windows simultaneously. Output windows, such as used for Edit / Find in Files, are a subtype of edit window. They currently have the same top menu as Editor windows but a different default title and context menu.

IDLE's menus dynamically change based on which window is currently selected. Each menu documented below indicates which window type it is associated with.

文件目录（命令行和编辑器）

新建文件 创建一个文件编辑器窗口。

打开… 使用打开窗口以打开一个已存在的文件。

近期文件 打开一个近期文件列表，选取一个以打开它。

打开模块… 打开一个已存在的模块（搜索 sys.path）

类浏览器 于当前所编辑的文件中使用树形结构展示函数、类以及方法。在命令行中，首先打开一个模块。

路径浏览 在树状结构中展示 sys.path 目录、模块、函数、类和方法。

保存 如果文件已经存在，则将当前窗口保存至对应的文件。自打开或上次保存之后经过修改的文件的窗口标题栏首尾将出现星号 * 。如果没有对应的文件，则使用“另存为”代替。

保存为… 使用“保存为”对话框保存当前窗口。被保存的文件将作为当前窗口新的对应文件。

另存为副本… 保存当前窗口至另一个文件，而不修改当前对应文件。

打印窗口 通过默认打印机打印当前窗口。

关闭 关闭当前窗口（如果未保存则询问）。

退出 关闭所有窗口并退出 IDLE（如果未保存则询问）

编辑目录（命令行和编辑器）

撤销操作 撤销当前窗口的最近一次操作。最高可以撤回 1000 条操作记录。

重做 重做当前窗口最近一次所撤销的操作。

剪切 复制选区至系统剪贴板，然后删除选区。

复制 复制选区至系统剪贴板。

粘贴 插入系统剪贴板的内容至当前窗口。

剪贴板功能也可用于上下文目录。

全选 选择当前窗口的全部内容。

查找… 打开一个提供多选项的查找窗口。

再次查找 重复上次搜索，如果结果存在。

查找选区 查找当前选中的字符串，如果存在

在文件中查找… 打开文件查找对话框。将结果输出至新的输出窗口。

替换… 打开查找并替换对话框。

前往行 Move cursor to the line number requested and make that line visible.

提示完成 Open a scrollable list allowing selection of keywords and attributes. See Completions in the Tips sections below.

展开文本 展开键入的前缀以匹配同一窗口中的完整单词；重复以获得不同的扩展。

显示调用贴士 After an unclosed parenthesis for a function, open a small window with function parameter hints.

显示周围括号 突出显示周围的括号。

格式菜单（仅 window 编辑器）

缩进区域 将选定的行右移缩进宽度（默认为 4 个空格）。

区域减少缩进 将选定的行向左移动缩进宽度（默认为 4 个空格）。

区域注释 在所选行的前面插入 ##。

区域取消注释 从所选行中删除开头的#或 ##。

区域添加制表符 将 前导空格变成制表符。（注意：我们建议使用 4 个空格来缩进 Python 代码。）

区域取消制表符 将 所有制表符转换为正确的空格数。

切换标签 打开一个对话框，以在缩进和空格之间切换。

新缩进宽度 打开一个对话框以更改缩进宽度。Python 社区接受的默认值为 4 个空格。

格式段落 在注释块或多行字符串或字符串中的选定行中，重新格式化当前以空行分隔的段落。段落中的所有行的格式都将少于 N 列，其中 N 默认为 72。

尾随空格 Remove any space characters after the last non-space character of a line.

运行菜单（仅 window 编辑器）

Python Shell 打开或唤醒 Python Shell 窗口。

检查模块 检查“编辑器”窗口中当前打开的模块的语法。如果尚未保存该模块，则 IDLE 会提示用户保存或自动保存，如在“空闲设置”对话框的“常规”选项卡中所选择的那样。如果存在语法错误，则会在“编辑器”窗口中指示大概位置。

运行模块 Do Check Module (above). If no error, restart the shell to clean the environment, then execute the module. Output is displayed in the Shell window. Note that output requires use of `print` or `write`. When execution is complete, the Shell retains focus and displays a prompt. At this point, one may interactively explore the result of execution. This is similar to executing a file with `python -i file` at a command line.

Shell 菜单（仅 window 编辑器）

查看最近重启 将 Shell 窗口滚动到上一次 Shell 重启。

重启 Shell 重新启动 shell 以清理环境。

中断执行 停止正在运行的程序。

调试菜单（仅 window 编辑器）

跳转到文件/行 Look on the current line. with the cursor, and the line above for a filename and line number. If found, open the file if not already open, and show the line. Use this to view source lines referenced in an exception traceback and lines found by Find in Files. Also available in the context menu of the Shell window and Output windows.

调试器（切换） 激活后，在 Shell 中输入的代码或从编辑器中运行的代码将在调试器下运行。在编辑器中，可以使用上下文菜单设置断点。此功能不完整，具有实验性。

堆栈查看器 在树状目录中显示最后一个异常的堆栈回溯，可以访问本地和全局。

自动打开堆栈查看器 在未处理的异常上切换自动打开堆栈查看器。

选项菜单（命令行和编辑器）

配置 IDLE Open a configuration dialog and change preferences for the following: fonts, indentation, keybindings, text color themes, startup windows and size, additional help sources, and extensions (see below). On OS X, open the configuration dialog by selecting Preferences in the application menu. To use a new built-in color theme (IDLE Dark) with older IDLEs, save it as a new custom theme.

Non-default user settings are saved in a .idlerc directory in the user’s home directory. Problems caused by bad user configuration files are solved by editing or deleting one or more of the files in .idlerc.

Code Context (toggle)(Editor Window only) Open a pane at the top of the edit window which shows the block context of the code which has scrolled above the top of the window.

Window 菜单（命令行和编辑器）

Zoom Height Toggles the window between normal size and maximum height. The initial size defaults to 40 lines by 80 chars unless changed on the General tab of the Configure IDLE dialog.

The rest of this menu lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).

帮助菜单（命令行和编辑器）

关于 IDLE 显示版本，版权，许可证，荣誉等。

IDLE 帮助 Display a help file for IDLE detailing the menu options, basic editing and navigation, and other tips.

Python 文档 访问本地 Python 文档（如果已安装），或启动 Web 浏览器并打开 docs.python.org 显示最新的 Python 文档。

海龟演示 Run the turtledemo module with example python code and turtle drawings.

Additional help sources may be added here with the Configure IDLE dialog under the General tab.

上下文菜单

Open a context menu by right-clicking in a window (Control-click on OS X). Context menus have the standard clipboard functions also on the Edit menu.

剪切 复制选区至系统剪贴板，然后删除选区。

复制 复制选区至系统剪贴板。

粘贴 插入系统剪贴板的内容至当前窗口。

Editor windows also have breakpoint functions. Lines with a breakpoint set are specially marked. Breakpoints only have an effect when running under the debugger. Breakpoints for a file are saved in the user's .idlerc directory.

设置断点 在当前行设置断点

清除断点 清除当前行断点

Shell and Output windows have the following.

跳转到文件/行 与调试菜单相同。

24.6.2 编辑和导航

In this section, 'C' refers to the `Control` key on Windows and Unix and the `Command` key on Mac OSX.

- `Backspace` 向左删除; `Del` 向右删除
- `C-Backspace` 向左删除单词; `C-Del` 向右删除单词
- 方向键和 `Page Up`/`Page Down` 移动
- `C-LeftArrow` 和 `C-RightArrow` 按字移动
- `Home`/`End` 跳转到行首/尾
- `C-Home`/`C-End` 跳转到文档首/尾
- 一些有用的 Emacs 绑定是从 Tcl / Tk 继承的:
  - `C-a` 行首
  - `C-e` 行尾
  - `C-k` 删除行（但未将其放入剪贴板）
  - `C-l` center window around the insertion point
  - `C-b` go backward one character without deleting (usually you can also use the cursor key for this)
  - `C-f` go forward one character without deleting (usually you can also use the cursor key for this)
  - `C-p` go up one line (usually you can also use the cursor key for this)
  - `C-d` 删除下一个字符

Standard keybindings (like `C-c` to copy and `C-v` to paste) may work. Keybindings are selected in the Configure IDLE dialog.

自动缩进

After a block-opening statement, the next line is indented by 4 spaces (in the Python Shell window by one tab). After certain keywords (break, return etc.) the next line is dedented. In leading indentation, `Backspace` deletes up to 4 spaces if they are there. `Tab` inserts spaces (in the Python Shell window one tab), number depends on Indent width. Currently, tabs are restricted to four spaces due to Tcl/Tk limitations.

See also the indent/dedent region commands in the edit menu.

完成

Completions are supplied for functions, classes, and attributes of classes, both built-in and user-defined. Completions are also provided for filenames.

The AutoCompleteWindow (ACW) will open after a predefined delay (default is two seconds) after a '.' or (in a string) an os.sep is typed. If after one of those characters (plus zero or more other characters) a tab is typed the ACW will open immediately if a possible continuation is found.

If there is only one possible completion for the characters entered, a `Tab` will supply that completion without opening the ACW.

'Show Completions' will force open a completions window, by default the `C-space` will open a completions window. In an empty string, this will contain the files in the current directory. On a blank line, it will contain the built-in and user-defined functions and classes in the current namespaces, plus any modules imported. If some characters have been entered, the ACW will attempt to be more specific.

If a string of characters is typed, the ACW selection will jump to the entry most closely matching those characters. Entering a `tab` will cause the longest non-ambiguous match to be entered in the Editor window or Shell. Two `tab` in a row will supply the current ACW selection, as will return or a double click. Cursor keys, Page Up/Down, mouse selection, and the scroll wheel all operate on the ACW.

"Hidden" attributes can be accessed by typing the beginning of hidden name after a '.', e.g. '_'. This allows access to modules with `__all__` set, or to class-private attributes.

Completions and the 'Expand Word' facility can save a lot of typing!

Completions are currently limited to those in the namespaces. Names in an Editor window which are not via `__main__` and *`sys.modules`* will not be found. Run the module once with your imports to correct this situation. Note that IDLE itself places quite a few modules in sys.modules, so much can be found by default, e.g. the re module.

If you don't like the ACW popping up unbidden, simply make the delay longer or disable the extension.

提示

A calltip is shown when one types `(` after the name of an *accessible* function. A name expression may include dots and subscripts. A calltip remains until it is clicked, the cursor is moved out of the argument area, or `)` is typed. When the cursor is in the argument part of a definition, the menu or shortcut display a calltip.

A calltip consists of the function signature and the first line of the docstring. For builtins without an accessible signature, the calltip consists of all lines up the fifth line or the first blank line. These details may change.

The set of *accessible* functions depends on what modules have been imported into the user process, including those imported by Idle itself, and what definitions have been run, all since the last restart.

For example, restart the Shell and enter `itertools.count(`. A calltip appears because Idle imports itertools into the user process for its own use. (This could change.) Enter `turtle.write(` and nothing appears. Idle does not import turtle. The menu or shortcut do nothing either. Enter `import turtle` and then `turtle.write(` will work.

In an editor, import statements have no effect until one runs the file. One might want to run a file after writing the import statements at the top, or immediately run an existing file before editing.

Python Shell 窗口

- C-c 中断执行命令
- C-d sends end-of-file; closes window if typed at a >>> prompt
- Alt-/ (Expand word) is also useful to reduce typing

  历史命令

  - Alt-p retrieves previous command matching what you have typed. On OS X use C-p.
  - Alt-n retrieves next. On OS X use C-n.
  - Return while on any previous command retrieves that command

文本颜色

Idle defaults to black on white text, but colors text with special meanings. For the shell, these are shell output, shell error, user output, and user error. For Python code, at the shell prompt or in an editor, these are keywords, builtin class and function names, names following `class` and `def`, strings, and comments. For any text window, these are the cursor (when present), found text (when possible), and selected text.

Text coloring is done in the background, so uncolorized text is occasionally visible. To change the color scheme, use the Configure IDLE dialog Highlighting tab. The marking of debugger breakpoint lines in the editor and text in popups and dialogs is not user-configurable.

24.6.3 启动和代码执行

Upon startup with the -s option, IDLE will execute the file referenced by the environment variables `IDLESTARTUP` or `PYTHONSTARTUP`. IDLE first checks for `IDLESTARTUP`; if `IDLESTARTUP` is present the file referenced is run. If `IDLESTARTUP` is not present, IDLE checks for `PYTHONSTARTUP`. Files referenced by these environment variables are convenient places to store functions that are used frequently from the IDLE shell, or for executing import statements to import common modules.

In addition, `Tk` also loads a startup file if it is present. Note that the Tk file is loaded unconditionally. This additional file is `.Idle.py` and is looked for in the user's home directory. Statements in this file will be executed in the Tk namespace, so this file is not useful for importing functions to be used from IDLE's Python shell.

命令行语法

```
idle.py [-c command] [-d] [-e] [-h] [-i] [-r file] [-s] [-t title] [-] [arg] ...

-c command  run command in the shell window
-d          enable debugger and open shell window
-e          open editor window
-h          print help message with legal combinations and exit
-i          open shell window
-r file     run file in shell window
-s          run $IDLESTARTUP or $PYTHONSTARTUP first, in shell window
-t title    set title of shell window
-           run stdin in shell (- must be last option before args)
```

如果有参数：

- If -, -c, or r is used, all arguments are placed in sys.argv[1:...] and sys.argv[0] is set to '', '-c', or '-r'. No editor window is opened, even if that is the default set in the Options dialog.
- Otherwise, arguments are files opened for editing and sys.argv reflects the arguments passed to IDLE itself.

IDLE-console differences

As much as possible, the result of executing Python code with IDLE is the same as executing the same code in a console window. However, the different interface and operation occasionally affect visible results. For instance, sys.modules starts with more entries.

IDLE also replaces sys.stdin, sys.stdout, and sys.stderr with objects that get input from and send output to the Shell window. When this window has the focus, it controls the keyboard and screen. This is normally transparent, but functions that directly access the keyboard and screen will not work. If sys is reset with reload(sys), IDLE's changes are lost and things like input, raw_input, and print will not work correctly.

With IDLE's Shell, one enters, edits, and recalls complete statements. Some consoles only work with a single physical line at a time. IDLE uses exec to run each statement. As a result, '__builtins__' is always defined for each statement.

在没有子进程的情况下运行

By default, IDLE executes user code in a separate subprocess via a socket, which uses the internal loopback interface. This connection is not externally visible and no data is sent to or received from the Internet. If firewall software complains anyway, you can ignore it.

If the attempt to make the socket connection fails, Idle will notify you. Such failures are sometimes transient, but if persistent, the problem may be either a firewall blocking the connection or misconfiguration of a particular system. Until the problem is fixed, one can run Idle with the -n command line switch.

If IDLE is started with the -n command line switch it will run in a single process and will not create the subprocess which runs the RPC Python execution server. This can be useful if Python cannot create the subprocess or the RPC socket interface on your platform. However, in this mode user code is not isolated from IDLE itself. Also, the environment is not restarted when Run/Run Module (F5) is selected. If your code has been modified, you must reload() the affected modules and re-import any specific items (e.g. from foo import baz) if the changes are to take effect. For these reasons, it is preferable to run IDLE with the default subprocess if at all possible.

3.4 版后已移除.

24.6.4 帮助和偏好

Additional help sources

IDLE includes a help menu entry called "Python Docs" that will open the extensive sources of help, including tutorials, available at docs.python.org. Selected URLs can be added or removed from the help menu at any time using the Configure IDLE dialog. See the IDLE help option in the help menu of IDLE for more information.

偏好设定

The font preferences, highlighting, keys, and general preferences can be changed via Configure IDLE on the Option menu. Keys can be user defined; IDLE ships with four built-in key sets. In addition, a user can create a custom key set in the Configure IDLE dialog under the keys tab.

扩展

IDLE contains an extension facility. Preferences for extensions can be changed with Configure Extensions. See the beginning of config-extensions.def in the idlelib directory for further information. The default extensions are currently:

- FormatParagraph
- AutoExpand
- ZoomHeight
- ScriptBinding
- CallTips
- ParenMatch
- AutoComplete
- CodeContext
- RstripExtension

24.7 其他图形用户界面（GUI）包

Python 可用的主要跨平台（Windows，Mac OS X，类 Unix）GUI 工具：

参见:

PyGTK is a set of bindings for the GTK widget set. It provides an object oriented interface that is slightly higher level than the C one. It comes with many more widgets than Tkinter provides, and has good Python-specific reference documentation. There are also bindings to GNOME. An online tutorial is available.

PyQt PyQt is a **sip**-wrapped binding to the Qt toolkit. Qt is an extensive C++ GUI application development framework that is available for Unix, Windows and Mac OS X. **sip** is a tool for generating bindings for C++ libraries as Python classes, and is specifically designed for Python. The *PyQt3* bindings have a book, GUI Programming with Python: QT Edition by Boudewijn Rempt. The *PyQt4* bindings also have a book, Rapid GUI Programming with Python and Qt, by Mark Summerfield.

wxPython wxPython is a cross-platform GUI toolkit for Python that is built around the popular wxWidgets (formerly wxWindows) C++ toolkit. It provides a native look and feel for applications on Windows, Mac OS X, and Unix systems by using each platform's native widgets where ever possible, (GTK+ on Unix-like systems). In addition to an extensive set of widgets, wxPython provides classes for online documentation and context sensitive help, printing, HTML viewing, low-level device context drawing, drag and drop, system clipboard access, an XML-based resource format and more, including an ever growing library of user-contributed modules. wxPython has a book, wxPython in Action, by Noel Rappin and Robin Dunn.

PyGTK, PyQt 和 wxPython 都拥有比 Tkinter 更现代的外观效果和更多的可视化部件。此外还存在许多其他适用于 Python 的 GUI 工具集，既有跨平台的，也有特定平台专属的。请参阅 Python Wiki 中的 GUI 编程 页面查看更完整的列表，以及不同 GUI 工具集对比文档的链接。

CHAPTER 25

开发工具

本章中描述的各模块可帮你编写 Python 程序。例如，*pydoc* 模块接受一个模块并根据该模块的内容来生成文档。*doctest* 和*unittest* 这两个模块包含了用于编写单元测试的框架，并可用于自动测试所编写的代码，验证预期的输出是否产生。**2to3** 程序能够将 Python 2.x 源代码翻译成有效的 Python 3.x 源代码。

本章中描述的模块列表是：

25.1 pydoc —Documentation generator and online help system

2.1 新版功能.

Source code: Lib/pydoc.py

The *pydoc* module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a Web browser, or saved to HTML files.

For modules, classes, functions and methods, the displayed documentation is derived from the docstring (i.e. the __doc__ attribute) of the object, and recursively of its documentable members. If there is no docstring, *pydoc* tries to obtain a description from the block of comment lines just above the definition of the class, function or method in the source file, or at the top of the module (see *inspect.getcomments()*).

The built-in function *help()* invokes the online help system in the interactive interpreter, which uses *pydoc* to generate its documentation as text on the console. The same text documentation can also be viewed from outside the Python interpreter by running **pydoc** as a script at the operating system's command prompt. For example, running

```
pydoc sys
```

at a shell prompt will display documentation on the *sys* module, in a style similar to the manual pages shown by the Unix **man** command. The argument to **pydoc** can be the name of a function, module, or package, or a dotted reference to a class, method, or function within a module or module in a package. If the argument to **pydoc** looks like a path (that is, it contains the path separator for your operating system, such as a slash in Unix), and refers to an existing Python source file, then documentation is produced for that file.

注解: In order to find objects and their documentation, *pydoc* imports the module(s) to be documented. Therefore, any code on module level will be executed on that occasion. Use an `if __name__ == '__main__':` guard to only execute code when a file is invoked as a script and not just imported.

When printing output to the console, **pydoc** attempts to paginate the output for easier reading. If the `PAGER` environment variable is set, **pydoc** will use its value as a pagination program.

Specifying a `-w` flag before the argument will cause HTML documentation to be written out to a file in the current directory, instead of displaying text on the console.

Specifying a `-k` flag before the argument will search the synopsis lines of all available modules for the keyword given as the argument, again in a manner similar to the Unix **man** command. The synopsis line of a module is the first line of its documentation string.

You can also use **pydoc** to start an HTTP server on the local machine that will serve documentation to visiting Web browsers. **pydoc -p 1234** will start a HTTP server on port 1234, allowing you to browse the documentation at `http://localhost:1234/` in your preferred Web browser. **pydoc -g** will start the server and additionally bring up a small *Tkinter*-based graphical interface to help you search for documentation pages.

When **pydoc** generates documentation, it uses the current environment and path to locate modules. Thus, invoking **pydoc spam** documents precisely the version of the module you would get if you started the Python interpreter and typed `import spam`.

Module docs for core modules are assumed to reside in https://docs.python.org/library/. This can be overridden by setting the `PYTHONDOCS` environment variable to a different URL or to a local directory containing the Library Reference Manual pages.

25.2 doctest —测试交互性的 Python 示例

doctest 模块寻找像 Python 交互式代码的文本，然后执行这些代码来确保它们的确就像展示的那样正确运行，有许多方法来使用 doctest：

- 通过验证所有交互式示例仍然按照记录的方式工作，以此来检查模块的文档字符串是否是最新的。
- To perform regression testing by verifying that interactive examples from a test file or a test object work as expected.
- To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of "literate testing" or "executable documentation".

下面是一个小却完整的示例模块:

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    If the result is small enough to fit in an int, return an int.
    Else return a long.
```

(下页继续)

(续上页)

```
    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> [factorial(long(n)) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000L
    >>> factorial(30L)
    265252859812191058636308480000000L
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000L

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

如果你直接在命令行里运行 `example.py`，*doctest* 将发挥它的作用。

```
$ python example.py
$
```

There's no output! That's normal, and it means all the examples worked. Pass `-v` to the script, and *doctest* prints a detailed log of what it's trying, and prints a summary at the end:

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
Trying:
    [factorial(long(n)) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

And so on, eventually ending with:

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
   1 tests in __main__
   8 tests in __main__.factorial
9 tests in 2 items.
9 passed and 0 failed.
Test passed.
$
```

That's all you need to know to start making productive use of *doctest*! Jump in. The following sections provide full details. Note that there are many examples of doctests in the standard Python test suite and libraries. Especially useful examples can be found in the standard test file `Lib/test/test_doctest.py`.

25.2.1 简单用法：检查 Docstrings 中的示例

开始使用 doctest 的最简单方法（但不一定是你将继续这样做的方式）是结束每个模块 `M` 使用:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

doctest then examines docstrings in module `M`.

Running the module as a script causes the examples in the docstrings to get executed and verified:

```
python M.py
```

This won't display anything unless an example fails, in which case the failing example(s) and the cause(s) of the failure(s) are printed to stdout, and the final line of output is `***Test Failed*** N failures.`, where *N* is the number of examples that failed.

Run it with the `-v` switch instead:

```
python M.py -v
```

and a detailed report of all examples tried is printed to standard output, along with assorted summaries at the end.

You can force verbose mode by passing `verbose=True` to *`testmod()`*, or prohibit it by passing `verbose=False`. In either of those cases, `sys.argv` is not examined by *`testmod()`* (so passing `-v` or not has no effect).

Since Python 2.6, there is also a command line shortcut for running *`testmod()`*. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the module name(s) on the command line:

```
python -m doctest -v example.py
```

This will import `example.py` as a standalone module and run *`testmod()`* on it. Note that this may not work correctly if the file is part of a package and imports other submodules from that package.

For more information on *`testmod()`*, see section *Basic API*.

25.2.2 Simple Usage: Checking Examples in a Text File

Another simple application of doctest is testing interactive examples in a text file. This can be done with the *`testfile()`* function:

```
import doctest
doctest.testfile("example.txt")
```

That short script executes and verifies any interactive Python examples contained in the file `example.txt`. The file content is treated as if it were a single giant docstring; the file doesn't need to contain a Python program! For example, perhaps `example.txt` contains this:

```
The ``example`` module
======================

Using ``factorial``
-------------------

This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

Running `doctest.testfile("example.txt")` then finds the error in this documentation:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

As with *`testmod()`*, *`testfile()`* won't display anything unless an example fails. If an example does fail, then the failing example(s) and the cause(s) of the failure(s) are printed to stdout, using the same format as *`testmod()`*.

By default, `testfile()` looks for files in the calling module's directory. See section *Basic API* for a description of the optional arguments that can be used to tell it to look for files in other locations.

Like `testmod()`, `testfile()`'s verbosity can be set with the `-v` command-line switch or with the optional keyword argument *verbose*.

Since Python 2.6, there is also a command line shortcut for running `testfile()`. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the file name(s) on the command line:

```
python -m doctest -v example.txt
```

Because the file name does not end with `.py`, `doctest` infers that it must be run with `testfile()`, not `testmod()`.

For more information on `testfile()`, see section *Basic API*.

25.2.3 How It Works

This section examines in detail how doctest works: which docstrings it looks at, how it finds interactive examples, what execution context it uses, how it handles exceptions, and how option flags can be used to control its behavior. This is the information that you need to know to write doctest examples; for information about actually running doctest on these examples, see the following sections.

Which Docstrings Are Examined?

The module docstring, and all function, class and method docstrings are searched. Objects imported into the module are not searched.

In addition, if `M.__test__` exists and "is true", it must be a dict, and each entry maps a (string) name to a function object, class object, or string. Function and class object docstrings found from `M.__test__` are searched, and strings are treated as if they were docstrings. In output, a key `K` in `M.__test__` appears with name

```
<name of M>.__test__.K
```

Any classes found are recursively searched similarly, to test docstrings in their contained methods and nested classes.

在 2.4 版更改: A "private name" concept is deprecated and no longer documented.

How are Docstring Examples Recognized?

In most cases a copy-and-paste of an interactive console session works fine, but doctest isn't trying to do an exact emulation of any specific Python shell.

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print "yes"
... else:
...     print "no"
...     print "NO"
...     print "NO!!!"
...
no
NO
```

(下页继续)

(续上页)

```
NO!!!
>>>
```

Any expected output must immediately follow the final `'>>> '` or `'... '` line containing the code, and the expected output (if any) extends to the next `'>>> '` or all-whitespace line.

The fine print:

- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output. If expected output does contain a blank line, put `<BLANKLINE>` in your doctest example each place a blank line is expected.

  2.4 新版功能: `<BLANKLINE>` was added; there was no way to use expected output containing empty lines in previous versions.

- All hard tab characters are expanded to spaces, using 8-column tab stops. Tabs in output generated by the tested code are not modified. Because any hard tabs in the sample output *are* expanded, this means that if the code output includes hard tabs, the only way the doctest can pass is if the *NORMALIZE_WHITESPACE* option or *directive* is in effect. Alternatively, the test can be rewritten to capture the output and compare it to an expected value as part of the test. This handling of tabs in the source was arrived at through trial and error, and has proven to be the least error prone way of handling them. It is possible to use a different algorithm for handling tabs by writing a custom *DocTestParser* class.

  在 2.4 版更改: Expanding tabs to spaces is new; previous versions tried to preserve hard tabs, with confusing results.

- Output to stdout is captured, but not output to stderr (exception tracebacks are captured via a different means).

- If you continue a line via backslashing in an interactive session, or for any other reason use a backslash, you should use a raw docstring, which will preserve your backslashes exactly as you type them:

  ```
  >>> def f(x):
  ...     r'''Backslashes in a raw docstring: m\n'''
  >>> print f.__doc__
  Backslashes in a raw docstring: m\n
  ```

  Otherwise, the backslash will be interpreted as part of the string. For example, the `\n` above would be interpreted as a newline character. Alternatively, you can double each backslash in the doctest version (and not use a raw string):

  ```
  >>> def f(x):
  ...     '''Backslashes in a raw docstring: m\\n'''
  >>> print f.__doc__
  Backslashes in a raw docstring: m\n
  ```

- The starting column doesn't matter:

  ```
  >>> assert "Easy!"
        >>> import math
            >>> math.floor(1.9)
            1.0
  ```

  and as many leading whitespace characters are stripped from the expected output as appeared in the initial `'>>> '` line that started the example.

What's the Execution Context?

By default, each time *doctest* finds a docstring to test, it uses a *shallow copy* of M's globals, so that running tests doesn't change the module's real globals, and so that one test in M can't leave behind crumbs that accidentally allow another test to work. This means examples can freely use any names defined at top-level in M, and names defined earlier in the docstring being run. Examples cannot see names defined in other docstrings.

You can force use of your own dict as the execution context by passing `globs=your_dict` to *testmod()* or *testfile()* instead.

What About Exceptions?

No problem, provided that the traceback is the only output produced by the example: just paste in the traceback.[1] Since tracebacks contain details that are likely to change rapidly (for example, exact file paths and line numbers), this is one case where doctest works hard to be flexible in what it accepts.

Simple example:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

That doctest succeeds if *ValueError* is raised, with the `list.remove(x): x not in list` detail as shown.

The expected output for an exception must start with a traceback header, which may be either of the following two lines, indented the same as the first line of the example:

```
Traceback (most recent call last):
Traceback (innermost last):
```

The traceback header is followed by an optional traceback stack, whose contents are ignored by doctest. The traceback stack is typically omitted, or copied verbatim from an interactive session.

The traceback stack is followed by the most interesting part: the line(s) containing the exception type and detail. This is usually the last line of a traceback, but can extend across multiple lines if the exception has a multi-line detail:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

The last three lines (starting with *ValueError*) are compared against the exception's type and detail, and the rest are ignored.

在 2.4 版更改: Previous versions were unable to handle multi-line exception details.

Best practice is to omit the traceback stack, unless it adds significant documentation value to the example. So the last example is probably better as:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
```

(下页继续)

[1] Examples containing both expected output and an exception are not supported. Trying to guess where one ends and the other begins is too error-prone, and that also makes for a confusing test.

(续上页)

```
ValueError: multi
    line
detail
```

Note that tracebacks are treated very specially. In particular, in the rewritten example, the use of `...` is independent of doctest's *ELLIPSIS* option. The ellipsis in that example could be left out, or could just as well be three (or three hundred) commas or digits, or an indented transcript of a Monty Python skit.

Some details you should read once, but won't need to remember:

- Doctest can't guess whether your expected output came from an exception traceback or from ordinary printing. So, e.g., an example that expects `ValueError: 42 is prime` will pass whether *ValueError* is actually raised or if the example merely prints that traceback text. In practice, ordinary output rarely begins with a traceback header line, so this doesn't create real problems.
- Each line of the traceback stack (if present) must be indented further than the first line of the example, *or* start with a non-alphanumeric character. The first line following the traceback header indented the same and starting with an alphanumeric is taken to be the start of the exception detail. Of course this does the right thing for genuine tracebacks.
- When the *IGNORE_EXCEPTION_DETAIL* doctest option is specified, everything following the leftmost colon and any module information in the exception name is ignored.
- The interactive shell omits the traceback header line for some *SyntaxError*s. But doctest uses the traceback header line to distinguish exceptions from non-exceptions. So in the rare case where you need to test a *SyntaxError* that omits the traceback header, you will need to manually add the traceback header line to your test example.
- For some *SyntaxError*s, Python displays the character position of the syntax error, using a ^ marker:

  ```
  >>> 1 1
    File "<stdin>", line 1
      1 1
        ^
  SyntaxError: invalid syntax
  ```

  Since the lines showing the position of the error come before the exception type and detail, they are not checked by doctest. For example, the following test would pass, even though it puts the ^ marker in the wrong location:

  ```
  >>> 1 1
    File "<stdin>", line 1
      1 1
      ^
  SyntaxError: invalid syntax
  ```

Option Flags

A number of option flags control various aspects of doctest's behavior. Symbolic names for the flags are supplied as module constants, which can be bitwise ORed together and passed to various functions. The names can also be used in *doctest directives*.

The first group of options define test semantics, controlling aspects of how doctest decides whether actual output matches an example's expected output:

`doctest.`**DONT_ACCEPT_TRUE_FOR_1**

By default, if an expected output block contains just `1`, an actual output block containing just `1` or just `True` is considered to be a match, and similarly for `0` versus `False`. When *DONT_ACCEPT_TRUE_FOR_1* is specified,

neither substitution is allowed. The default behavior caters to that Python changed the return type of many functions from integer to boolean; doctests expecting "little integer" output still work in these cases. This option will probably go away, but not for several years.

doctest.**DONT_ACCEPT_BLANKLINE**
By default, if an expected output block contains a line containing only the string `<BLANKLINE>`, then that line will match a blank line in the actual output. Because a genuinely blank line delimits the expected output, this is the only way to communicate that a blank line is expected. When *DONT_ACCEPT_BLANKLINE* is specified, this substitution is not allowed.

doctest.**NORMALIZE_WHITESPACE**
When specified, all sequences of whitespace (blanks and newlines) are treated as equal. Any sequence of whitespace within the expected output will match any sequence of whitespace within the actual output. By default, whitespace must match exactly. *NORMALIZE_WHITESPACE* is especially useful when a line of expected output is very long, and you want to wrap it across multiple lines in your source.

doctest.**ELLIPSIS**
When specified, an ellipsis marker (`...`) in the expected output can match any substring in the actual output. This includes substrings that span line boundaries, and empty substrings, so it's best to keep usage of this simple. Complicated uses can lead to the same kinds of "oops, it matched too much!" surprises that `.*` is prone to in regular expressions.

doctest.**IGNORE_EXCEPTION_DETAIL**
When specified, an example that expects an exception passes if an exception of the expected type is raised, even if the exception detail does not match. For example, an example expecting `ValueError: 42` will pass if the actual exception raised is `ValueError: 3*14`, but will fail, e.g., if *TypeError* is raised.

It will also ignore the module name used in Python 3 doctest reports. Hence both of these variations will work with the flag specified, regardless of whether the test is run under Python 2.7 or Python 3.2 (or later versions):

```
>>> raise CustomError('message')
Traceback (most recent call last):
CustomError: message

>>> raise CustomError('message')
Traceback (most recent call last):
my_module.CustomError: message
```

Note that *ELLIPSIS* can also be used to ignore the details of the exception message, but such a test may still fail based on whether or not the module details are printed as part of the exception name. Using *IGNORE_EXCEPTION_DETAIL* and the details from Python 2.3 is also the only clear way to write a doctest that doesn't care about the exception detail yet continues to pass under Python 2.3 or earlier (those releases do not support *doctest directives* and ignore them as irrelevant comments). For example:

```
>>> (1, 2)[3] = 'moo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object doesn't support item assignment
```

passes under Python 2.3 and later Python versions with the flag specified, even though the detail changed in Python 2.4 to say "does not" instead of "doesn't".

在 2.7 版更改: *IGNORE_EXCEPTION_DETAIL* now also ignores any information relating to the module containing the exception under test

doctest.**SKIP**
When specified, do not run the example at all. This can be useful in contexts where doctest examples serve as both documentation and test cases, and an example should be included for documentation purposes, but should not be

checked. E.g., the example's output might be random; or the example might depend on resources which would be unavailable to the test driver.

The SKIP flag can also be used for temporarily "commenting out" examples.

2.5 新版功能.

doctest.**COMPARISON_FLAGS**
A bitmask or'ing together all the comparison flags above.

The second group of options controls how test failures are reported:

doctest.**REPORT_UDIFF**
When specified, failures that involve multi-line expected and actual outputs are displayed using a unified diff.

doctest.**REPORT_CDIFF**
When specified, failures that involve multi-line expected and actual outputs will be displayed using a context diff.

doctest.**REPORT_NDIFF**
When specified, differences are computed by `difflib.Differ`, using the same algorithm as the popular `ndiff.py` utility. This is the only method that marks differences within lines as well as across lines. For example, if a line of expected output contains digit `1` where actual output contains letter `l`, a line is inserted with a caret marking the mismatching column positions.

doctest.**REPORT_ONLY_FIRST_FAILURE**
When specified, display the first failing example in each doctest, but suppress output for all remaining examples. This will prevent doctest from reporting correct examples that break because of earlier failures; but it might also hide incorrect examples that fail independently of the first failure. When *REPORT_ONLY_FIRST_FAILURE* is specified, the remaining examples are still run, and still count towards the total number of failures reported; only the output is suppressed.

doctest.**REPORTING_FLAGS**
A bitmask or'ing together all the reporting flags above.

2.4 新版功能: The constants *DONT_ACCEPT_BLANKLINE*, *NORMALIZE_WHITESPACE*, *ELLIPSIS*, *IGNORE_EXCEPTION_DETAIL*, *REPORT_UDIFF*, *REPORT_CDIFF*, *REPORT_NDIFF*, *REPORT_ONLY_FIRST_FAILURE*, *COMPARISON_FLAGS* and *REPORTING_FLAGS* were added.

There's also a way to register new option flag names, although this isn't useful unless you intend to extend *doctest* internals via subclassing:

doctest.**register_optionflag**(*name*)
Create a new option flag with a given name, and return the new flag's integer value. *register_optionflag()* can be used when subclassing *OutputChecker* or *DocTestRunner* to create new options that are supported by your subclasses. *register_optionflag()* should always be called using the following idiom:

```
MY_FLAG = register_optionflag('MY_FLAG')
```

2.4 新版功能.

Directives

Doctest directives may be used to modify the *option flags* for an individual example. Doctest directives are special Python comments following an example's source code:

```
directive             ::=   "#" "doctest:" directive_options
directive_options     ::=   directive_option ("," directive_option)\*
directive_option      ::=   on_or_off directive_option_name
on_or_off             ::=   "+" \| "-"
directive_option_name ::=   "DONT_ACCEPT_BLANKLINE" \| "NORMALIZE_WHITESPACE" \| ...
```

Whitespace is not allowed between the + or − and the directive option name. The directive option name can be any of the option flag names explained above.

An example's doctest directives modify doctest's behavior for that single example. Use + to enable the named behavior, or − to disable it.

For example, this test passes:

```
>>> print range(20)
[0,   1,  2,  3,  4,  5,  6,  7,  8,  9,
10,  11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Without the directive it would fail, both because the actual output doesn't have two blanks before the single-digit list elements, and because the actual output is on a single line. This test also passes, and also requires a directive to do so:

```
>>> print range(20)
[0, 1, ..., 18, 19]
```

Multiple directives can be used on a single physical line, separated by commas:

```
>>> print range(20)
[0,    1, ...,   18,    19]
```

If multiple directive comments are used for a single example, then they are combined:

```
>>> print range(20)
...
[0,    1, ...,   18,    19]
```

As the previous example shows, you can add `...` lines to your example containing only directives. This can be useful when an example is too long for a directive to comfortably fit on the same line:

```
>>> print range(5) + range(10,20) + range(30,40) + range(50,60)
...
[0, ..., 4, 10, ..., 19, 30, ..., 39, 50, ..., 59]
```

Note that since all options are disabled by default, and directives apply only to the example they appear in, enabling options (via + in a directive) is usually the only meaningful choice. However, option flags can also be passed to functions that run doctests, establishing different defaults. In such cases, disabling an option via − in a directive can be useful.

2.4 新版功能: Support for doctest directives was added.

警告

doctest is serious about requiring exact matches in expected output. If even a single character doesn't match, the test fails. This will probably surprise you a few times, as you learn exactly what Python does and doesn't guarantee about output. For example, when printing a dict, Python doesn't guarantee that the key-value pairs will be printed in any particular order, so a test like

```
>>> foo()
{"Hermione": "hippogryph", "Harry": "broomstick"}
```

is vulnerable! One workaround is to do

```
>>> foo() == {"Hermione": "hippogryph", "Harry": "broomstick"}
True
```

instead. Another is to do

```
>>> d = foo().items()
>>> d.sort()
>>> d
[('Harry', 'broomstick'), ('Hermione', 'hippogryph')]
```

There are others, but you get the idea.

Another bad idea is to print things that embed an object address, like

```
>>> id(1.0) # certain to fail some of the time
7948648
>>> class C: pass
>>> C()    # the default repr() for instances embeds an address
<__main__.C instance at 0x00AC18F0>
```

The *ELLIPSIS* directive gives a nice approach for the last example:

```
>>> C()
<__main__.C instance at 0x...>
```

Floating-point numbers are also subject to small output variations across platforms, because Python defers to the platform C library for float formatting, and C libraries vary widely in quality here.

```
>>> 1./7  # risky
0.14285714285714285
>>> print 1./7 # safer
0.142857142857
>>> print round(1./7, 6) # much safer
0.142857
```

Numbers of the form `I/2.**J` are safe across all platforms, and I often contrive doctest examples to produce numbers of that form:

```
>>> 3./4  # utterly safe
0.75
```

Simple fractions are also easier for people to understand, and that makes for better documentation.

25.2.4 Basic API

The functions `testmod()` and `testfile()` provide a simple interface to doctest that should be sufficient for most basic uses. For a less formal introduction to these two functions, see sections 简单用法：检查 *Docstrings* 中的示例 and *Simple Usage: Checking Examples in a Text File*.

doctest.**testfile**(*filename[, module_relative][, name][, package][, globs][, verbose][, report][, optionflags][, extraglobs][, raise_on_error][, parser][, encoding]*)

All arguments except *filename* are optional, and should be specified in keyword form.

Test examples in the file named *filename*. Return `(failure_count, test_count)`.

Optional argument *module_relative* specifies how the filename should be interpreted:

- If *module_relative* is `True` (the default), then *filename* specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the *package* argument is specified, then it is relative to that package. To ensure OS-independence, *filename* should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If *module_relative* is `False`, then *filename* specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument *name* gives the name of the test; by default, or if `None`, `os.path.basename(filename)` is used.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for a module-relative filename. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is `False`.

Optional argument *globs* gives a dict to be used as the globals when executing examples. A new shallow copy of this dict is created for the doctest, so its examples start with a clean slate. By default, or if `None`, a new empty dict is used.

Optional argument *extraglobs* gives a dict merged into the globals used to execute examples. This works like `dict.update()`: if *globs* and *extraglobs* have a common key, the associated value in *extraglobs* appears in the combined dict. By default, or if `None`, no extra globals are used. This is an advanced feature that allows parameterization of doctests. For example, a doctest can be written for a base class, using a generic name for the class, then reused to test any number of subclasses by passing an *extraglobs* dict mapping the generic name to the subclass to be tested.

Optional argument *verbose* prints lots of stuff if true, and prints only failures if false; by default, or if `None`, it's true if and only if `'-v'` is in `sys.argv`.

Optional argument *report* prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else the summary is very brief (in fact, empty if all tests passed).

Optional argument *optionflags* or's together option flags. See section *Option Flags*.

Optional argument *raise_on_error* defaults to false. If true, an exception is raised upon the first failure or unexpected exception in an example. This allows failures to be post-mortem debugged. Default behavior is to continue running examples.

Optional argument *parser* specifies a `DocTestParser` (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

2.4 新版功能.

在 2.5 版更改: The parameter *encoding* was added.

doctest.**testmod**(*[m][, name][, globs][, verbose][, report][, optionflags][, extraglobs][, raise_on_error][, exclude_empty]*)

All arguments are optional, and all except for *m* should be specified in keyword form.

Test examples in docstrings in functions and classes reachable from module *m* (or module *__main__* if *m* is not supplied or is None), starting with m.__doc__.

Also test examples reachable from dict m.__test__, if it exists and is not None. m.__test__ maps names (strings) to functions, classes and strings; function and class docstrings are searched for examples; strings are searched directly, as if they were docstrings.

Only docstrings attached to objects belonging to module *m* are searched.

Return (failure_count, test_count).

Optional argument *name* gives the name of the module; by default, or if None, m.__name__ is used.

Optional argument *exclude_empty* defaults to false. If true, objects for which no doctests are found are excluded from consideration. The default is a backward compatibility hack, so that code still using doctest.master.summarize() in conjunction with *testmod()* continues to get output for objects with no tests. The *exclude_empty* argument to the newer *DocTestFinder* constructor defaults to true.

Optional arguments *extraglobs*, *verbose*, *report*, *optionflags*, *raise_on_error*, and *globs* are the same as for function *testfile()* above, except that *globs* defaults to m.__dict__.

在 2.3 版更改: The parameter *optionflags* was added.

在 2.4 版更改: The parameters *extraglobs*, *raise_on_error* and *exclude_empty* were added.

在 2.5 版更改: The optional argument *isprivate*, deprecated in 2.4, was removed.

doctest.**run_docstring_examples**(*f, globs[, verbose][, name][, compileflags][, optionflags]*)

Test examples associated with object *f*; for example, *f* may be a string, a module, a function, or a class object.

A shallow copy of dictionary argument *globs* is used for the execution context.

Optional argument *name* is used in failure messages, and defaults to "NoName".

If optional argument *verbose* is true, output is generated even if there are no failures. By default, output is generated only in case of an example failure.

Optional argument *compileflags* gives the set of flags that should be used by the Python compiler when running the examples. By default, or if None, flags are deduced corresponding to the set of future features found in *globs*.

Optional argument *optionflags* works as for function *testfile()* above.

25.2.5 Unittest API

As your collection of doctest'ed modules grows, you'll want a way to run all their doctests systematically. Prior to Python 2.4, *doctest* had a barely documented Tester class that supplied a rudimentary way to combine doctests from multiple modules. Tester was feeble, and in practice most serious Python testing frameworks build on the *unittest* module, which supplies many flexible ways to combine tests from multiple sources. So, in Python 2.4, *doctest*'s Tester class is deprecated, and *doctest* provides two functions that can be used to create *unittest* test suites from modules and text files containing doctests. To integrate with *unittest* test discovery, include a load_tests() function in your test module:

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
```

(下页继续)

(续上页)

```
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

There are two main functions for creating *unittest.TestSuite* instances from text files and modules with doctests:

doctest.**DocFileSuite**(**paths, [module_relative][, package][, setUp][, tearDown][, globs][, optionflags][, parser][, encoding]*)

Convert doctest tests from one or more text files to a *unittest.TestSuite*.

The returned *unittest.TestSuite* is to be run by the unittest framework and runs the interactive examples in each file. If an example in any file fails, then the synthesized unit test fails, and a `failureException` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Pass one or more paths (as strings) to text files to be examined.

Options may be provided as keyword arguments:

Optional argument *module_relative* specifies how the filenames in *paths* should be interpreted:

- If *module_relative* is `True` (the default), then each filename in *paths* specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the *package* argument is specified, then it is relative to that package. To ensure OS-independence, each filename should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If *module_relative* is `False`, then each filename in *paths* specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for module-relative filenames in *paths*. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is `False`.

Optional argument *setUp* specifies a set-up function for the test suite. This is called before running the tests in each file. The *setUp* function will be passed a *DocTest* object. The setUp function can access the test globals as the *globs* attribute of the test passed.

Optional argument *tearDown* specifies a tear-down function for the test suite. This is called after running the tests in each file. The *tearDown* function will be passed a *DocTest* object. The setUp function can access the test globals as the *globs* attribute of the test passed.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is a new empty dictionary.

Optional argument *optionflags* specifies the default doctest options for the tests, created by or-ing together individual option flags. See section *Option Flags*. See function *set_unittest_reportflags()* below for a better way to set reporting options.

Optional argument *parser* specifies a *DocTestParser* (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

2.4 新版功能.

在 2.5 版更改: The global `__file__` was added to the globals provided to doctests loaded from a text file using *DocFileSuite()*.

在 2.5 版更改: The parameter *encoding* was added.

注解： Unlike *testmod()* and *DocTestFinder*, this function raises a *ValueError* if *module* contains no docstrings. You can prevent this error by passing a *DocTestFinder* instance as the *test_finder* argument with its *exclude_empty* keyword argument set to `False`:

```
>>> finder = doctest.DocTestFinder(exclude_empty=False)
>>> suite = doctest.DocTestSuite(test_finder=finder)
```

doctest.**DocTestSuite**(*[module][, globs][, extraglobs][, test_finder][, setUp][, tearDown][, checker]*)

Convert doctest tests for a module to a *unittest.TestSuite*.

The returned *unittest.TestSuite* is to be run by the unittest framework and runs each doctest in the module. If any of the doctests fail, then the synthesized unit test fails, and a `failureException` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Optional argument *module* provides the module to be tested. It can be a module object or a (possibly dotted) module name. If not specified, the module calling this function is used.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is a new empty dictionary.

Optional argument *extraglobs* specifies an extra set of global variables, which is merged into *globs*. By default, no extra globals are used.

Optional argument *test_finder* is the *DocTestFinder* object (or a drop-in replacement) that is used to extract doctests from the module.

Optional arguments *setUp*, *tearDown*, and *optionflags* are the same as for function *DocFileSuite()* above.

2.3 新版功能.

在 2.4 版更改: The parameters *globs*, *extraglobs*, *test_finder*, *setUp*, *tearDown*, and *optionflags* were added; this function now uses the same search technique as *testmod()*.

Under the covers, *DocTestSuite()* creates a *unittest.TestSuite* out of `doctest.DocTestCase` instances, and `DocTestCase` is a subclass of *unittest.TestCase*. `DocTestCase` isn't documented here (it's an internal detail), but studying its code can answer questions about the exact details of *unittest* integration.

Similarly, *DocFileSuite()* creates a *unittest.TestSuite* out of `doctest.DocFileCase` instances, and `DocFileCase` is a subclass of `DocTestCase`.

So both ways of creating a *unittest.TestSuite* run instances of `DocTestCase`. This is important for a subtle reason: when you run *doctest* functions yourself, you can control the *doctest* options in use directly, by passing option flags to *doctest* functions. However, if you're writing a *unittest* framework, *unittest* ultimately controls when and how tests get run. The framework author typically wants to control *doctest* reporting options (perhaps, e.g., specified by command line options), but there's no way to pass options through *unittest* to *doctest* test runners.

For this reason, *doctest* also supports a notion of *doctest* reporting flags specific to *unittest* support, via this function:

doctest.**set_unittest_reportflags**(*flags*)

Set the *doctest* reporting flags to use.

Argument *flags* or's together option flags. See section *Option Flags*. Only "reporting flags" can be used.

This is a module-global setting, and affects all future doctests run by module *unittest*: the `runTest()` method of `DocTestCase` looks at the option flags specified for the test case when the `DocTestCase` instance was constructed. If no reporting flags were specified (which is the typical and expected case), *doctest*'s *unittest* reporting flags are bitwise ORed into the option flags, and the option flags so augmented are passed

to the *DocTestRunner* instance created to run the doctest. If any reporting flags were specified when the DocTestCase instance was constructed, *doctest*'s *unittest* reporting flags are ignored.

The value of the *unittest* reporting flags in effect before the function was called is returned by the function.

2.4 新版功能.

25.2.6 Advanced API

The basic API is a simple wrapper that's intended to make doctest easy to use. It is fairly flexible, and should meet most users' needs; however, if you require more fine-grained control over testing, or wish to extend doctest's capabilities, then you should use the advanced API.

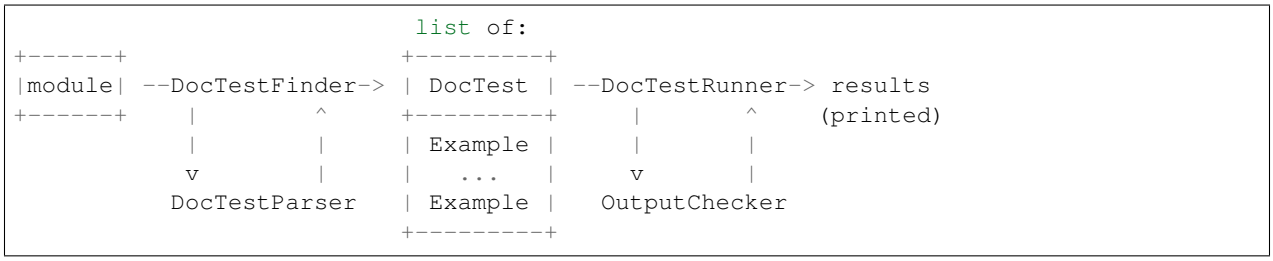
The advanced API revolves around two container classes, which are used to store the interactive examples extracted from doctest cases:

- *Example*: A single Python *statement*, paired with its expected output.
- *DocTest*: A collection of *Example*s, typically extracted from a single docstring or text file.

Additional processing classes are defined to find, parse, and run, and check doctest examples:

- *DocTestFinder*: Finds all docstrings in a given module, and uses a *DocTestParser* to create a *DocTest* from every docstring that contains interactive examples.
- *DocTestParser*: Creates a *DocTest* object from a string (such as an object's docstring).
- *DocTestRunner*: Executes the examples in a *DocTest*, and uses an *OutputChecker* to verify their output.
- *OutputChecker*: Compares the actual output from a doctest example with the expected output, and decides whether they match.

The relationships among these processing classes are summarized in the following diagram:



DocTest 对象

class doctest.**DocTest**(*examples*, *globs*, *name*, *filename*, *lineno*, *docstring*)

A collection of doctest examples that should be run in a single namespace. The constructor arguments are used to initialize the attributes of the same names.

2.4 新版功能.

DocTest defines the following attributes. They are initialized by the constructor, and should not be modified directly.

examples

A list of *Example* objects encoding the individual interactive Python examples that should be run by this test.

globs
The namespace (aka globals) that the examples should be run in. This is a dictionary mapping names to values. Any changes to the namespace made by the examples (such as binding new variables) will be reflected in *globs* after the test is run.

name
A string name identifying the *DocTest*. Typically, this is the name of the object or file that the test was extracted from.

filename
The name of the file that this *DocTest* was extracted from; or None if the filename is unknown, or if the *DocTest* was not extracted from a file.

lineno
The line number within *filename* where this *DocTest* begins, or None if the line number is unavailable. This line number is zero-based with respect to the beginning of the file.

docstring
The string that the test was extracted from, or None if the string is unavailable, or if the test was not extracted from a string.

Example Objects

class doctest.**Example**(*source, want[, exc_msg][, lineno][, indent][, options]*)
A single interactive example, consisting of a Python statement and its expected output. The constructor arguments are used to initialize the attributes of the same names.

2.4 新版功能.

Example defines the following attributes. They are initialized by the constructor, and should not be modified directly.

source
A string containing the example's source code. This source code consists of a single Python statement, and always ends with a newline; the constructor adds a newline when necessary.

want
The expected output from running the example's source code (either from stdout, or a traceback in case of exception). *want* ends with a newline unless no output is expected, in which case it's an empty string. The constructor adds a newline when necessary.

exc_msg
The exception message generated by the example, if the example is expected to generate an exception; or None if it is not expected to generate an exception. This exception message is compared against the return value of *traceback.format_exception_only()*. *exc_msg* ends with a newline unless it's None. The constructor adds a newline if needed.

lineno
The line number within the string containing this example where the example begins. This line number is zero-based with respect to the beginning of the containing string.

indent
The example's indentation in the containing string, i.e., the number of space characters that precede the example's first prompt.

options
A dictionary mapping from option flags to True or False, which is used to override default options for this example. Any option flags not contained in this dictionary are left at their default value (as specified by the *DocTestRunner*'s optionflags). By default, no options are set.

DocTestFinder 对象

class doctest.**DocTestFinder**(*[verbose][, parser][, recurse][, exclude_empty]*)

A processing class used to extract the *DocTest*s that are relevant to a given object, from its docstring and the docstrings of its contained objects. *DocTest*s can currently be extracted from the following object types: modules, functions, classes, methods, staticmethods, classmethods, and properties.

The optional argument *verbose* can be used to display the objects searched by the finder. It defaults to `False` (no output).

The optional argument *parser* specifies the *DocTestParser* object (or a drop-in replacement) that is used to extract doctests from docstrings.

If the optional argument *recurse* is false, then *DocTestFinder.find()* will only examine the given object, and not any contained objects.

If the optional argument *exclude_empty* is false, then *DocTestFinder.find()* will include tests for objects with empty docstrings.

2.4 新版功能.

DocTestFinder defines the following method:

find(*obj[, name][, module][, globs][, extraglobs]*)

Return a list of the *DocTest*s that are defined by *obj*'s docstring, or by any of its contained objects' docstrings.

The optional argument *name* specifies the object's name; this name will be used to construct names for the returned *DocTest*s. If *name* is not specified, then `obj.__name__` is used.

The optional parameter *module* is the module that contains the given object. If the module is not specified or is `None`, then the test finder will attempt to automatically determine the correct module. The object's module is used:

- As a default namespace, if *globs* is not specified.
- To prevent the DocTestFinder from extracting DocTests from objects that are imported from other modules. (Contained objects with modules other than *module* are ignored.)
- To find the name of the file containing the object.
- To help find the line number of the object within its file.

If *module* is `False`, no attempt to find the module will be made. This is obscure, of use mostly in testing doctest itself: if *module* is `False`, or is `None` but cannot be found automatically, then all objects are considered to belong to the (non-existent) module, so all contained objects will (recursively) be searched for doctests.

The globals for each *DocTest* is formed by combining *globs* and *extraglobs* (bindings in *extraglobs* override bindings in *globs*). A new shallow copy of the globals dictionary is created for each *DocTest*. If *globs* is not specified, then it defaults to the module's *__dict__*, if specified, or `{}` otherwise. If *extraglobs* is not specified, then it defaults to `{}`.

DocTestParser 对象

class doctest.**DocTestParser**
A processing class used to extract interactive examples from a string, and use them to create a *DocTest* object.

2.4 新版功能.

DocTestParser defines the following methods:

get_doctest(*string*, *globs*, *name*, *filename*, *lineno*)
Extract all doctest examples from the given string, and collect them into a *DocTest* object.

globs, *name*, *filename*, and *lineno* are attributes for the new *DocTest* object. See the documentation for *DocTest* for more information.

get_examples(*string*[, *name*])
Extract all doctest examples from the given string, and return them as a list of *Example* objects. Line numbers are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

parse(*string*[, *name*])
Divide the given string into examples and intervening text, and return them as a list of alternating *Example*s and strings. Line numbers for the *Example*s are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

DocTestRunner 对象

class doctest.**DocTestRunner**(*[checker][, verbose][, optionflags]*)
A processing class used to execute and verify the interactive examples in a *DocTest*.

The comparison between expected outputs and actual outputs is done by an *OutputChecker*. This comparison may be customized with a number of option flags; see section *Option Flags* for more information. If the option flags are insufficient, then the comparison may also be customized by passing a subclass of *OutputChecker* to the constructor.

The test runner's display output can be controlled in two ways. First, an output function can be passed to `TestRunner.run()`; this function will be called with strings that should be displayed. It defaults to `sys.stdout.write`. If capturing the output is not sufficient, then the display output can be also customized by subclassing DocTestRunner, and overriding the methods *report_start()*, *report_success()*, *report_unexpected_exception()*, and *report_failure()*.

The optional keyword argument *checker* specifies the *OutputChecker* object (or drop-in replacement) that should be used to compare the expected outputs to the actual outputs of doctest examples.

The optional keyword argument *verbose* controls the *DocTestRunner*'s verbosity. If *verbose* is `True`, then information is printed about each example, as it is run. If *verbose* is `False`, then only failures are printed. If *verbose* is unspecified, or `None`, then verbose output is used iff the command-line switch `-v` is used.

The optional keyword argument *optionflags* can be used to control how the test runner compares expected output to actual output, and how it displays failures. For more information, see section *Option Flags*.

2.4 新版功能.

DocTestParser defines the following methods:

report_start(*out*, *test*, *example*)
Report that the test runner is about to process the given example. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *test* is the test *containing example*. *out* is the output function that was passed to *DocTestRunner.run()*.

report_success(*out*, *test*, *example*, *got*)
Report that the given example ran successfully. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to *DocTestRunner.run()*.

report_failure(*out*, *test*, *example*, *got*)
Report that the given example failed. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to *DocTestRunner.run()*.

report_unexpected_exception(*out*, *test*, *example*, *exc_info*)
Report that the given example raised an unexpected exception. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *exc_info* is a tuple containing information about the unexpected exception (as returned by *sys.exc_info()*). *test* is the test containing *example*. *out* is the output function that was passed to *DocTestRunner.run()*.

run(*test[, compileflags][, out][, clear_globs]*)
Run the examples in *test* (a *DocTest* object), and display the results using the writer function *out*.

The examples are run in the namespace `test.globs`. If *clear_globs* is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use *clear_globs=False*.

compileflags gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to *globs*.

The output of each example is checked using the *DocTestRunner*'s output checker, and the results are formatted by the `DocTestRunner.report_*()` methods.

summarize([*verbose*])
Print a summary of all the test cases that have been run by this DocTestRunner, and return a *named tuple* `TestResults(failed, attempted)`.

The optional *verbose* argument controls how detailed the summary is. If the verbosity is not specified, then the *DocTestRunner*'s verbosity is used.

在 2.6 版更改: Use a named tuple.

OutputChecker 对象

class doctest.**OutputChecker**
A class used to check the whether the actual output from a doctest example matches the expected output. *OutputChecker* defines two methods: *check_output()*, which compares a given pair of outputs, and returns true if they match; and *output_difference()*, which returns a string describing the differences between two outputs.

2.4 新版功能.

OutputChecker defines the following methods:

check_output(*want*, *got*, *optionflags*)
Return `True` iff the actual output from an example (*got*) matches the expected output (*want*). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See section *Option Flags* for more information about option flags.

output_difference(*example*, *got*, *optionflags*)
Return a string describing the differences between the expected output for a given example (*example*) and the actual output (*got*). *optionflags* is the set of option flags used to compare *want* and *got*.

25.2.7 调试

Doctest provides several mechanisms for debugging doctest examples:

- Several functions convert doctests to executable Python programs, which can be run under the Python debugger, *pdb*.
- The *DebugRunner* class is a subclass of *DocTestRunner* that raises an exception for the first failing example, containing information about that example. This information can be used to perform post-mortem debugging on the example.
- The *unittest* cases generated by *DocTestSuite()* support the *debug()* method defined by *unittest.TestCase*.
- You can add a call to *pdb.set_trace()* in a doctest example, and you'll drop into the Python debugger when that line is executed. Then you can inspect current values of variables, and so on. For example, suppose `a.py` contains just this module docstring:

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print x+3
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

Then an interactive Python session may look like this:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1     def g(x):
  2         print x+3
  3  ->     import pdb; pdb.set_trace()
[EOF]
(Pdb) print x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
  1     def f(x):
  2  ->     g(x*2)
[EOF]
(Pdb) print x
3
(Pdb) step
```

(下页继续)

(续上页)

```
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

在 2.4 版更改: The ability to use *pdb.set_trace()* usefully inside doctests was added.

Functions that convert doctests to Python code, and possibly run the synthesized code under the debugger:

doctest.**script_from_examples**(*s*)

Convert text with examples to a script.

Argument *s* is a string containing doctest examples. The string is converted to a Python script, where doctest examples in *s* are converted to regular code, and everything else is converted to Python comments. The generated script is returned as a string. For example,

```
import doctest
print doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print x+y
    3
""")
```

displays:

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print x+y
# Expected:
## 3
```

This function is used internally by other functions (see below), but can also be useful when you want to transform an interactive Python session into a Python script.

2.4 新版功能.

doctest.**testsource**(*module*, *name*)

Convert the doctest for an object to a script.

Argument *module* is a module object, or dotted name of a module, containing the object whose doctests are of interest. Argument *name* is the name (within the module) of the object with the doctests of interest. The result is a string, containing the object's docstring converted to a Python script, as described for *script_from_examples()* above. For example, if module `a.py` contains a top-level function `f()`, then

```
import a, doctest
print doctest.testsource(a, "a.f")
```

prints a script version of function `f()`'s docstring, with doctests converted to code, and the rest placed in comments.

2.3 新版功能.

doctest.**debug**(*module*, *name*[, *pm*])
Debug the doctests for an object.

The *module* and *name* arguments are the same as for function *testsource()* above. The synthesized Python script for the named object's docstring is written to a temporary file, and then that file is run under the control of the Python debugger, *pdb*.

A shallow copy of `module.__dict__` is used for both local and global execution context.

Optional argument *pm* controls whether post-mortem debugging is used. If *pm* has a true value, the script file is run directly, and the debugger gets involved only if the script terminates via raising an unhandled exception. If it does, then post-mortem debugging is invoked, via *pdb.post_mortem()*, passing the traceback object from the unhandled exception. If *pm* is not specified, or is false, the script is run under the debugger from the start, via passing an appropriate *execfile()* call to *pdb.run()*.

2.3 新版功能.

在 2.4 版更改: The *pm* argument was added.

doctest.**debug_src**(*src[, pm][, globs]*)
Debug the doctests in a string.

This is like function *debug()* above, except that a string containing doctest examples is specified directly, via the *src* argument.

Optional argument *pm* has the same meaning as in function *debug()* above.

Optional argument *globs* gives a dictionary to use as both local and global execution context. If not specified, or `None`, an empty dictionary is used. If specified, a shallow copy of the dictionary is used.

2.4 新版功能.

The *DebugRunner* class, and the special exceptions it may raise, are of most interest to testing framework authors, and will only be sketched here. See the source code, and especially *DebugRunner*'s docstring (which is a doctest!) for more details:

class doctest.**DebugRunner**(*[checker][, verbose][, optionflags]*)
A subclass of *DocTestRunner* that raises an exception as soon as a failure is encountered. If an unexpected exception occurs, an *UnexpectedException* exception is raised, containing the test, the example, and the original exception. If the output doesn't match, then a *DocTestFailure* exception is raised, containing the test, the example, and the actual output.

For information about the constructor parameters and methods, see the documentation for *DocTestRunner* in section *Advanced API*.

There are two exceptions that may be raised by *DebugRunner* instances:

exception doctest.**DocTestFailure**(*test*, *example*, *got*)
An exception raised by *DocTestRunner* to signal that a doctest example's actual output did not match its expected output. The constructor arguments are used to initialize the attributes of the same names.

DocTestFailure defines the following attributes:

DocTestFailure.**test**
The *DocTest* object that was being run when the example failed.

DocTestFailure.**example**
The *Example* that failed.

DocTestFailure.**got**
The example's actual output.

exception doctest.**UnexpectedException**(*test*, *example*, *exc_info*)
An exception raised by *DocTestRunner* to signal that a doctest example raised an unexpected exception. The constructor arguments are used to initialize the attributes of the same names.

UnexpectedException defines the following attributes:

UnexpectedException.**test**
The *DocTest* object that was being run when the example failed.

UnexpectedException.**example**
The *Example* that failed.

UnexpectedException.**exc_info**
A tuple containing information about the unexpected exception, as returned by *sys.exc_info()*.

25.2.8 Soapbox

As mentioned in the introduction, *doctest* has grown to have three primary uses:

1. Checking examples in docstrings.
2. Regression testing.
3. Executable documentation / literate testing.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.

When writing a docstring, choose docstring examples with care. There's an art to this that needs to be learned—it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and things change. I'm still amazed at how often one of my *doctest* examples stops working after a "harmless" change.

Doctest also makes an excellent tool for regression testing, especially if you don't skimp on explanatory text. By interleaving prose and examples, it becomes much easier to keep track of what's actually being tested, and why. When a test fails, good prose can make it much easier to figure out what the problem is, and how it should be fixed. It's true that you could write extensive comments in code-based testing, but few programmers do. Many have found that using doctest approaches instead leads to much clearer tests. Perhaps this is simply because doctest makes writing prose a little easier than writing code, while writing comments in code is a little harder. I think it goes deeper than just that: the natural attitude when writing a doctest-based test is that you want to explain the fine points of your software, and illustrate them with examples. This in turn naturally leads to test files that start with the simplest features, and logically progress to complications and edge cases. A coherent narrative is the result, instead of a collection of isolated functions that test isolated bits of functionality seemingly at random. It's a different attitude, and produces different results, blurring the distinction between testing and explaining.

Regression testing is best confined to dedicated objects or files. There are several options for organizing tests:

- Write text files containing test cases as interactive examples, and test the files using *testfile()* or *DocFileSuite()*. This is recommended, although is easiest to do for new projects, designed from the start to use doctest.
- Define functions named `_regrtest_topic` that consist of single docstrings, containing test cases for the named topics. These functions can be included in the same file as the module, or separated out into a separate test file.
- Define a `__test__` dictionary mapping from regression test topics to docstrings containing test cases.

When you have placed your tests in a module, the module can itself be the test runner. When a test fails, you can arrange for your test runner to re-run only the failing doctest while you debug the problem. Here is a minimal example of such a test runner:

```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.REPORT_ONLY_FIRST_FAILURE
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), name=name,
                                       optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print("{} failures out of {} tests".format(fail, total))
```

备注

25.3 unittest —单元测试框架

2.1 新版功能.

(如果你已经对测试的概念比较熟悉了，你可能想直接跳转到这一部分*断言方法*。)

The Python unit testing framework, sometimes referred to as "PyUnit," is a Python language version of JUnit, by Kent Beck and Erich Gamma. JUnit is, in turn, a Java version of Kent's Smalltalk testing framework. Each is the de facto standard unit testing framework for its respective language.

unittest supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. The *unittest* module provides classes that make it easy to support these qualities for a set of tests.

To achieve this, *unittest* supports some important concepts:

测试脚手架 *test fixture* 表示为了开展一项或多项测试所需要进行的准备工作，以及所有相关的清理操作。举个例子，这可能包含创建临时或代理的数据库、目录，再或者启动一个服务器进程。

测试用例 A *test case* is the smallest unit of testing. It checks for a specific response to a particular set of inputs. *unittest* provides a base class, *TestCase*, which may be used to create new test cases.

测试套件 *test suite* 是一系列的测试用例，或测试套件，或两者皆有。它用于归档需要一起执行的测试。

测试运行器（test runner） *test runner* 是一个用于执行和输出测试结果的组件。这个运行器可能使用图形接口、文本接口，或返回一个特定的值表示运行测试的结果。

The test case and test fixture concepts are supported through the *TestCase* and *FunctionTestCase* classes; the former should be used when creating new tests, and the latter can be used when integrating existing test code with a *unittest*-driven framework. When building test fixtures using *TestCase*, the *setUp()* and *tearDown()* methods can be overridden to provide initialization and cleanup for the fixture. With *FunctionTestCase*, existing functions can be passed to the constructor for these purposes. When the test is run, the fixture initialization is run first; if it succeeds, the cleanup method is run after the test has been executed, regardless of the outcome of the test. Each instance of the *TestCase* will only be used to run a single test method, so a new fixture is created for each test.

Test suites are implemented by the *TestSuite* class. This class allows individual tests and test suites to be aggregated; when the suite is executed, all tests added directly to the suite and in "child" test suites are run.

A test runner is an object that provides a single method, `run()`, which accepts a *TestCase* or *TestSuite* object as a parameter, and returns a result object. The class *TestResult* is provided for use as the result object. *unittest* provides the *TextTestRunner* as an example test runner which reports test results on the standard error stream by

default. Alternate runners can be implemented for other environments (such as graphical environments) without any need to derive from a specific class.

参见:

doctest —**文档测试模块** 另一个风格完全不同的测试模块。

unittest2: A backport of new unittest features for Python 2.4-2.6 Many new features were added to unittest in Python 2.7, including test discovery. unittest2 allows you to use these features with earlier versions of Python.

Simple Smalltalk Testing: With Patterns Kent Beck's original paper on testing frameworks using the pattern shared by *unittest*.

Nose and pytest Third-party unittest frameworks with a lighter-weight syntax for writing tests. For example, `assert func(10) == 42`.

The Python Testing Tools Taxonomy An extensive list of Python testing tools including functional testing frameworks and mock object libraries.

Testing in Python Mailing List A special-interest-group for discussion of testing, and testing tools, in Python.

25.3.1 基本实例

unittest 模块提供了一系列创建和运行测试的工具。这一段落演示了这些工具的一小部分，但也足以满足大部分用户的需求。

这是一段简短的代码，来测试三种字符串方法:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

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

继承*unittest.TestCase* 就创建了一个测试样例。上述三个独立的测试是三个类的方法，这些方法的命名都以 `test` 开头。这个命名约定告诉测试运行者类的哪些方法表示测试。

每个测试的关键是：调用*assertEqual()* 来检查预期的输出；调用*assertTrue()* 或*assertFalse()* 来验证一个条件；调用*assertRaises()* 来验证抛出了一个特定的异常。使用这些方法而不是 `assert` 语句是为了让测试运行者能聚合所有的测试结果并产生结果报告。

通过*setUp()* 和*tearDown()* 方法，可以设置测试开始前与完成后需要执行的指令。在*组织你的测试代码* 中，对此有更为详细的描述。

最后的代码块中，演示了运行测试的一个简单的方法。*unittest.main()* 提供了一个测试脚本的命令行接口。当在命令行运行该测试脚本，上文的脚本生成如以下格式的输出:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Instead of *unittest.main()*, there are other ways to run the tests with a finer level of control, less terse output, and no requirement to be run from the command line. For example, the last two lines may be replaced with:

```
suite = unittest.TestLoader().loadTestsFromTestCase(TestStringMethods)
unittest.TextTestRunner(verbosity=2).run(suite)
```

Running the revised script from the interpreter or another script produces the following output:

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

以上例子演示了*unittest* 中最常用的、足够满足许多日常测试需求的特性。文档的剩余部分详述该框架的完整特性。

25.3.2 命令行界面

unittest 模块可以通过命令行运行模块、类和独立测试方法的测试:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

你可以传入模块名、类或方法名或他们的任意组合。

在运行测试时，你可以通过添加 -v 参数获取更详细（更多的冗余）的信息。

```
python -m unittest -v test_module
```

用于获取命令行选项列表:

```
python -m unittest -h
```

在 2.7 版更改: 在早期版本中，只支持运行独立的测试方法，而不支持模块和类。

命令行选项

unittest supports these command-line options:

-b, --buffer
在测试运行时，标准输出流与标准错误流会被放入缓冲区。成功的测试的运行时输出会被丢弃；测试不通过时，测试运行中的输出会正常显示，错误会被加入到测试失败信息。

-c, --catch
当测试正在运行时，Control-C 会等待当前测试完成，并在完成后报告已执行的测试的结果。当再次按下 Control-C 时，引发平常的*KeyboardInterrupt* 异常。

See *Signal Handling* for the functions that provide this functionality.

-f, --failfast
当出现第一个错误或者失败时，停止运行测试。

2.7 新版功能: 添加命令行选项 -b, -c 和 -f 。

命令行亦可用于探索性测试，以运行一个项目的所有测试或其子集。

25.3.3 探索性测试

2.7 新版功能.

Unittest supports simple test discovery. In order to be compatible with test discovery, all of the test files must be modules or packages importable from the top-level directory of the project (this means that their filenames must be valid identifiers).

探索性测试在*TestLoader.discover()* 中实现，但也可以通过命令行使用。它在命令行中的基本用法如下：

```
cd project_directory
python -m unittest discover
```

discover 有以下选项：

-v, --verbose
更详细地输出结果。

-s, --start-directory directory
开始进行搜索的目录 (默认值为当前目录 .)。

-p, --pattern pattern
用于匹配测试文件的模式（默认为 test*.py)。

-t, --top-level-directory directory
指定项目的最上层目录（通常为开始时所在目录)。

-s ， *-p* 和*-t* 选项可以按顺序作为位置参数传入。以下两条命令是等价的：

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

正如可以传入路径那样，传入一个包名作为起始目录也是可行的，如 myproject.subpackage.test 。你提供的包名会被导入，它在文件系统中的位置会被作为起始目录。

警告: 探索性测试通过导入测试对测试进行加载。在找到所有你指定的开始目录下的所有测试文件后，它把路径转换为包名并进行导入。如 foo/bar/baz.py 会被导入为 foo.bar.baz 。

如果你有一个全局安装的包，并尝试对这个包的副本进行探索性测试，可能会从错误的地方开始导入。如果出现这种情况，测试会输出警告并退出。

如果你使用包名而不是路径作为开始目录，搜索时会假定它导入的是你想要的目录，所以你不会收到警告。

测试模块和包可以通过*load_tests protocol* 自定义测试的加载和搜索。

25.3.4 组织你的测试代码

The basic building blocks of unit testing are *test cases* —single scenarios that must be set up and checked for correctness. In *unittest*, test cases are represented by instances of *unittest*' s *TestCase* class. To make your own test cases you must write subclasses of *TestCase*, or use *FunctionTestCase*.

An instance of a *TestCase*-derived class is an object that can completely run a single test method, together with optional set-up and tidy-up code.

一个*TestCase* 实例的测试代码必须是完全自含的，因此它可以独立运行，或与其它任意组合任意数量的测试用例一起运行。

The simplest *TestCase* subclass will simply override the `runTest()` method in order to perform specific testing code:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def runTest(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50), 'incorrect default size')
```

Note that in order to test something, we use one of the `assert*()` methods provided by the *TestCase* base class. If the test fails, an exception will be raised, and *unittest* will identify the test case as a *failure*. Any other exceptions will be treated as *errors*. This helps you identify where the problem is: *failures* are caused by incorrect results - a 5 where you expected a 6. *Errors* are caused by incorrect code - e.g., a *TypeError* caused by an incorrect function call.

The way to run a test case will be described later. For now, note that to construct an instance of such a test case, we call its constructor without arguments:

```
testCase = DefaultWidgetSizeTestCase()
```

Now, such test cases can be numerous, and their set-up can be repetitive. In the above case, constructing a `Widget` in each of 100 Widget test case subclasses would mean unsightly duplication.

Luckily, we can factor out such set-up code by implementing a method called *setUp()*, which the testing framework will automatically call for us when we run the test:

```
import unittest

class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

class DefaultWidgetSizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')
```

(下页继续)

(续上页)

```
class WidgetResizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

If the *setUp()* method raises an exception while the test is running, the framework will consider the test to have suffered an error, and the `runTest()` method will not be executed.

Similarly, we can provide a *tearDown()* method that tidies up after the `runTest()` method has been run:

```
import unittest

class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
        self.widget = None
```

If *setUp()* succeeded, the *tearDown()* method will be run whether `runTest()` succeeded or not.

Such a working environment for the testing code is called a *fixture*.

Often, many small test cases will use the same fixture. In this case, we would end up subclassing `SimpleWidgetTestCase` into many small one-method classes such as `DefaultWidgetSizeTestCase`. This is time-consuming and discouraging, so in the same vein as JUnit, *unittest* provides a simpler mechanism:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
        self.widget = None

    def test_default_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def test_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

Here we have not provided a `runTest()` method, but have instead provided two different test methods. Class instances will now each run one of the `test_*()` methods, with `self.widget` created and destroyed separately for each instance. When creating an instance we must specify the test method it is to run. We do this by passing the method name in the constructor:

```
defaultSizeTestCase = WidgetTestCase('test_default_size')
resizeTestCase = WidgetTestCase('test_resize')
```

Test case instances are grouped together according to the features they test. *unittest* provides a mechanism for this: the *test suite*, represented by *unittest*'s *TestSuite* class:

```
widgetTestSuite = unittest.TestSuite()
widgetTestSuite.addTest(WidgetTestCase('test_default_size'))
widgetTestSuite.addTest(WidgetTestCase('test_resize'))
```

For the ease of running tests, as we will see later, it is a good idea to provide in each test module a callable object that returns a pre-built test suite:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_size'))
    suite.addTest(WidgetTestCase('test_resize'))
    return suite
```

or even:

```
def suite():
    tests = ['test_default_size', 'test_resize']

    return unittest.TestSuite(map(WidgetTestCase, tests))
```

Since it is a common pattern to create a *TestCase* subclass with many similarly named test functions, *unittest* provides a *TestLoader* class that can be used to automate the process of creating a test suite and populating it with individual tests. For example,

```
suite = unittest.TestLoader().loadTestsFromTestCase(WidgetTestCase)
```

will create a test suite that will run `WidgetTestCase.test_default_size()` and `WidgetTestCase.test_resize`. *TestLoader* uses the `'test'` method name prefix to identify test methods automatically.

Note that the order in which the various test cases will be run is determined by sorting the test function names with respect to the built-in ordering for strings.

Often it is desirable to group suites of test cases together, so as to run tests for the whole system at once. This is easy, since *TestSuite* instances can be added to a *TestSuite* just as *TestCase* instances can be added to a *TestSuite*:

```
suite1 = module1.TheTestSuite()
suite2 = module2.TheTestSuite()
alltests = unittest.TestSuite([suite1, suite2])
```

You can place the definitions of test cases and test suites in the same modules as the code they are to test (such as `widget.py`), but there are several advantages to placing the test code in a separate module, such as `test_widget.py`:

- The test module can be run standalone from the command line.
- The test code can more easily be separated from shipped code.
- There is less temptation to change test code to fit the code it tests without a good reason.
- Test code should be modified much less frequently than the code it tests.
- Tested code can be refactored more easily.
- Tests for modules written in C must be in separate modules anyway, so why not be consistent?
- If the testing strategy changes, there is no need to change the source code.

25.3.5 复用已有的测试代码

一些用户希望直接使用*unittest* 运行已有的测试代码，而不需要把已有的每个测试函数转化为一个*TestCase* 的子类。

因此，*unittest* 提供*FunctionTestCase* 类。这个*TestCase* 的子类可用于打包已有的测试函数，并支持设置前置与后置函数。

假定有一个测试函数：

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

one can create an equivalent test case instance as follows:

```
testcase = unittest.FunctionTestCase(testSomething)
```

If there are additional set-up and tear-down methods that should be called as part of the test case's operation, they can also be provided like so:

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

To make migrating existing test suites easier, *unittest* supports tests raising *AssertionError* to indicate test failure. However, it is recommended that you use the explicit `TestCase.fail*()` and `TestCase.assert*()` methods instead, as future versions of *unittest* may treat *AssertionError* differently.

注解： Even though *FunctionTestCase* can be used to quickly convert an existing test base over to a *unittest*-based system, this approach is not recommended. Taking the time to set up proper *TestCase* subclasses will make future test refactorings infinitely easier.

In some cases, the existing tests may have been written using the *doctest* module. If so, *doctest* provides a `DocTestSuite` class that can automatically build *unittest.TestSuite* instances from the existing *doctest*-based tests.

25.3.6 跳过测试与预计的失败

2.7 新版功能.

Unittest supports skipping individual test methods and even whole classes of tests. In addition, it supports marking a test as an "expected failure," a test that is broken and will fail, but shouldn't be counted as a failure on a *TestResult*.

Skipping a test is simply a matter of using the *skip() decorator* or one of its conditional variants.

跳过测试的基本用法如下：

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")
```

(下页继续)

(续上页)

```
    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass
```

在F嗦模式下运行以上测试例子时，程序输出如下：

```
test_format (__main__.MyTestCase) ... skipped 'not supported in this library version'
test_nothing (__main__.MyTestCase) ... skipped 'demonstrating skipping'
test_windows_support (__main__.MyTestCase) ... skipped 'requires Windows'

----------------------------------------------------------------------
Ran 3 tests in 0.005s

OK (skipped=3)
```

跳过测试类的写法跟跳过测试方法的写法相似:

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

TestCase.setUp() 也可以跳过测试。可以用于所需资源不可用的情况下跳过接下来的测试。

使用*expectedFailure()* 装饰器表明这个测试预计失败。:

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

It’s easy to roll your own skipping decorators by making a decorator that calls *skip()* on the test when it wants it to be skipped. This decorator skips the test unless the passed object has a certain attribute:

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

The following decorators implement test skipping and expected failures:

unittest.**skip**(*reason*)
: 跳过被此装饰器装饰的测试。*reason* 为测试被跳过的原因。

unittest.**skipIf**(*condition*, *reason*)
: 当 *condition* 为真时，跳过被装饰的测试。

unittest.**skipUnless**(*condition*, *reason*)
: 跳过被装饰的测试，除非 *condition* 为真。

unittest.**expectedFailure**()
: Mark the test as an expected failure. If the test fails when run, the test is not counted as a failure.

exception unittest.**SkipTest**(*reason*)
引发此异常以跳过一个测试。

通常来说，你可以使用*TestCase.skipTest()* 或其中一个跳过测试的装饰器实现跳过测试的功能，而不是直接引发此异常。

Skipped tests will not have `setUp()` or `tearDown()` run around them. Skipped classes will not have `setUpClass()` or `tearDownClass()` run.

25.3.7 类与函数

本节深入介绍了*unittest* 的 API。

测试用例

class unittest.**TestCase**(*methodName='runTest'*)
Instances of the *TestCase* class represent the smallest testable units in the *unittest* universe. This class is intended to be used as a base class, with specific tests being implemented by concrete subclasses. This class implements the interface needed by the test runner to allow it to drive the test, and methods that the test code can use to check for and report various kinds of failure.

Each instance of *TestCase* will run a single test method: the method named *methodName*. If you remember, we had an earlier example that went something like this:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_size'))
    suite.addTest(WidgetTestCase('test_resize'))
    return suite
```

Here, we create two instances of `WidgetTestCase`, each of which runs a single test.

methodName defaults to `runTest()`.

TestCase instances provide three groups of methods: one group used to run the test, another used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.

Methods in the first group (running the test) are:

setUp()
Method called to prepare the test fixture. This is called immediately before calling the test method; other than *AssertionError* or *SkipTest*, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

tearDown()
Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception, other than *AssertionError* or *SkipTest*, raised by this method will be considered an additional error rather than a test failure (thus increasing the total number of reported errors). This method will only be called if the *setUp()* succeeds, regardless of the outcome of the test method. The default implementation does nothing.

setUpClass()
A class method called before tests in an individual class are run. `setUpClass` is called with the class as the only argument and must be decorated as a *classmethod()*:

```
@classmethod
def setUpClass(cls):
    ...
```

查看*Class and Module Fixtures* 获取更详细的说明。

2.7 新版功能.

tearDownClass()
A class method called after tests in an individual class have run. `tearDownClass` is called with the class as the only argument and must be decorated as a *classmethod()*:

```
@classmethod
def tearDownClass(cls):
    ...
```

查看*Class and Module Fixtures* 获取更详细的说明。

2.7 新版功能.

run(*result=None*)
Run the test, collecting the result into the test result object passed as *result*. If *result* is omitted or `None`, a temporary result object is created (by calling the *defaultTestResult()* method) and used. The result object is not returned to *run()*' s caller.

The same effect may be had by simply calling the *TestCase* instance.

skipTest(*reason*)
Calling this during a test method or *setUp()* skips the current test. See 跳过测试与预计的失败 for more information.

2.7 新版功能.

debug()
Run the test without collecting the result. This allows exceptions raised by the test to be propagated to the caller, and can be used to support running tests under a debugger.

The *TestCase* class provides several assert methods to check for and report failures. The following table lists the most commonly used methods (see the tables below for more assert methods):

Method	Checks that	New in
assertEqual(a, b)	`a == b`	
assertNotEqual(a, b)	`a != b`	
assertTrue(x)	`bool(x) is True`	
assertFalse(x)	`bool(x) is False`	
assertIs(a, b)	`a is b`	2.7
assertIsNot(a, b)	`a is not b`	2.7
assertIsNone(x)	`x is None`	2.7
assertIsNotNone(x)	`x is not None`	2.7
assertIn(a, b)	`a in b`	2.7
assertNotIn(a, b)	`a not in b`	2.7
assertIsInstance(a, b)	`isinstance(a, b)`	2.7
assertNotIsInstance(a, b)	`not isinstance(a, b)`	2.7

All the assert methods (except *assertRaises()*, *assertRaisesRegexp()*) accept a *msg* argument that, if specified, is used as the error message on failure (see also *longMessage*).

assertEqual(*first*, *second*, *msg=None*)
Test that *first* and *second* are equal. If the values do not compare equal, the test will fail.

In addition, if *first* and *second* are the exact same type and one of list, tuple, dict, set, frozenset or unicode or any type that a subclass registers with `addTypeEqualityFunc()` the type-specific equality function will be called in order to generate a more useful default error message (see also the *list of type-specific methods*).

在 2.7 版更改: Added the automatic calling of type-specific equality function.

assertNotEqual(*first*, *second*, *msg=None*)
Test that *first* and *second* are not equal. If the values do compare equal, the test will fail.

assertTrue(*expr*, *msg=None*)
assertFalse(*expr*, *msg=None*)
Test that *expr* is true (or false).

Note that this is equivalent to `bool(expr) is True` and not to `expr is True` (use `assertIs(expr, True)` for the latter). This method should also be avoided when more specific methods are available (e.g. `assertEqual(a, b)` instead of `assertTrue(a == b)`), because they provide a better error message in case of failure.

assertIs(*first*, *second*, *msg=None*)
assertIsNot(*first*, *second*, *msg=None*)
Test that *first* and *second* evaluate (or don't evaluate) to the same object.

2.7 新版功能.

assertIsNone(*expr*, *msg=None*)
assertIsNotNone(*expr*, *msg=None*)
Test that *expr* is (or is not) `None`.

2.7 新版功能.

assertIn(*first*, *second*, *msg=None*)
assertNotIn(*first*, *second*, *msg=None*)
Test that *first* is (or is not) in *second*.

2.7 新版功能.

assertIsInstance(*obj*, *cls*, *msg=None*)
assertNotIsInstance(*obj*, *cls*, *msg=None*)
Test that *obj* is (or is not) an instance of *cls* (which can be a class or a tuple of classes, as supported by `isinstance()`). To check for the exact type, use `assertIs(type(obj), cls)`.

2.7 新版功能.

It is also possible to check that exceptions and warnings are raised using the following methods:

Method	Checks that	New in
`assertRaises(exc, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *exc*	
`assertRaisesRegexp(exc, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *exc* and the message matches regex *r*	2.7

assertRaises(*exception*, *callable*, **args*, ***kwds*)
assertRaises(*exception*)
Test that an exception is raised when *callable* is called with any positional or keyword arguments that are also passed to `assertRaises()`. The test passes if *exception* is raised, is an error if another exception

is raised, or fails if no exception is raised. To catch any of a group of exceptions, a tuple containing the exception classes may be passed as *exception*.

If only the *exception* argument is given, returns a context manager so that the code under test can be written inline rather than as a function:

```
with self.assertRaises(SomeException):
    do_something()
```

The context manager will store the caught exception object in its `exception` attribute. This can be useful if the intention is to perform additional checks on the exception raised:

```
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

在 2.7 版更改: Added the ability to use *assertRaises()* as a context manager.

assertRaisesRegexp(*exception*, *regexp*, *callable*, **args*, ***kwds*)
assertRaisesRegexp(*exception*, *regexp*)

Like *assertRaises()* but also tests that *regexp* matches on the string representation of the raised exception. *regexp* may be a regular expression object or a string containing a regular expression suitable for use by *re.search()*. Examples:

```
self.assertRaisesRegexp(ValueError, "invalid literal for.*XYZ'$",
                        int, 'XYZ')
```

或者:

```
with self.assertRaisesRegexp(ValueError, 'literal'):
   int('XYZ')
```

2.7 新版功能.

There are also other methods used to perform more specific checks, such as:

Method	Checks that	New in
assertAlmostEqual(a, b)	`round(a-b, 7) == 0`	
assertNotAlmostEqual(a, b)	`round(a-b, 7) != 0`	
assertGreater(a, b)	`a > b`	2.7
assertGreaterEqual(a, b)	`a >= b`	2.7
assertLess(a, b)	`a < b`	2.7
assertLessEqual(a, b)	`a <= b`	2.7
assertRegexpMatches(s, r)	`r.search(s)`	2.7
assertNotRegexpMatches(s, r)	`not r.search(s)`	2.7
assertItemsEqual(a, b)	sorted(a) == sorted(b) and works with unhashable objs	2.7
assertDictContainsSubset(a, b)	all the key/value pairs in *a* exist in *b*	2.7

assertAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)
assertNotAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)

Test that *first* and *second* are approximately (or not approximately) equal by computing the difference, round-

ing to the given number of decimal *places* (default 7), and comparing to zero. Note that these methods round the values to the given number of *decimal places* (i.e. like the `round()` function) and not *significant digits*.

If *delta* is supplied instead of *places* then the difference between *first* and *second* must be less or equal to (or greater than) *delta*.

Supplying both *delta* and *places* raises a `TypeError`.

在 2.7 版更改: `assertAlmostEqual()` automatically considers almost equal objects that compare equal. `assertNotAlmostEqual()` automatically fails if the objects compare equal. Added the *delta* keyword argument.

assertGreater(*first*, *second*, *msg=None*)
assertGreaterEqual(*first*, *second*, *msg=None*)
assertLess(*first*, *second*, *msg=None*)
assertLessEqual(*first*, *second*, *msg=None*)

Test that *first* is respectively >, >=, < or <= than *second* depending on the method name. If not, the test will fail:

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

2.7 新版功能.

assertRegexpMatches(*text*, *regexp*, *msg=None*)

Test that a *regexp* search matches *text*. In case of failure, the error message will include the pattern and the *text* (or the pattern and the part of *text* that unexpectedly matched). *regexp* may be a regular expression object or a string containing a regular expression suitable for use by `re.search()`.

2.7 新版功能.

assertNotRegexpMatches(*text*, *regexp*, *msg=None*)

Verifies that a *regexp* search does not match *text*. Fails with an error message including the pattern and the part of *text* that matches. *regexp* may be a regular expression object or a string containing a regular expression suitable for use by `re.search()`.

2.7 新版功能.

assertItemsEqual(*actual*, *expected*, *msg=None*)

Test that sequence *expected* contains the same elements as *actual*, regardless of their order. When they don't, an error message listing the differences between the sequences will be generated.

Duplicate elements are *not* ignored when comparing *actual* and *expected*. It verifies if each element has the same count in both sequences. It is the equivalent of `assertEqual(sorted(expected), sorted(actual))` but it works with sequences of unhashable objects as well.

In Python 3, this method is named `assertCountEqual`.

2.7 新版功能.

assertDictContainsSubset(*expected*, *actual*, *msg=None*)

Tests whether the key/value pairs in dictionary *actual* are a superset of those in *expected*. If not, an error message listing the missing keys and mismatched values is generated.

2.7 新版功能.

3.2 版后已移除.

The `assertEqual()` method dispatches the equality check for objects of the same type to different type-specific methods. These methods are already implemented for most of the built-in types, but it's also possible to register new methods using `addTypeEqualityFunc()`:

addTypeEqualityFunc(*typeobj*, *function*)
Registers a type-specific method called by `assertEqual()` to check if two objects of exactly the same *typeobj* (not subclasses) compare equal. *function* must take two positional arguments and a third msg=None keyword argument just as `assertEqual()` does. It must raise `self.failureException(msg)` when inequality between the first two parameters is detected –possibly providing useful information and explaining the inequalities in details in the error message.

2.7 新版功能.

The list of type-specific methods automatically used by `assertEqual()` are summarized in the following table. Note that it's usually not necessary to invoke these methods directly.

Method	Used to compare	New in
`assertMultiLineEqual(a, b)`	strings	2.7
`assertSequenceEqual(a, b)`	sequences	2.7
`assertListEqual(a, b)`	lists	2.7
`assertTupleEqual(a, b)`	tuples	2.7
`assertSetEqual(a, b)`	sets or frozensets	2.7
`assertDictEqual(a, b)`	dicts	2.7

assertMultiLineEqual(*first*, *second*, *msg=None*)
Test that the multiline string *first* is equal to the string *second*. When not equal a diff of the two strings highlighting the differences will be included in the error message. This method is used by default when comparing Unicode strings with `assertEqual()`.

2.7 新版功能.

assertSequenceEqual(*seq1*, *seq2*, *msg=None*, *seq_type=None*)
Tests that two sequences are equal. If a *seq_type* is supplied, both *seq1* and *seq2* must be instances of *seq_type* or a failure will be raised. If the sequences are different an error message is constructed that shows the difference between the two.

This method is not called directly by `assertEqual()`, but it's used to implement `assertListEqual()` and `assertTupleEqual()`.

2.7 新版功能.

assertListEqual(*list1*, *list2*, *msg=None*)
assertTupleEqual(*tuple1*, *tuple2*, *msg=None*)
Tests that two lists or tuples are equal. If not, an error message is constructed that shows only the differences between the two. An error is also raised if either of the parameters are of the wrong type. These methods are used by default when comparing lists or tuples with `assertEqual()`.

2.7 新版功能.

assertSetEqual(*set1*, *set2*, *msg=None*)
Tests that two sets are equal. If not, an error message is constructed that lists the differences between the sets. This method is used by default when comparing sets or frozensets with `assertEqual()`.

Fails if either of *set1* or *set2* does not have a `set.difference()` method.

2.7 新版功能.

assertDictEqual(*expected*, *actual*, *msg=None*)
Test that two dictionaries are equal. If not, an error message is constructed that shows the differences in the dictionaries. This method will be used by default to compare dictionaries in calls to `assertEqual()`.

2.7 新版功能.

Finally the `TestCase` provides the following methods and attributes:

fail(*msg=None*)
Signals a test failure unconditionally, with *msg* or `None` for the error message.

failureException
This class attribute gives the exception raised by the test method. If a test framework needs to use a specialized exception, possibly to carry additional information, it must subclass this exception in order to "play fair" with the framework. The initial value of this attribute is *AssertionError*.

longMessage
If set to `True` then any explicit failure message you pass in to the *assert methods* will be appended to the end of the normal failure message. The normal messages contain useful information about the objects involved, for example the message from assertEqual shows you the repr of the two unequal objects. Setting this attribute to `True` allows you to have a custom error message in addition to the normal one.

This attribute defaults to `False`, meaning that a custom message passed to an assert method will silence the normal message.

The class setting can be overridden in individual tests by assigning an instance attribute to `True` or `False` before calling the assert methods.

2.7 新版功能.

maxDiff
This attribute controls the maximum length of diffs output by assert methods that report diffs on failure. It defaults to 80*8 characters. Assert methods affected by this attribute are *assertSequenceEqual()* (including all the sequence comparison methods that delegate to it), *assertDictEqual()* and *assertMultiLineEqual()*.

Setting `maxDiff` to `None` means that there is no maximum length of diffs.

2.7 新版功能.

Testing frameworks can use the following methods to collect information on the test:

countTestCases()
Return the number of tests represented by this test object. For *TestCase* instances, this will always be `1`.

defaultTestResult()
Return an instance of the test result class that should be used for this test case class (if no other result instance is provided to the *run()* method).

For *TestCase* instances, this will always be an instance of *TestResult*; subclasses of *TestCase* should override this as necessary.

id()
Return a string identifying the specific test case. This is usually the full name of the test method, including the module and class name.

shortDescription()
Returns a description of the test, or `None` if no description has been provided. The default implementation of this method returns the first line of the test method's docstring, if available, or *None*.

addCleanup(*function*, **args*, ***kwargs*)
Add a function to be called after *tearDown()* to cleanup resources used during the test. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into *addCleanup()* when they are added.

If *setUp()* fails, meaning that *tearDown()* is not called, then any cleanup functions added will still be called.

2.7 新版功能.

doCleanups()
This method is called unconditionally after `tearDown()`, or after `setUp()` if `setUp()` raises an exception.

It is responsible for calling all the cleanup functions added by `addCleanup()`. If you need cleanup functions to be called *prior* to `tearDown()` then you can call `doCleanups()` yourself.

`doCleanups()` pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

2.7 新版功能.

class unittest.**FunctionTestCase**(*testFunc*, *setUp=None*, *tearDown=None*, *description=None*)
This class implements the portion of the `TestCase` interface which allows the test runner to drive the test, but does not provide the methods which test code can use to check and report errors. This is used to create test cases using legacy test code, allowing it to be integrated into a `unittest`-based test framework.

Deprecated aliases

For historical reasons, some of the `TestCase` methods had one or more aliases that are now deprecated. The following table lists the correct names along with their deprecated aliases:

方法名	Deprecated alias(es)
`assertEqual()`	failUnlessEqual, assertEquals
`assertNotEqual()`	failIfEqual
`assertTrue()`	failUnless, assert_
`assertFalse()`	failIf
`assertRaises()`	failUnlessRaises
`assertAlmostEqual()`	failUnlessAlmostEqual
`assertNotAlmostEqual()`	failIfAlmostEqual

2.7 版后已移除: the aliases listed in the second column

Grouping tests

class unittest.**TestSuite**(*tests=()*)
This class represents an aggregation of individual test cases and test suites. The class presents the interface needed by the test runner to allow it to be run as any other test case. Running a `TestSuite` instance is the same as iterating over the suite, running each test individually.

If *tests* is given, it must be an iterable of individual test cases or other test suites that will be used to build the suite initially. Additional methods are provided to add test cases and suites to the collection later on.

`TestSuite` objects behave much like `TestCase` objects, except they do not actually implement a test. Instead, they are used to aggregate tests into groups of tests that should be run together. Some additional methods are available to add tests to `TestSuite` instances:

addTest(*test*)
Add a `TestCase` or `TestSuite` to the suite.

addTests(*tests*)
Add all the tests from an iterable of `TestCase` and `TestSuite` instances to this test suite.

This is equivalent to iterating over *tests*, calling `addTest()` for each element.

`TestSuite` shares the following methods with `TestCase`:

run(*result*)
Run the tests associated with this suite, collecting the result into the test result object passed as *result*. Note that unlike *TestCase.run()*, *TestSuite.run()* requires the result object to be passed in.

debug()
Run the tests associated with this suite without collecting the result. This allows exceptions raised by the test to be propagated to the caller and can be used to support running tests under a debugger.

countTestCases()
Return the number of tests represented by this test object, including all individual tests and sub-suites.

__iter__()
Tests grouped by a *TestSuite* are always accessed by iteration. Subclasses can lazily provide tests by overriding *__iter__()*. Note that this method maybe called several times on a single suite (for example when counting tests or comparing for equality) so the tests returned must be the same for repeated iterations.

在 2.7 版更改: In earlier versions the *TestSuite* accessed tests directly rather than through iteration, so overriding *__iter__()* wasn't sufficient for providing tests.

In the typical usage of a *TestSuite* object, the *run()* method is invoked by a `TestRunner` rather than by the end-user test harness.

Loading and running tests

class unittest.**TestLoader**
The *TestLoader* class is used to create test suites from classes and modules. Normally, there is no need to create an instance of this class; the *unittest* module provides an instance that can be shared as *unittest.defaultTestLoader*. Using a subclass or instance, however, allows customization of some configurable properties.

TestLoader objects have the following methods:

loadTestsFromTestCase(*testCaseClass*)
Return a suite of all test cases contained in the *TestCase*-derived `testCaseClass`.

loadTestsFromModule(*module*)
Return a suite of all test cases contained in the given module. This method searches *module* for classes derived from *TestCase* and creates an instance of the class for each test method defined for the class.

注解: While using a hierarchy of *TestCase*-derived classes can be convenient in sharing fixtures and helper functions, defining test methods on base classes that are not intended to be instantiated directly does not play well with this method. Doing so, however, can be useful when the fixtures are different and defined in subclasses.

If a module provides a `load_tests` function it will be called to load the tests. This allows modules to customize test loading. This is the *load_tests protocol*.

在 2.7 版更改: Support for `load_tests` added.

loadTestsFromName(*name*, *module=None*)
Return a suite of all test cases given a string specifier.

The specifier *name* is a "dotted name" that may resolve either to a module, a test case class, a test method within a test case class, a *TestSuite* instance, or a callable object which returns a *TestCase* or *TestSuite* instance. These checks are applied in the order listed here; that is, a method on a possible test case class will be picked up as "a test method within a test case class", rather than "a callable object".

For example, if you have a module `SampleTests` containing a *TestCase*-derived class `SampleTestCase` with three test methods (`test_one()`, `test_two()`, and `test_three()`), the specifier `'SampleTests.SampleTestCase'` would cause this method to return a suite which will run all three test methods. Using the specifier `'SampleTests.SampleTestCase.test_two'` would cause it to return a test suite which will run only the `test_two()` test method. The specifier can refer to modules and packages which have not been imported; they will be imported as a side-effect.

The method optionally resolves *name* relative to the given *module*.

loadTestsFromNames(*names*, *module=None*)
Similar to *loadTestsFromName()*, but takes a sequence of names rather than a single name. The return value is a test suite which supports all the tests defined for each name.

getTestCaseNames(*testCaseClass*)
Return a sorted sequence of method names found within *testCaseClass*; this should be a subclass of *TestCase*.

discover(*start_dir*, *pattern='test*.py'*, *top_level_dir=None*)
Find all the test modules by recursing into subdirectories from the specified start directory, and return a TestSuite object containing them. Only test files that match *pattern* will be loaded. (Using shell style pattern matching.) Only module names that are importable (i.e. are valid Python identifiers) will be loaded.

All test modules must be importable from the top level of the project. If the start directory is not the top level directory then the top level directory must be specified separately.

If importing a module fails, for example due to a syntax error, then this will be recorded as a single error and discovery will continue.

If a test package name (directory with `__init__.py`) matches the pattern then the package will be checked for a `load_tests` function. If this exists then it will be called with *loader*, *tests*, *pattern*.

If load_tests exists then discovery does *not* recurse into the package, `load_tests` is responsible for loading all tests in the package.

The pattern is deliberately not stored as a loader attribute so that packages can continue discovery themselves. *top_level_dir* is stored so `load_tests` does not need to pass this argument in to `loader.discover()`.

start_dir can be a dotted module name as well as a directory.

2.7 新版功能.

The following attributes of a *TestLoader* can be configured either by subclassing or assignment on an instance:

testMethodPrefix
String giving the prefix of method names which will be interpreted as test methods. The default value is `'test'`.

This affects *getTestCaseNames()* and all the `loadTestsFrom*()` methods.

sortTestMethodsUsing
Function to be used to compare method names when sorting them in *getTestCaseNames()* and all the `loadTestsFrom*()` methods. The default value is the built-in *cmp()* function; the attribute can also be set to *None* to disable the sort.

suiteClass
Callable object that constructs a test suite from a list of tests. No methods on the resulting object are needed. The default value is the *TestSuite* class.

This affects all the `loadTestsFrom*()` methods.

class unittest.**TestResult**
This class is used to compile information about which tests have succeeded and which have failed.

A *TestResult* object stores the results of a set of tests. The *TestCase* and *TestSuite* classes ensure that results are properly recorded; test authors do not need to worry about recording the outcome of tests.

Testing frameworks built on top of *unittest* may want access to the *TestResult* object generated by running a set of tests for reporting purposes; a *TestResult* instance is returned by the `TestRunner.run()` method for this purpose.

TestResult instances have the following attributes that will be of interest when inspecting the results of running a set of tests:

errors
A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents a test which raised an unexpected exception.

在 2.2 版更改: Contains formatted tracebacks instead of *sys.exc_info()* results.

failures
A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents a test where a failure was explicitly signalled using the `TestCase.assert*()` methods.

在 2.2 版更改: Contains formatted tracebacks instead of *sys.exc_info()* results.

skipped
A list containing 2-tuples of *TestCase* instances and strings holding the reason for skipping the test.

2.7 新版功能.

expectedFailures
A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents an expected failure of the test case.

unexpectedSuccesses
A list containing *TestCase* instances that were marked as expected failures, but succeeded.

shouldStop
Set to `True` when the execution of tests should stop by *stop()*.

testsRun
The total number of tests run so far.

buffer
If set to true, `sys.stdout` and `sys.stderr` will be buffered in between *startTest()* and *stopTest()* being called. Collected output will only be echoed onto the real `sys.stdout` and `sys.stderr` if the test fails or errors. Any output is also attached to the failure / error message.

2.7 新版功能.

failfast
If set to true *stop()* will be called on the first failure or error, halting the test run.

2.7 新版功能.

wasSuccessful()
Return `True` if all tests run so far have passed, otherwise returns `False`.

stop()
This method can be called to signal that the set of tests being run should be aborted by setting the *shouldStop* attribute to `True`. `TestRunner` objects should respect this flag and return without running any additional tests.

For example, this feature is used by the *TextTestRunner* class to stop the test framework when the user signals an interrupt from the keyboard. Interactive tools which provide `TestRunner` implementations can use this in a similar manner.

The following methods of the *TestResult* class are used to maintain the internal data structures, and may be extended in subclasses to support additional reporting requirements. This is particularly useful in building tools which support interactive reporting while tests are being run.

startTest(*test*)
Called when the test case *test* is about to be run.

stopTest(*test*)
Called after the test case *test* has been executed, regardless of the outcome.

startTestRun()
Called once before any tests are executed.

2.7 新版功能.

stopTestRun()
Called once after all tests are executed.

2.7 新版功能.

addError(*test*, *err*)
Called when the test case *test* raises an unexpected exception. *err* is a tuple of the form returned by *sys.exc_info()*: `(type, value, traceback)`.

The default implementation appends a tuple `(test, formatted_err)` to the instance's *errors* attribute, where *formatted_err* is a formatted traceback derived from *err*.

addFailure(*test*, *err*)
Called when the test case *test* signals a failure. *err* is a tuple of the form returned by *sys.exc_info()*: `(type, value, traceback)`.

The default implementation appends a tuple `(test, formatted_err)` to the instance's *failures* attribute, where *formatted_err* is a formatted traceback derived from *err*.

addSuccess(*test*)
Called when the test case *test* succeeds.

The default implementation does nothing.

addSkip(*test*, *reason*)
Called when the test case *test* is skipped. *reason* is the reason the test gave for skipping.

The default implementation appends a tuple `(test, reason)` to the instance's *skipped* attribute.

addExpectedFailure(*test*, *err*)
Called when the test case *test* fails, but was marked with the *expectedFailure()* decorator.

The default implementation appends a tuple `(test, formatted_err)` to the instance's *expectedFailures* attribute, where *formatted_err* is a formatted traceback derived from *err*.

addUnexpectedSuccess(*test*)
Called when the test case *test* was marked with the *expectedFailure()* decorator, but succeeded.

The default implementation appends the test to the instance's *unexpectedSuccesses* attribute.

class unittest.**TextTestResult**(*stream*, *descriptions*, *verbosity*)
A concrete implementation of *TestResult* used by the *TextTestRunner*.

2.7 新版功能: This class was previously named `_TextTestResult`. The old name still exists as an alias but is deprecated.

unittest.**defaultTestLoader**
Instance of the *TestLoader* class intended to be shared. If no customization of the *TestLoader* is needed, this instance can be used instead of repeatedly creating new instances.

class unittest.**TextTestRunner**(*stream=sys.stderr*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*)

A basic test runner implementation which prints results on standard error. It has a few configurable parameters, but is essentially very simple. Graphical applications which run test suites should provide alternate implementations.

_makeResult()

This method returns the instance of `TestResult` used by `run()`. It is not intended to be called directly, but can be overridden in subclasses to provide a custom `TestResult`.

`_makeResult()` instantiates the class or callable passed in the `TextTestRunner` constructor as the `resultclass` argument. It defaults to *TextTestResult* if no `resultclass` is provided. The result class is instantiated with the following arguments:

```
stream, descriptions, verbosity
```

unittest.**main**([*module*[, *defaultTest*[, *argv*[, *testRunner*[, *testLoader*[, *exit*[, *verbosity*[, *failfast*[, *catchbreak*[, *buffer*]]]]]]]]])

A command-line program that loads a set of tests from *module* and runs them; this is primarily for making test modules conveniently executable. The simplest use for this function is to include the following line at the end of a test script:

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

You can run tests with more detailed information by passing in the verbosity argument:

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

The *defaultTest* argument is the name of the test to run if no test names are specified via *argv*. If not specified or `None` and no test names are provided via *argv*, all tests found in *module* are run.

The *argv* argument can be a list of options passed to the program, with the first element being the program name. If not specified or `None`, the values of *sys.argv* are used.

The *testRunner* argument can either be a test runner class or an already created instance of it. By default `main` calls *sys.exit()* with an exit code indicating success or failure of the tests run.

The *testLoader* argument has to be a *TestLoader* instance, and defaults to *defaultTestLoader*.

`main` supports being used from the interactive interpreter by passing in the argument `exit=False`. This displays the result on standard output without calling *sys.exit()*:

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

The *failfast*, *catchbreak* and *buffer* parameters have the same effect as the same-name *command-line options*.

Calling `main` actually returns an instance of the `TestProgram` class. This stores the result of the tests run as the `result` attribute.

在 2.7 版更改: The *exit*, *verbosity*, *failfast*, *catchbreak* and *buffer* parameters were added.

load_tests Protocol

2.7 新版功能.

Modules or packages can customize how tests are loaded from them during normal test runs or test discovery by implementing a function called `load_tests`.

If a test module defines `load_tests` it will be called by *TestLoader.loadTestsFromModule()* with the following arguments:

```
load_tests(loader, standard_tests, None)
```

It should return a *TestSuite*.

loader is the instance of *TestLoader* doing the loading. *standard_tests* are the tests that would be loaded by default from the module. It is common for test modules to only want to add or remove tests from the standard set of tests. The third argument is used when loading packages as part of test discovery.

A typical `load_tests` function that loads tests from a specific set of *TestCase* classes may look like:

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

If discovery is started, either from the command line or by calling *TestLoader.discover()*, with a pattern that matches a package name then the package `__init__.py` will be checked for `load_tests`.

注解: The default pattern is `'test*.py'`. This matches all Python files that start with `'test'` but *won't* match any test directories.

A pattern like `'test*'` will match test packages as well as modules.

If the package `__init__.py` defines `load_tests` then it will be called and discovery not continued into the package. `load_tests` is called with the following arguments:

```
load_tests(loader, standard_tests, pattern)
```

This should return a *TestSuite* representing all the tests from the package. (`standard_tests` will only contain tests collected from `__init__.py`.)

Because the pattern is passed into `load_tests` the package is free to continue (and potentially modify) test discovery. A 'do nothing' `load_tests` function for a test package would look like:

```
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

25.3.8 Class and Module Fixtures

Class and module level fixtures are implemented in *TestSuite*. When the test suite encounters a test from a new class then `tearDownClass()` from the previous class (if there is one) is called, followed by `setUpClass()` from the new class.

Similarly if a test is from a different module from the previous test then `tearDownModule` from the previous module is run, followed by `setUpModule` from the new module.

After all the tests have run the final `tearDownClass` and `tearDownModule` are run.

Note that shared fixtures do not play well with [potential] features like test parallelization and they break test isolation. They should be used with care.

The default ordering of tests created by the unittest test loaders is to group all tests from the same modules and classes together. This will lead to `setUpClass` / `setUpModule` (etc) being called exactly once per class and module. If you randomize the order, so that tests from different modules and classes are adjacent to each other, then these shared fixture functions may be called multiple times in a single test run.

Shared fixtures are not intended to work with suites with non-standard ordering. A `BaseTestSuite` still exists for frameworks that don't want to support shared fixtures.

If there are any exceptions raised during one of the shared fixture functions the test is reported as an error. Because there is no corresponding test instance an `_ErrorHolder` object (that has the same interface as a *TestCase*) is created to represent the error. If you are just using the standard unittest test runner then this detail doesn't matter, but if you are a framework author it may be relevant.

setUpClass and tearDownClass

These must be implemented as class methods:

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

If you want the `setUpClass` and `tearDownClass` on base classes called then you must call up to them yourself. The implementations in *TestCase* are empty.

If an exception is raised during a `setUpClass` then the tests in the class are not run and the `tearDownClass` is not run. Skipped classes will not have `setUpClass` or `tearDownClass` run. If the exception is a *SkipTest* exception then the class will be reported as having been skipped instead of as an error.

setUpModule and tearDownModule

These should be implemented as functions:

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

If an exception is raised in a `setUpModule` then none of the tests in the module will be run and the `tearDownModule` will not be run. If the exception is a *`SkipTest`* exception then the module will be reported as having been skipped instead of as an error.

25.3.9 Signal Handling

The *`-c/--catch`* command-line option to unittest, along with the `catchbreak` parameter to *`unittest.main()`*, provide more friendly handling of control-C during a test run. With catch break behavior enabled control-C will allow the currently running test to complete, and the test run will then end and report all the results so far. A second control-c will raise a *`KeyboardInterrupt`* in the usual way.

The control-c handling signal handler attempts to remain compatible with code or tests that install their own `signal.SIGINT` handler. If the `unittest` handler is called but *isn't* the installed `signal.SIGINT` handler, i.e. it has been replaced by the system under test and delegated to, then it calls the default handler. This will normally be the expected behavior by code that replaces an installed handler and delegates to it. For individual tests that need `unittest` control-c handling disabled the *`removeHandler()`* decorator can be used.

There are a few utility functions for framework authors to enable control-c handling functionality within test frameworks.

`unittest.`**`installHandler`**`()`
: Install the control-c handler. When a `signal.SIGINT` is received (usually in response to the user pressing control-c) all registered results have *`stop()`* called.

  2.7 新版功能.

`unittest.`**`registerResult`**`(`*`result`*`)`
: Register a *`TestResult`* object for control-c handling. Registering a result stores a weak reference to it, so it doesn't prevent the result from being garbage collected.

  Registering a *`TestResult`* object has no side-effects if control-c handling is not enabled, so test frameworks can unconditionally register all results they create independently of whether or not handling is enabled.

  2.7 新版功能.

`unittest.`**`removeResult`**`(`*`result`*`)`
: Remove a registered result. Once a result has been removed then *`stop()`* will no longer be called on that result object in response to a control-c.

  2.7 新版功能.

`unittest.`**`removeHandler`**`(`*`function=None`*`)`
: When called without arguments this function removes the control-c handler if it has been installed. This function can also be used as a test decorator to temporarily remove the handler while the test is being executed:

  ```
  @unittest.removeHandler
  def test_signal_handling(self):
      ...
  ```

2.7 新版功能.

25.4 2to3 - 自动将 Python 2 代码转为 Python 3 代码

2to3 是一个 Python 程序，它可以用来读取 Python 2.x 版本的代码，并使用一系列的 修复器来将其转换为合法的 Python 3.x 代码。标准库中已经包含了丰富的修复器，这足以处理绝大多数代码。不过 2to3 的支持库 *lib2to3* 是一个很灵活通用的库，所以你也可以为 2to3 编写你自己的修复器。*lib2to3* 也可以用在那些需要自动处理 Python 代码的应用中。

25.4.1 使用 2to3

2to3 通常会作为脚本和 Python 解释器一起安装，你可以在 Python 根目录的 `Tools/scripts` 文件夹下找到它。

2to3 的基本调用参数是一个需要转换的文件或目录列表。对于目录，会递归地寻找其中的 Python 源码。

这里有一个 Python 2.x 的源码文件，`example.py`：

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

它可以在命令行中使用 2to3 转换成 Python 3.x 版本的代码：

```
$ 2to3 example.py
```

这个命令会打印出和源文件的区别。通过传入 `-w` 参数，2to3 也可以把需要的修改写回到原文件中（除非传入了 `-n` 参数，否则会为原始文件创建一个副本）：

```
$ 2to3 -w example.py
```

在转换完成后，`example.py` 看起来像是这样：

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

注释和缩进都会在转换过程中保持不变。

默认情况下，2to3 会执行预定义修复器 的集合。使用 `-l` 参数可以列出所有可用的修复器。使用 `-f` 参数可以明确指定需要使用的修复器集合。而使用 `-x` 参数则可以明确指定不使用的修复器。下面的例子会只使用 `imports` 和 `has_key` 修复器运行：

```
$ 2to3 -f imports -f has_key example.py
```

这个命令会执行除了 `apply` 之外的所有修复器：

```
$ 2to3 -x apply example.py
```

有一些修复器是需要 显式指定的，它们默认不会执行，必须在命令行中列出才会执行。比如下面的例子，除了默认的修复器以外，还会执行 `idioms` 修复器：

```
$ 2to3 -f all -f idioms example.py
```

注意这里使用 all 来启用所有默认的修复器。

有些情况下 2to3 会找到源码中有一些需要修改，但是无法自动处理的代码。在这种情况下，2to3 会在差异处下面打印一个警告信息。你应该定位到相应的代码并对其进行修改，以使其兼容 Python 3.x。

2to3 也可以重构 doctests。使用 -d 开启这个模式。需要注意 * 只有 * doctests 会被重构。这种模式下不需要文件是合法的 Python 代码。举例来说，reST 文档中类似 doctests 的示例也可以使用这个选项进行重构。

-v 选项可以输出更多转换程序的详细信息。

由于某些 print 语句可被解读为函数调用或是语句，2to3 并不是总能读取包含 print 函数的文件。当 2to3 检测到存在 `from __future__ import print_function` 编译器指令时，会修改其内部语法将*print()* 解读为函数。这一变动也可以使用 -p 选项手动开启。使用 -p 来为已经转换过 print 语句的代码运行修复器。

-o 或 --output-dir 选项可以指定将转换后的文件写入其他目录中。由于这种情况下不会覆写原始文件，所以创建副本文件毫无意义，因此也需要使用 -n 选项来禁用创建副本。

2.7.3 新版功能: 增加了 -o 选项。

-W 或 --write-unchanged-files 选项用来告诉 2to3 始终需要输出文件，即使没有任何改动。这在使用 -o 参数时十分有用，这样就可以将整个 Python 源码包完整地转换到另一个目录。这个选项隐含了 -w 选项，否则等于没有作用。

2.7.3 新版功能: 增加了 -W 选项。

--add-suffix 选项接受一个字符串，用来作为后缀附加在输出文件名后面的后面。由于写入的文件名与原始文件不同，所以没有必要创建副本，因此 -n 选项也是必要的。举个例子:

```
$ 2to3 -n -W --add-suffix=3 example.py
```

这样会把转换后的文件写入 example.py3 文件。

2.7.3 新版功能: 增加了 --add-suffix 选项。

将整个项目从一个目录转换到另一个目录可以用这样的命令:

```
$ 2to3 --output-dir=python3-version/mycode -W -n python2-version/mycode
```

25.4.2 修复器

转换代码的每一个步骤都封装在修复器中。可以使用 2to3 -l 来列出可用的修复器。*之前已经提到*，每个修复器都可以独立地打开或是关闭。下面会对各个修复器做更详细的描述。

apply
: 移除对*apply()* 的使用，举例来说，apply(function, *args, **kwargs) 会被转换成 function(*args, **kwargs)。

asserts
: 将已弃用的*unittest* 方法替换为正确的。

从	到
failUnlessEqual(a, b)	*assertEqual(a, b)*
assertEquals(a, b)	*assertEqual(a, b)*
failIfEqual(a, b)	*assertNotEqual(a, b)*
assertNotEquals(a, b)	*assertNotEqual(a, b)*
failUnless(a)	*assertTrue(a)*
assert_(a)	*assertTrue(a)*
failIf(a)	*assertFalse(a)*
failUnlessRaises(exc, cal)	*assertRaises(exc, cal)*
failUnlessAlmostEqual(a, b)	*assertAlmostEqual(a, b)*
assertAlmostEquals(a, b)	*assertAlmostEqual(a, b)*
failIfAlmostEqual(a, b)	*assertNotAlmostEqual(a, b)*
assertNotAlmostEquals(a, b)	*assertNotAlmostEqual(a, b)*

basestring
将*basestring* 转换为*str*。

buffer
将*buffer* 转换为*memoryview*。这个修复器是可选的，因为*memoryview* API 和*buffer* 很相似，但不完全一样。

dict
修复字典迭代方法。*dict.iteritems()* 会转换成*dict.items()*，*dict.iterkeys()* 会转换成*dict.keys()*，*dict.itervalues()* 会转换成*dict.values()*。类似的，*dict.viewitems()*，*dict.viewkeys()* 和*dict.viewvalues()* 会分别转换成*dict.items()*，*dict.keys()* 和*dict.values()*。另外也会将原有的*dict.items()*，*dict.keys()* 和*dict.values()* 方法调用用 list 包装一层。

except
将 except X, T 转换为 except X as T。

exec
Converts the exec statement to the exec() function.

execfile
移除*execfile()* 的使用。*execfile()* 的实参会使用*open()*，*compile()* 和 exec() 包装。

exitfunc
将对*sys.exitfunc* 的赋值改为使用*atexit* 模块代替。

filter
将*filter()* 函数用 list 包装一层。

funcattrs
修复已经重命名的函数属性。比如 my_function.func_closure 会被转换为 my_function.__closure__。

future
移除 from __future__ import new_feature 语句。

getcwdu
将*os.getcwdu()* 重命名为*os.getcwd()*。

has_key
将 dict.has_key(key) 转换为 key in dict。

idioms
这是一个可选的修复器，会进行多种转换，将 Python 代码变成更加常见的写法。类似 type(x) is SomeClass 和 type(x) == SomeClass 的类型对比会被转换成 isinstance(x, SomeClass)。

while 1 转换成 while True。这个修复器还会在合适的地方使用*sorted()* 函数。举个例子，这样的代码块：

```
L = list(some_iterable)
L.sort()
```

会被转换为：

```
L = sorted(some_iterable)
```

import
检测 sibling imports，并将其转换成相对 import。

imports
处理标准库模块的重命名。

imports2
处理标准库中其他模块的重命名。这个修复器由于一些技术上的限制，因此和*imports* 拆分开了。

input
将 input(prompt) 转换为 eval(input(prompt))。

intern
将*intern()* 转换为 sys.intern()。

isinstance
Fixes duplicate types in the second argument of *isinstance()*. For example, isinstance(x, (int, int)) is converted to isinstance(x, (int)).

itertools_imports
移除*itertools.ifilter()*，*itertools.izip()* 以及*itertools.imap()* 的 import。对*itertools.ifilterfalse()* 的 import 也会替换成 itertools.filterfalse()。

itertools
修改*itertools.ifilter()*，*itertools.izip()* 和*itertools.imap()* 的调用为对应的内建实现。*itertools.ifilterfalse()* 会替换成 itertools.filterfalse()。

long
将*long* 重命名为*int*。

map
用 list 包装*map()*。同时也会将 map(None, x) 替换为 list(x)。使用 from future_builtins import map 禁用这个修复器。

metaclass
将老的元类语法(类体中的 __metaclass__ = Meta)替换为新的(class X(metaclass=Meta))。

methodattrs
修复老的方法属性名。例如 meth.im_func 会被转换为 meth.__func__。

ne
转换老的不等语法，将 <> 转为 !=。

next
Converts the use of iterator's *next()* methods to the *next()* function. It also renames *next()* methods to __next__().

nonzero
将 __nonzero__() 转换为 __bool__()。

numliterals
将八进制字面量转为新的语法。

paren
在列表生成式中增加必须的括号。例如将 `[x for x in 1, 2]` 转换为 `[x for x in (1, 2)]`。

print
Converts the `print` statement to the *print()* function.

raise
Converts `raise E, V` to `raise E(V)`, and `raise E, V, T` to `raise E(V).with_traceback(T)`. If `E` is a tuple, the translation will be incorrect because substituting tuples for exceptions has been removed in Python 3.

raw_input
将*raw_input()* 转换为*input()*。

reduce
将*reduce()* 转换为*functools.reduce()*。

renames
将*sys.maxint* 转换为*sys.maxsize*。

repr
将反引号 repr 表达式替换为*repr()* 函数。

set_literal
将*set* 构造函数替换为 set literals 写法。这个修复器是可选的。

standarderror
将*StandardError* 重命名为*Exception*。

sys_exc
将弃用的*sys.exc_value*, *sys.exc_type*, *sys.exc_traceback* 替换为*sys.exc_info()* 的用法。

throw
修复生成器的 `throw()` 方法的 API 变更。

tuple_params
移除隐式的元组参数解包。这个修复器会插入临时变量。

types
修复*type* 模块中一些成员的移除引起的代码问题。

unicode
将*unicode* 重命名为*str*。

urllib
将*urllib* 和*urllib2* 重命名为*urllib* 包。

ws_comma
移除逗号分隔的元素之间多余的空白。这个修复器是可选的。

xrange
将*xrange()* 重命名为*range()*，并用 `list` 包装原有的*range()*。

xreadlines
将 `for x in file.xreadlines()` 转换为 `for x in file`。

zip
用 `list` 包装*zip()*。如果使用了 `from future_builtins import zip` 的话会禁用。

25.4.3 lib2to3 ——2to3 支持库

注解： *lib2to3* API 并不稳定，并可能在未来大幅修改。

25.5 test —Regression tests package for Python

注解： The *test* package is meant for internal use by Python only. It is documented for the benefit of the core developers of Python. Any use of this package outside of Python's standard library is discouraged as code mentioned here can change or be removed without notice between releases of Python.

The *test* package contains all regression tests for Python as well as the modules *test.support* and test.regrtest. *test.support* is used to enhance your tests while test.regrtest drives the testing suite.

Each module in the *test* package whose name starts with test_ is a testing suite for a specific module or feature. All new tests should be written using the *unittest* or *doctest* module. Some older tests are written using a "traditional" testing style that compares output printed to sys.stdout; this style of test is considered deprecated.

参见：

Module *unittest* Writing PyUnit regression tests.

***doctest*—文档测试模块** Tests embedded in documentation strings.

25.5.1 Writing Unit Tests for the test package

It is preferred that tests that use the *unittest* module follow a few guidelines. One is to name the test module by starting it with test_ and end it with the name of the module being tested. The test methods in the test module should start with test_ and end with a description of what the method is testing. This is needed so that the methods are recognized by the test driver as test methods. Also, no documentation string for the method should be included. A comment (such as # Tests function returns only True or False) should be used to provide documentation for test methods. This is done because documentation strings get printed out if they exist and thus what test is being run is not stated.

A basic boilerplate is often used:

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...
```

(下页继续)

(续上页)

```
    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

def test_main():
    support.run_unittest(MyTestCase1,
                         MyTestCase2,
                         ... list other tests ...
                         )

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

This boilerplate code allows the testing suite to be run by `test.regrtest` as well as on its own as a script.

The goal for regression testing is to try to break code. This leads to a few guidelines to be followed:

- The testing suite should exercise all classes, functions, and constants. This includes not just the external API that is to be presented to the outside world but also "private" code.
- Whitebox testing (examining the code being tested when the tests are being written) is preferred. Blackbox testing (testing only the published user interface) is not complete enough to make sure all boundary and edge cases are tested.
- Make sure all possible values are tested including invalid ones. This makes sure that not only all valid values are acceptable but also that improper values are handled correctly.
- Exhaust as many code paths as possible. Test where branching occurs and thus tailor input to make sure as many different paths through the code are taken.
- Add an explicit test for any bugs discovered for the tested code. This will make sure that the error does not crop up again if the code is changed in the future.
- Make sure to clean up after your tests (such as close and remove all temporary files).
- If a test is dependent on a specific condition of the operating system then verify the condition already exists before attempting the test.
- Import as few modules as possible and do it as soon as possible. This minimizes external dependencies of tests and also minimizes possible anomalous behavior from side-effects of importing a module.
- Try to maximize code reuse. On occasion, tests will vary by something as small as what type of input is used. Minimize code duplication by subclassing a basic test class with a class that specifies the input:

```
class TestFuncAcceptsSequences(unittest.TestCase):

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)
```

(下页继续)

(续上页)

```
class AcceptLists(TestFuncAcceptsSequences):
    arg = [1, 2, 3]

class AcceptStrings(TestFuncAcceptsSequences):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequences):
    arg = (1, 2, 3)
```

参见:

Test Driven Development A book by Kent Beck on writing tests before code.

25.5.2 Running tests using the command-line interface

The `test.regrtest` module can be run as a script to drive Python's regression test suite, thanks to the `-m` option: **`python -m test.regrtest`**. Running the script by itself automatically starts running all regression tests in the *`test`* package. It does this by finding all modules in the package whose name starts with `test_`, importing them, and executing the function `test_main()` if present. The names of tests to execute may also be passed to the script. Specifying a single regression test (**`python -m test.regrtest test_spam`**) will minimize output and only print whether the test passed or failed and thus minimize output.

Running `test.regrtest` directly allows what resources are available for tests to use to be set. You do this by using the `-u` command-line option. Specifying `all` as the value for the `-u` option enables all possible resources: **`python -m test.regrtest -uall`**. If all but one resource is desired (a more common case), a comma-separated list of resources that are not desired may be listed after `all`. The command **`python -m test.regrtest -uall,-audio,-largefile`** will run `test.regrtest` with all resources except the `audio` and `largefile` resources. For a list of all resources and more command-line options, run **`python -m test.regrtest -h`**.

Some other ways to execute the regression tests depend on what platform the tests are being executed on. On Unix, you can run **`make test`** at the top-level directory where Python was built. On Windows, executing **`rt.bat`** from your `PCBuild` directory will run all regression tests.

在 2.7.14 版更改: The *`test`* package can be run as a script: **`python -m test`**. This works the same as running the `test.regrtest` module.

25.6 `test.support` —Utility functions for tests

注解: The `test.test_support` module has been renamed to *`test.support`* in Python 3.x and 2.7.14. The name `test.test_support` has been retained as an alias in 2.7.

The *`test.support`* module provides support for Python's regression tests.

This module defines the following exceptions:

exception `test.support.`**`TestFailed`**
Exception to be raised when a test fails. This is deprecated in favor of *`unittest`*-based tests and *`unittest.TestCase`*'s assertion methods.

exception `test.support.`**`ResourceDenied`**
Subclass of *`unittest.SkipTest`*. Raised when a resource (such as a network connection) is not available. Raised by the *`requires()`* function.

The *test.support* module defines the following constants:

test.support.**verbose**
: *True* when verbose output is enabled. Should be checked when more detailed information is desired about a running test. *verbose* is set by test.regrtest.

test.support.**have_unicode**
: *True* when Unicode support is available.

test.support.**is_jython**
: *True* if the running interpreter is Jython.

test.support.**TESTFN**
: Set to a name that is safe to use as the name of a temporary file. Any temporary file that is created should be closed and unlinked (removed).

test.support.**TEST_HTTP_URL**
: Define the URL of a dedicated HTTP server for the network tests.

The *test.support* module defines the following functions:

test.support.**forget**(*module_name*)
: Remove the module named *module_name* from sys.modules and delete any byte-compiled files of the module.

test.support.**is_resource_enabled**(*resource*)
: Return *True* if *resource* is enabled and available. The list of available resources is only set when test.regrtest is executing the tests.

test.support.**requires**(*resource*[, *msg*])
: Raise *ResourceDenied* if *resource* is not available. *msg* is the argument to *ResourceDenied* if it is raised. Always returns *True* if called by a function whose __name__ is '__main__'. Used when tests are executed by test.regrtest.

test.support.**findfile**(*filename*)
: Return the path to the file named *filename*. If no match is found *filename* is returned. This does not equal a failure since it could be the path to the file.

test.support.**run_unittest**(**classes*)
: Execute *unittest.TestCase* subclasses passed to the function. The function scans the classes for methods starting with the prefix test_ and executes the tests individually.

  It is also legal to pass strings as parameters; these should be keys in sys.modules. Each associated module will be scanned by unittest.TestLoader.loadTestsFromModule(). This is usually seen in the following test_main() function:

```
def test_main():
    support.run_unittest(__name__)
```

  This will run all tests defined in the named module.

test.support.**check_warnings**(**filters*, *quiet=True*)
: A convenience wrapper for *warnings.catch_warnings()* that makes it easier to test that a warning was correctly raised. It is approximately equivalent to calling warnings.catch_warnings(record=True) with *warnings.simplefilter()* set to always and with the option to automatically validate the results that are recorded.

  check_warnings accepts 2-tuples of the form ("message regexp", WarningCategory) as positional arguments. If one or more *filters* are provided, or if the optional keyword argument *quiet* is *False*, it checks to make sure the warnings are as expected: each specified filter must match at least one of the warnings raised by the enclosed code or the test fails, and if any warnings are raised that do not match any of the specified filters the test fails. To disable the first of these checks, set *quiet* to *True*.

If no arguments are specified, it defaults to:

```
check_warnings(("", Warning), quiet=True)
```

In this case all warnings are caught and no errors are raised.

On entry to the context manager, a `WarningRecorder` instance is returned. The underlying warnings list from *`catch_warnings()`* is available via the recorder object's *`warnings`* attribute. As a convenience, the attributes of the object representing the most recent warning can also be accessed directly through the recorder object (see example below). If no warning has been raised, then any of the attributes that would otherwise be expected on an object representing a warning will return *`None`*.

The recorder object also has a `reset()` method, which clears the warnings list.

The context manager is designed to be used like this:

```
with check_warnings(("assertion is always true", SyntaxWarning),
                    ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

In this case if either warning was not raised, or some other warning was raised, *`check_warnings()`* would raise an error.

When a test needs to look more deeply into the warnings, rather than just checking whether or not they occurred, code like this can be used:

```
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0
```

Here all warnings will be caught, and the test code tests the captured warnings directly.

2.6 新版功能.

在 2.7 版更改: New optional arguments *filters* and *quiet*.

`test.support.`**`check_py3k_warnings`**`(`*filters*, *quiet=False*`)`

Similar to *`check_warnings()`*, but for Python 3 compatibility warnings. If `sys.py3kwarning == 1`, it checks if the warning is effectively raised. If `sys.py3kwarning == 0`, it checks that no warning is raised. It accepts 2-tuples of the form `("message regexp", WarningCategory)` as positional arguments. When the optional keyword argument *quiet* is *`True`*, it does not fail if a filter catches nothing. Without arguments, it defaults to:

```
check_py3k_warnings(("", DeprecationWarning), quiet=False)
```

2.7 新版功能.

`test.support.`**`captured_stdout`**`()`

This is a context manager that runs the `with` statement body using a *`StringIO.StringIO`* object as sys.stdout. That object can be retrieved using the `as` clause of the `with` statement.

Example use:

```
with captured_stdout() as s:
    print "hello"
assert s.getvalue() == "hello\n"
```

2.6 新版功能.

test.support.**import_module**(*name*, *deprecated=False*)
This function imports and returns the named module. Unlike a normal import, this function raises *unittest.SkipTest* if the module cannot be imported.

Module and package deprecation messages are suppressed during this import if *deprecated* is *True*.

2.7 新版功能.

test.support.**import_fresh_module**(*name*, *fresh=()*, *blocked=()*, *deprecated=False*)
This function imports and returns a fresh copy of the named Python module by removing the named module from sys.modules before doing the import. Note that unlike *reload()*, the original module is not affected by this operation.

fresh is an iterable of additional module names that are also removed from the sys.modules cache before doing the import.

blocked is an iterable of module names that are replaced with 0 in the module cache during the import to ensure that attempts to import them raise *ImportError*.

The named module and any modules named in the *fresh* and *blocked* parameters are saved before starting the import and then reinserted into sys.modules when the fresh import is complete.

Module and package deprecation messages are suppressed during this import if *deprecated* is *True*.

This function will raise *unittest.SkipTest* if the named module cannot be imported.

Example use:

```
# Get copies of the warnings module for testing without
# affecting the version being used by the rest of the test suite
# One copy uses the C implementation, the other is forced to use
# the pure Python fallback implementation
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

2.7 新版功能.

The *test.support* module defines the following classes:

class test.support.**TransientResource**(*exc*[, ***kwargs*])
Instances are a context manager that raises *ResourceDenied* if the specified exception type is raised. Any keyword arguments are treated as attribute/value pairs to be compared against any exception raised within the with statement. Only if all pairs match properly against attributes on the exception is *ResourceDenied* raised.

2.6 新版功能.

class test.support.**EnvironmentVarGuard**
Class used to temporarily set or unset environment variables. Instances can be used as a context manager and have a complete dictionary interface for querying/modifying the underlying os.environ. After exit from the context manager all changes to environment variables done through this instance will be rolled back.

2.6 新版功能.

在 2.7 版更改: Added dictionary interface.

EnvironmentVarGuard.**set**(*envvar*, *value*)
Temporarily set the environment variable envvar to the value of value.

EnvironmentVarGuard.**unset**(*envvar*)
Temporarily unset the environment variable `envvar`.

class test.support.**WarningsRecorder**
Class used to record warnings for unit tests. See documentation of *check_warnings()* above for more details.

2.6 新版功能.

CHAPTER 26

调试和分析

这些库可以帮助你进行 Python 开发：调试器使你能够逐步执行代码，分析堆栈帧并设置断点等，而分析器运行代码并为你提供执行时间的详细分类，从而使你能够找出你程序中的瓶颈。

26.1 bdb —Debugger framework

Source code: Lib/bdb.py

The *bdb* module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

定义了以下异常：

exception bdb.**BdbQuit**
Exception raised by the *Bdb* class for quitting the debugger.

The *bdb* module also defines two classes:

class bdb.**Breakpoint**(*self*, *file*, *line*, *temporary=0*, *cond=None*, *funcname=None*)
This class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.

Breakpoints are indexed by number through a list called `bpbynumber` and by `(file, line)` pairs through `bplist`. The former points to a single instance of class *Breakpoint*. The latter points to a list of such instances since there may be more than one breakpoint per line.

When creating a breakpoint, its associated filename should be in canonical form. If a *funcname* is defined, a breakpoint hit will be counted when the first line of that function is executed. A conditional breakpoint always counts a hit.

Breakpoint instances have the following methods:

deleteMe()
Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.

enable()
Mark the breakpoint as enabled.

disable()
Mark the breakpoint as disabled.

pprint([*out*])
Print all the information about the breakpoint:

- The breakpoint number.
- If it is temporary or not.
- Its file,line position.
- The condition that causes a break.
- If it must be ignored the next N times.
- The breakpoint hit count.

class bdb.**Bdb**(*skip=None*)
The *Bdb* class acts as a generic Python debugger base class.

This class takes care of the details of the trace facility; a derived class should implement user interaction. The standard debugger class (*pdb.Pdb*) is an example.

The *skip* argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. Whether a frame is considered to originate in a certain module is determined by the `__name__` in the frame globals.

2.7 新版功能: The *skip* argument.

The following methods of *Bdb* normally don't need to be overridden.

canonic(*filename*)
Auxiliary method for getting a filename in a canonical form, that is, as a case-normalized (on case-insensitive filesystems) absolute path, stripped of surrounding angle brackets.

reset()
Set the `botframe`, `stopframe`, `returnframe` and `quitting` attributes with values ready to start debugging.

trace_dispatch(*frame*, *event*, *arg*)
This function is installed as the trace function of debugged frames. Its return value is the new trace function (in most cases, that is, itself).

The default implementation decides how to dispatch a frame, depending on the type of event (passed as a string) that is about to be executed. *event* can be one of the following:

- `"line"`: A new line of code is going to be executed.
- `"call"`: A function is about to be called, or another code block entered.
- `"return"`: A function or other code block is about to return.
- `"exception"`: An exception has occurred.
- `"c_call"`: A C function is about to be called.
- `"c_return"`: A C function has returned.
- `"c_exception"`: A C function has raised an exception.

For the Python events, specialized functions (see below) are called. For the C events, no action is taken.

The *arg* parameter depends on the previous event.

See the documentation for *sys.settrace()* for more information on the trace function. For more information on code and frame objects, refer to types.

dispatch_line(*frame*)
If the debugger should stop on the current line, invoke the *user_line()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_line()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_call(*frame*, *arg*)
If the debugger should stop on this function call, invoke the *user_call()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_call()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_return(*frame*, *arg*)
If the debugger should stop on this function return, invoke the *user_return()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_return()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_exception(*frame*, *arg*)
If the debugger should stop at this exception, invokes the *user_exception()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_exception()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

Normally derived classes don't override the following methods, but they may if they want to redefine the definition of stopping and breakpoints.

stop_here(*frame*)
This method checks if the *frame* is somewhere below `botframe` in the call stack. `botframe` is the frame in which debugging started.

break_here(*frame*)
This method checks if there is a breakpoint in the filename and line belonging to *frame* or, at least, in the current function. If the breakpoint is a temporary one, this method deletes it.

break_anywhere(*frame*)
This method checks if there is a breakpoint in the filename of the current frame.

Derived classes should override these methods to gain control over debugger operation.

user_call(*frame*, *argument_list*)
This method is called from *dispatch_call()* when there is the possibility that a break might be necessary anywhere inside the called function.

user_line(*frame*)
This method is called from *dispatch_line()* when either *stop_here()* or *break_here()* yields `True`.

user_return(*frame*, *return_value*)
This method is called from *dispatch_return()* when *stop_here()* yields `True`.

user_exception(*frame*, *exc_info*)
This method is called from *dispatch_exception()* when *stop_here()* yields `True`.

do_clear(*arg*)
Handle how a breakpoint must be removed when it is a temporary one.

This method must be implemented by derived classes.

Derived classes and clients can call the following methods to affect the stepping state.

set_step()
Stop after one line of code.

set_next(*frame*)
Stop on the next line in or below the given frame.

set_return(*frame*)
Stop when returning from the given frame.

set_until(*frame*)
Stop when the line with the line no greater than the current one is reached or when returning from current frame.

set_trace([*frame*])
Start debugging from *frame*. If *frame* is not specified, debugging starts from caller's frame.

set_continue()
Stop only at breakpoints or when finished. If there are no breakpoints, set the system trace function to `None`.

set_quit()
Set the `quitting` attribute to `True`. This raises *BdbQuit* in the next call to one of the `dispatch_*()` methods.

Derived classes and clients can call the following methods to manipulate breakpoints. These methods return a string containing an error message if something went wrong, or `None` if all is well.

set_break(*filename*, *lineno*, *temporary=0*, *cond=None*, *funcname=None*)
Set a new breakpoint. If the *lineno* line doesn't exist for the *filename* passed as argument, return an error message. The *filename* should be in canonical form, as described in the *canonic()* method.

clear_break(*filename*, *lineno*)
Delete the breakpoints in *filename* and *lineno*. If none were set, an error message is returned.

clear_bpbynumber(*arg*)
Delete the breakpoint which has the index *arg* in the `Breakpoint.bpbynumber`. If *arg* is not numeric or out of range, return an error message.

clear_all_file_breaks(*filename*)
Delete all breakpoints in *filename*. If none were set, an error message is returned.

clear_all_breaks()
Delete all existing breakpoints.

get_break(*filename*, *lineno*)
Check if there is a breakpoint for *lineno* of *filename*.

get_breaks(*filename*, *lineno*)
Return all breakpoints for *lineno* in *filename*, or an empty list if none are set.

get_file_breaks(*filename*)
Return all breakpoints in *filename*, or an empty list if none are set.

get_all_breaks()
Return all breakpoints that are set.

Derived classes and clients can call the following methods to get a data structure representing a stack trace.

get_stack(*f*, *t*)
Get a list of records for a frame and all higher (calling) and lower frames, and the size of the higher part.

format_stack_entry(*frame_lineno*[, *lprefix=': '*])
Return a string with information about a stack entry, identified by a `(frame, lineno)` tuple:

- The canonical form of the filename which contains the frame.
- The function name, or `"<lambda>"`.
- The input arguments.
- The return value.
- The line of code (if it exists).

The following two methods can be called by clients to use a debugger to debug a *statement*, given as a string.

run(*cmd*[, *globals*[, *locals*]])
Debug a statement executed via the `exec` statement. *globals* defaults to `__main__.__dict__`, *locals* defaults to *globals*.

runeval(*expr*[, *globals*[, *locals*]])
Debug an expression executed via the *eval()* function. *globals* and *locals* have the same meaning as in *run()*.

runctx(*cmd*, *globals*, *locals*)
For backwards compatibility. Calls the *run()* method.

runcall(*func*, **args*, ***kwds*)
Debug a single function call, and return its result.

Finally, the module defines the following functions:

bdb.**checkfuncname**(*b*, *frame*)
Check whether we should break here, depending on the way the breakpoint *b* was set.

If it was set via line number, it checks if `b.line` is the same as the one in the frame also passed as argument. If the breakpoint was set via function name, we have to check we are in the right frame (the right function) and if we are in its first executable line.

bdb.**effective**(*file*, *line*, *frame*)
Determine if there is an effective (active) breakpoint at this line of code. Return a tuple of the breakpoint and a boolean that indicates if it is ok to delete a temporary breakpoint. Return `(None, None)` if there is no matching breakpoint.

bdb.**set_trace**()
Start debugging with a *Bdb* instance from caller's frame.

26.2 pdb —Python 的调试器

源代码： Lib/pdb.py

The module *pdb* defines an interactive source code debugger for Python programs. It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control.

The debugger is extensible —it is actually defined as the class *Pdb*. This is currently undocumented but easily understood by reading the source. The extension interface uses the modules *bdb* and *cmd*.

The debugger's prompt is `(Pdb)`. Typical usage to run a program under control of the debugger is:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

`pdb.py` can also be invoked as a script to debug other scripts. For example:

```
python -m pdb myscript.py
```

When invoked as a script, pdb will automatically enter post-mortem debugging if the program being debugged exits abnormally. After post-mortem debugging (or after normal exit of the program), pdb will restart the program. Automatic restarting preserves pdb's state (such as breakpoints) and in most cases is more useful than quitting the debugger upon program's exit.

2.4 新版功能: Restarting post-mortem behavior added.

The typical usage to break into the debugger from a running program is to insert

```
import pdb; pdb.set_trace()
```

at the location you want to break into the debugger. You can then step through the code following this statement, and continue running without the debugger using the `c` command.

The typical usage to inspect a crashed program is:

```
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print spam
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print spam
(Pdb)
```

The module defines the following functions; each enters the debugger in a slightly different way:

pdb.**run**(*statement*[, *globals*[, *locals*]])
: Execute the *statement* (given as a string) under debugger control. The debugger prompt appears before any code is executed; you can set breakpoints and type `continue`, or you can step through the statement using `step` or `next` (all these commands are explained below). The optional *globals* and *locals* arguments specify the environment in which the code is executed; by default the dictionary of the module *__main__* is used. (See the explanation of the `exec` statement or the *eval()* built-in function.)

pdb.**runeval**(*expression*[, *globals*[, *locals*]])
: Evaluate the *expression* (given as a string) under debugger control. When *runeval()* returns, it returns the value of the expression. Otherwise this function is similar to *run()*.

pdb.**runcall**(*function*[, *argument*, ...])
Call the *function* (a function or method object, not a string) with the given arguments. When *runcall()* returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

pdb.**set_trace**()
Enter the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails).

pdb.**post_mortem**([*traceback*])
Enter post-mortem debugging of the given *traceback* object. If no *traceback* is given, it uses the one of the exception that is currently being handled (an exception must be being handled if the default is to be used).

pdb.**pm**()
Enter post-mortem debugging of the traceback found in *sys.last_traceback*.

The `run*` functions and *set_trace()* are aliases for instantiating the *Pdb* class and calling the method of the same name. If you want to access further features, you have to do this yourself:

class pdb.**Pdb**(*completekey='tab'*, *stdin=None*, *stdout=None*, *skip=None*)
Pdb is the debugger class.

The *completekey*, *stdin* and *stdout* arguments are passed to the underlying *cmd.Cmd* class; see the description there.

The *skip* argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns.[1]

Example call to enable tracing with *skip*:

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

2.7 新版功能: The *skip* argument.

run(*statement*[, *globals*[, *locals*]])
runeval(*expression*[, *globals*[, *locals*]])
runcall(*function*[, *argument*, ...])
set_trace()
See the documentation for the functions explained above.

26.3 Debugger Commands

The debugger recognizes the following commands. Most commands can be abbreviated to one or two letters; e.g. `h(elp)` means that either `h` or `help` can be used to enter the help command (but not `he` or `hel`, nor `H` or `Help` or `HELP`). Arguments to commands must be separated by whitespace (spaces or tabs). Optional arguments are enclosed in square brackets (`[]`) in the command syntax; the square brackets must not be typed. Alternatives in the command syntax are separated by a vertical bar (`|`).

Entering a blank line repeats the last command entered. Exception: if the last command was a `list` command, the next 11 lines are listed.

Commands that the debugger doesn't recognize are assumed to be Python statements and are executed in the context of the program being debugged. Python statements can also be prefixed with an exclamation point (`!`). This is a powerful way to inspect the program being debugged; it is even possible to change a variable or call a function. When an exception occurs in such a statement, the exception name is printed but the debugger's state is not changed.

Multiple commands may be entered on a single line, separated by `;;`. (A single `;` is not used as it is the separator for multiple commands in a line that is passed to the Python parser.) No intelligence is applied to separating the commands; the input is split at the first `;;` pair, even if it is in the middle of a quoted string.

[1] Whether a frame is considered to originate in a certain module is determined by the `__name__` in the frame globals.

The debugger supports aliases. Aliases can have parameters which allows one a certain level of adaptability to the context under examination.

If a file `.pdbrc` exists in the user's home directory or in the current directory, it is read in and executed as if it had been typed at the debugger prompt. This is particularly useful for aliases. If both files exist, the one in the home directory is read first and aliases defined there can be overridden by the local file.

h(elp) [*command*] Without argument, print the list of available commands. With a *command* as argument, print help about that command. `help pdb` displays the full documentation file; if the environment variable `PAGER` is defined, the file is piped through that command instead. Since the *command* argument must be an identifier, `help exec` must be entered to get help on the `!` command.

w(here) Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame, which determines the context of most commands.

d(own) Move the current frame one level down in the stack trace (to a newer frame).

u(p) Move the current frame one level up in the stack trace (to an older frame).

b(reak) [[*filename*:]*lineno* | *function*[, *condition*]] With a *lineno* argument, set a break there in the current file. With a *function* argument, set a break at the first executable statement within that function. The line number may be prefixed with a filename and a colon, to specify a breakpoint in another file (probably one that hasn't been loaded yet). The file is searched on `sys.path`. Note that each breakpoint is assigned a number to which all the other breakpoint commands refer.

If a second argument is present, it is an expression which must evaluate to true before the breakpoint is honored.

Without argument, list all breaks, including for each breakpoint, the number of times that breakpoint has been hit, the current ignore count, and the associated condition if any.

tbreak [[*filename*:]*lineno* | *function*[, *condition*]] Temporary breakpoint, which is removed automatically when it is first hit. The arguments are the same as break.

cl(ear) [*filename:lineno* | *bpnumber* [*bpnumber* …]] With a *filename:lineno* argument, clear all the breakpoints at this line. With a space separated list of breakpoint numbers, clear those breakpoints. Without argument, clear all breaks (but first ask confirmation).

disable [*bpnumber* [*bpnumber* …]] Disables the breakpoints given as a space separated list of breakpoint numbers. Disabling a breakpoint means it cannot cause the program to stop execution, but unlike clearing a breakpoint, it remains in the list of breakpoints and can be (re-)enabled.

enable [*bpnumber* [*bpnumber* …]] Enables the breakpoints specified.

ignore *bpnumber* [*count*] Sets the ignore count for the given breakpoint number. If count is omitted, the ignore count is set to 0. A breakpoint becomes active when the ignore count is zero. When non-zero, the count is decremented each time the breakpoint is reached and the breakpoint is not disabled and any associated condition evaluates to true.

condition *bpnumber* [*condition*] Condition is an expression which must evaluate to true before the breakpoint is honored. If condition is absent, any existing condition is removed; i.e., the breakpoint is made unconditional.

commands [*bpnumber*] Specify a list of commands for breakpoint number *bpnumber*. The commands themselves appear on the following lines. Type a line containing just 'end' to terminate the commands. An example:

```
(Pdb) commands 1
(com) print some_variable
(com) end
(Pdb)
```

To remove all commands from a breakpoint, type commands and follow it immediately with end; that is, give no commands.

With no *bpnumber* argument, commands refers to the last breakpoint set.

You can use breakpoint commands to start your program up again. Simply use the continue command, or step, or any other command that resumes execution.

Specifying any command resuming execution (currently continue, step, next, return, jump, quit and their abbreviations) terminates the command list (as if that command was immediately followed by end). This is because any time you resume execution (even with a simple next or step), you may encounter another breakpoint—which could have its own command list, leading to ambiguities about which list to execute.

If you use the 'silent' command in the command list, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the other commands print anything, you see no sign that the breakpoint was reached.

2.5 新版功能.

s(tep) Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).

n(ext) Continue execution until the next line in the current function is reached or it returns. (The difference between `next` and `step` is that `step` stops inside a called function, while `next` executes called functions at (nearly) full speed, only stopping at the next line in the current function.)

unt(il) Continue execution until the line with the line number greater than the current one is reached or when returning from current frame.

2.6 新版功能.

r(eturn) Continue execution until the current function returns.

c(ont(inue)) Continue execution, only stop when a breakpoint is encountered.

j(ump) ***lineno*** Set the next line that will be executed. Only available in the bottom-most frame. This lets you jump back and execute code again, or jump forward to skip code that you don't want to run.

It should be noted that not all jumps are allowed —for instance it is not possible to jump into the middle of a `for` loop or out of a `finally` clause.

l(ist) [*first*[, *last*]] List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With one argument, list 11 lines around at that line. With two arguments, list the given range; if the second argument is less than the first, it is interpreted as a count.

a(rgs) Print the argument list of the current function.

p *expression* Evaluate the *expression* in the current context and print its value.

注解： `print` can also be used, but is not a debugger command —this executes the Python `print` statement.

---

pp *expression* Like the `p` command, except the value of the expression is pretty-printed using the *`pprint`* module.

alias [*name* [command]] Creates an alias called *name* that executes *command*. The command must *not* be enclosed in quotes. Replaceable parameters can be indicated by `%1`, `%2`, and so on, while `%*` is replaced by all the parameters. If no command is given, the current alias for *name* is shown. If no arguments are given, all aliases are listed.

Aliases may be nested and can contain anything that can be legally typed at the pdb prompt. Note that internal pdb commands *can* be overridden by aliases. Such a command is then hidden until the alias is removed. Aliasing is recursively applied to the first word of the command line; all other words in the line are left alone.

As an example, here are two useful aliases (especially when placed in the `.pdbrc` file):

```
#Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print "%1.",k,"=",%1.__dict__[k]
#Print instance variables in self
alias ps pi self
```

unalias *name* Deletes the specified alias.

[!]*statement* Execute the (one-line) *statement* in the context of the current stack frame. The exclamation point can be omitted unless the first word of the statement resembles a debugger command. To set a global variable, you can prefix the assignment command with a `global` command on the same line, e.g.:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run [*args* **…]** Restart the debugged Python program. If an argument is supplied, it is split with "shlex" and the result is used as the new sys.argv. History, breakpoints, actions and debugger options are preserved. "restart" is an alias for "run".

2.6 新版功能.

q(uit) Quit from the debugger. The program being executed is aborted.

备注

26.4 Python 分析器

源代码： Lib/profile.py and Lib/pstats.py

26.4.1 分析器简介

cProfile 和*profile* 提供了 Python 程序的 *deterministic profiling* 。*profile* 是一组统计数据，描述程序的各个部分执行的频率和时间。这些统计数据可以通过*pstats* 模块格式化为报告。

The Python standard library provides three different implementations of the same profiling interface:

1. 对于大多数用户，建议使用*cProfile*；这是一个 C 扩展插件，开销合理，适合于分析长时间运行的程序。该插件基于 `lsprof`，由 Brett Rosen 和 Ted Chaotter 贡献。

   2.5 新版功能.

2. *profile* 是一个纯 Python 模块（*cProfile* 就是模仿其接口的 C 实现），但它会显著增加配置程序的开销。如果你正在尝试以某种方式扩展分析器，则使用此模块可能会更容易完成任务。该模块最初由 Jim Roskind 设计和编写。

   在 2.4 版更改: Now also reports the time spent in calls to built-in functions and methods.

3. *hotshot* was an experimental C module that focused on minimizing the overhead of profiling, at the expense of longer data post-processing times. It is no longer maintained and may be dropped in a future version of Python.

   在 2.5 版更改: The results should be more meaningful than in the past: the timing core contained a critical bug.

The *profile* and *cProfile* modules export the same interface, so they are mostly interchangeable; *cProfile* has a much lower overhead but is newer and might not be available on all systems. *cProfile* is really a compatibility layer on top of the internal `_lsprof` module. The *hotshot* module is reserved for specialized usage.

注解： profiler 分析器模块被设计为给指定的程序提供执行概要文件，而不是用于基准测试目的（*timeit* 才是用于此目标的，它能获得合理准确的结果）。这特别适用于将 Python 代码与 C 代码进行基准测试：分析器为 Python 代码引入开销，但不会为 C 级函数引入开销，因此 C 代码似乎比任何 Python 代码都更快。

26.4.2 即时用户手册

本节是为“不想阅读手册”的用户提供的。它提供了非常简短的概述，并允许用户快速对现有应用程序执行评测。

要分析采用单个参数的函数，可以执行以下操作：

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

（如果*cProfile*在您的系统上不可用，请使用*profile*。）

上述操作将运行*re.compile()*并打印分析结果，如下所示：

```
      197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 re.py:212(compile)
     1    0.000    0.000    0.001    0.001 re.py:268(_compile)
     1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
     1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
     4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
   3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

第一行显示监听了 197 个调用。在这些调用中，有 192 个是 *原始的*，这意味着调用不是通过递归引发的。下一行: `Ordered by: standard name`，表示最右边列中的文本字符串用于对输出进行排序。列标题包括：

ncalls for the number of calls,

tottime 在指定函数中花费的总时间（不包括调用子函数的时间）

percall 是 `tottime` 除以 `ncalls` 的商

cumtime 指定的函数及其所有子函数（从调用到退出）消耗的累积时间。这个数字对于递归函数来说是准确的。

percall 是 `cumtime` 除以原始调用（次数）的商

filename:lineno(function) 提供相应数据的每个函数

如果第一列中有两个数字（例如 3/1），则表示函数递归。第二个值是原始调用次数，第一个是调用的总次数。请注意，当函数不递归时，这两个值是相同的，并且只打印单个数字。

profile 运行结束时，打印输出不是必须的。也可以通过为 `run()` 函数指定文件名，将结果保存到文件中：

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

pstats.Stats 类从文件中读取 profile 结果，并以各种方式对其进行格式化。

The file *cProfile* can also be invoked as a script to profile another script. For example:

```
python -m cProfile [-o output_file] [-s sort_order] myscript.py
```

-o 将 profile 结果写入文件而不是标准输出

-s 指定*sort_stats()* 排序值之一以对输出进行排序。这仅适用于未提供 -o 的情况

The *pstats* module's *Stats* class has a variety of methods for manipulating and printing the data saved into a profile results file:

```
import pstats
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

The *strip_dirs()* method removed the extraneous path from all the module names. The *sort_stats()* method sorted all the entries according to the standard module/line/name string that is printed. The *print_stats()* method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats('name')
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```
p.sort_stats('cumulative').print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```
p.sort_stats('time').print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

您也可以尝试：

```
p.sort_stats('file').print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with `__init__` in them). As one final example, you could try:

```
p.sort_stats('time', 'cum').print_stats(.5, 'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: `.5`) of its original size, then only lines containing `init` are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (`p` is still sorted according to the last criteria) do:

```
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual, or guess what the following functions do:

```
p.print_callees()
p.add('restats')
```

Invoked as a script, the *pstats* module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using *cmd*) and interactive help.

26.4.3 profile 和 cProfile 模块参考

profile 和*cProfile* 模块都提供下列函数：

profile.**run**(*command*, *filename=None*, *sort=-1*)
This function takes a single argument that can be passed to the `exec()` function, and an optional file name. In all cases this routine executes:

```
exec(command, __main__.__dict__, __main__.__dict__)
```

and gathers profiling statistics from the execution. If no file name is present, then this function automatically creates a *Stats* instance and prints a simple profiling report. If the sort value is specified it is passed to this *Stats* instance to control how the results are sorted.

profile.**runctx**(*command*, *globals*, *locals*, *filename=None*)
This function is similar to *run()*, with added arguments to supply the globals and locals dictionaries for the *command* string. This routine executes:

```
exec(command, globals, locals)
```

and gathers profiling statistics as in the *run()* function above.

class profile.**Profile**(*timer=None*, *timeunit=0.0*, *subcalls=True*, *builtins=True*)
This class is normally only used if more precise control over profiling is needed than what the `cProfile.run()` function provides.

A custom timer can be supplied for measuring how long code takes to run via the *timer* argument. This must be a function that returns a single number representing the current time. If the number is an integer, the *timeunit* specifies a multiplier that specifies the duration of each unit of time. For example, if the timer returns times measured in thousands of seconds, the time unit would be `.001`.

Directly using the *Profile* class allows formatting profile results without writing the profile data to a file:

```
import cProfile, pstats, StringIO
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = StringIO.StringIO()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print s.getvalue()
```

enable()
Start collecting profiling data.

disable()
Stop collecting profiling data.

create_stats()
停止收集分析数据，并在内部将结果记录为当前 profile。

print_stats(*sort=-1*)
Create a *Stats* object based on the current profile and print the results to stdout.

dump_stats(*filename*)
将当前 profile 的结果写入 *filename* 。

run(*cmd*)
Profile the cmd via `exec()`.

runctx(*cmd*, *globals*, *locals*)
Profile the cmd via `exec()` with the specified global and local environment.

runcall(*func*, **args*, ***kwargs*)
Profile `func(*args, **kwargs)`

26.4.4 Stats 类

Analysis of the profiler data is done using the *Stats* class.

class pstats.**Stats**(**filenames or profile*, *stream=sys.stdout*)
This class constructor creates an instance of a "statistics object" from a *filename* (or list of filenames) or from a `Profile` instance. Output will be printed to the stream specified by *stream*.

The file selected by the above constructor must have been created by the corresponding version of *profile* or *cProfile*. To be specific, there is *no* file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers, or the same profiler run on a different operating system. If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing *Stats* object, the *add()* method can be used.

Instead of reading the profile data from a file, a `cProfile.Profile` or *profile.Profile* object can be used as the profile data source.

Stats 对象有以下方法:

strip_dirs()
This method for the *Stats* class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a "random" order, as it was just after object initialization and loading. If *strip_dirs()* causes two function names to be indistinguishable (they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

add(**filenames*)
This method of the *Stats* class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of *profile.run()* or `cProfile.run()`. Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

dump_stats(*filename*)
Save the data loaded into the *Stats* object to a file named *filename*. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the *profile.Profile* and `cProfile.Profile` classes.

2.3 新版功能.

sort_stats(**keys*)
This method modifies the *Stats* object by sorting it according to the supplied criteria. The argument is typically a string identifying the basis of a sort (example: `'time'` or `'name'`).

When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, `sort_stats('name', 'file')` will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

Abbreviations can be used for any key names, as long as the abbreviation is unambiguous. The following are the keys currently defined:

Valid Arg	含义
`'calls'`	调用次数
`'cumulative'`	累积时间
`'cumtime'`	累积时间
`'file'`	文件名
`'filename'`	文件名
`'module'`	文件名
`'ncalls'`	调用次数
`'pcalls'`	原始调用计数
`'line'`	行号
`'name'`	函数名称
`'nfl'`	名称/文件/行
`'stdname'`	标准名称
`'time'`	内部时间
`'tottime'`	内部时间

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle distinction between `'nfl'` and `'stdname'` is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, `'nfl'` does a numeric compare of the line numbers. In fact, `sort_stats('nfl')` is the same as `sort_stats('name', 'file', 'line')`.

For backward-compatibility reasons, the numeric arguments `-1`, `0`, `1`, and `2` are permitted. They are interpreted as `'stdname'`, `'calls'`, `'time'`, and `'cumulative'` respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

reverse_order()
This method for the *Stats* class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is properly selected based on the sort key of choice.

print_stats(**restrictions*)
This method for the *Stats* class prints out a report as described in the *profile.run()* definition.

The order of the printing is based on the last *sort_stats()* operation done on the object (subject to caveats in *add()* and *strip_dirs()*).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a regular expression (to pattern match the standard name that is printed. If several restrictions are provided, then they are applied sequentially. For example:

```
print_stats(.1, 'foo:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename `.*foo:`. In contrast, the command:

```
print_stats('foo:', .1)
```

would limit the list to all functions having file names `.*foo:`, and then proceed to only print the first 10% of them.

print_callers(**restrictions*)
This method for the *Stats* class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by *print_stats()*, and the definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats:

- With *profile*, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.
- With *cProfile*, each caller is preceded by three numbers: the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

print_callees(**restrictions*)
This method for the *Stats* class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the *print_callers()* method.

26.4.5 什么是确定性性能分析？

确定性性能分析旨在反映这样一个事实：即所有 函数调用、函数返回和 异常事件都被监控，并且对这些事件之间的间隔（在此期间用户的代码正在执行）进行精确计时。相反，统计分析（不是由该模块完成）随机采样有效指令指针，并推断时间花费在哪里。后一种技术传统上涉及较少的开销（因为代码不需要检测），但只提供了时间花在哪里的相对指示。

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a *hook* (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

调用计数统计信息可用于识别代码中的错误（意外计数），并识别可能的内联扩展点（高频调用）。内部时间统计可用于识别应仔细优化的“热循环”。累积时间统计可用于识别算法选择上的高级别错误。注意，该分析器中对累积时间的异常处理允许将算法的递归实现与迭代实现的统计信息直接进行比较。

26.4.6 局限性

一个限制是关于时间信息的准确性。确定性性能分析存在一个涉及精度的基本问题。最明显的限制是，底层的“时钟”只以大约 0.001 秒的速度（通常）运行。因此，没有什么测量会比底层时钟更精确。如果进行了足够的测量，那么“误差”将趋于平均。不幸的是，删除第一个错误会导致第二个错误来源。

第二个问题是，从调度事件到分析器获取时间的调用实际获取时钟状态，这需要“一段时间”。类似地，从获取时钟值（然后保存）开始，直到再次执行用户代码为止，退出分析器事件句柄时也存在一定的延迟。因此，多次调用单个函数或调用多个函数通常会累积此错误。以这种方式累积的误差通常小于时钟的精度（小于一个时钟周期），但它 可以累积并变得非常客观。

与开销较低的*cProfile* 相比，*profile* 的问题更为严重。出于这个原因，*profile* 提供了一种针对指定平台的自我校准方法，以便可以在很大程度上（平均地）消除此误差。

26.4.7 准确估量

profile 模块的 profiler 会从每个事件处理时间中减去一个常量，以补偿调用 time 函数和存储结果的开销。默认情况下，常数为 0。对于特定的平台，可用以下程序获得更好修正常数（*局限性*）。

```
import profile
pr = profile.Profile()
for i in range(5):
    print pr.calibrate(10000)
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on a 1.8Ghz Intel Core i5 running Mac OS X, and using Python's time.clock() as the timer, the magical number is about 4.04e-6.

The object of this exercise is to get a fairly consistent result. If your computer is *very* fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

When you have a consistent answer, there are three ways you can use it:[1]

```
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will "less often" show up as negative in profile statistics.

26.4.8 使用自定义计时器

If you want to change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the `Profile` class constructor:

```
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call `your_time_func`. Depending on whether you are using *profile.Profile* or `cProfile.Profile`, `your_time_func`'s return value will be interpreted differently:

profile.Profile `your_time_func` should return a single number, or a list of numbers whose sum is the current time (like what *os.times()* returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose (see *准确估量*). For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (*os.times()* is *pretty* bad, as it returns a tuple of floating point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

[1] Prior to Python 2.2, it was necessary to edit the profiler source code to embed the bias as a literal number. You still can, but that method is no longer described, because no longer needed.

cProfile.Profile `your_time_func` should return a single number. If it returns integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For example, if `your_integer_time_func` returns times measured in thousands of seconds, you would construct the `Profile` instance as follows:

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

As the `cProfile.Profile` class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal `_lsprof` module.

26.5 `hotshot` —High performance logging profiler

2.2 新版功能.

This module provides a nicer interface to the `_hotshot` C module. Hotshot is a replacement for the existing *profile* module. As it's written mostly in C, it should result in a much smaller performance impact than the existing *profile* module.

注解: The *hotshot* module focuses on minimizing the overhead while profiling, at the expense of long data post-processing times. For common usage it is recommended to use *cProfile* instead. *hotshot* is not maintained and might be removed from the standard library in the future.

在 2.5 版更改: The results should be more meaningful than in the past: the timing core contained a critical bug.

注解: The *hotshot* profiler does not yet work well with threads. It is useful to use an unthreaded script to run the profiler over the code you're interested in measuring if at all possible.

---

class `hotshot.`**Profile**(*logfile*[, *lineevents*[, *linetimings*]])

The profiler object. The argument *logfile* is the name of a log file to use for logged profile data. The argument *lineevents* specifies whether to generate events for every source line, or just on function call/return. It defaults to `0` (only log function call/return). The argument *linetimings* specifies whether to record timing information. It defaults to `1` (store timing information).

26.5.1 Profile Objects

Profile objects have the following methods:

`Profile.`**addinfo**(*key*, *value*)

Add an arbitrary labelled value to the profile output.

`Profile.`**close**()

Close the logfile and terminate the profiler.

`Profile.`**fileno**()

Return the file descriptor of the profiler's log file.

`Profile.`**run**(*cmd*)

Profile an `exec`-compatible string in the script environment. The globals from the *__main__* module are used as both the globals and locals for the script.

Profile.**runcall**(*func*, **args*, ***keywords*)
Profile a single call of a callable. Additional positional and keyword arguments may be passed along; the result of the call is returned, and exceptions are allowed to propagate cleanly, while ensuring that profiling is disabled on the way out.

Profile.**runctx**(*cmd*, *globals*, *locals*)
Evaluate an `exec`-compatible string in a specific environment. The string is compiled before profiling begins.

Profile.**start**()
Start the profiler.

Profile.**stop**()
Stop the profiler.

26.5.2 Using hotshot data

2.2 新版功能.

This module loads hotshot profiling data into the standard *pstats* Stats objects.

hotshot.stats.**load**(*filename*)
Load hotshot data from *filename*. Returns an instance of the *pstats.Stats* class.

参见:

Module *profile* The *profile* module's `Stats` class

26.5.3 Example Usage

Note that this example runs the Python "benchmark" pystones. It can take some time to run, and will produce large output files.

```
>>> import hotshot, hotshot.stats, test.pystone
>>> prof = hotshot.Profile("stones.prof")
>>> benchtime, stones = prof.runcall(test.pystone.pystones)
>>> prof.close()
>>> stats = hotshot.stats.load("stones.prof")
>>> stats.strip_dirs()
>>> stats.sort_stats('time', 'calls')
>>> stats.print_stats(20)
         850004 function calls in 10.090 CPU seconds

   Ordered by: internal time, call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    3.295    3.295   10.090   10.090 pystone.py:79(Proc0)
   150000    1.315    0.000    1.315    0.000 pystone.py:203(Proc7)
    50000    1.313    0.000    1.463    0.000 pystone.py:229(Func2)
 .
 .
 .
```

26.6 timeit —测量小代码片段的执行时间

2.3 新版功能.

源码： Lib/timeit.py

该模块提供了一种简单的方法来计算一小段 Python 代码的耗时。它有命令行界面 以及一个可调用 方法。它避免了许多用于测量执行时间的常见陷阱。另见 Tim Peters 对 O'Reilly 出版的 *Python Cookbook* 中 "算法" 章节的介绍。

26.6.1 基本示例

以下示例显示了如何使用命令行界面 来比较三个不同的表达式:

```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 40.3 usec per loop
$ python -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 3: 33.4 usec per loop
$ python -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 3: 25.2 usec per loop
```

这可以通过*Python 接口* 实现

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.8187260627746582
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.7288308143615723
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.5858950614929199
```

Note however that `timeit` will automatically determine the number of repetitions only when the command-line interface is used. In the 例子 section you can find more advanced examples.

26.6.2 Python 接口

该模块定义了三个便利函数和一个公共类:

timeit.**timeit**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *number=1000000*)
Create a `Timer` instance with the given statement, *setup* code and *timer* function and run its `timeit()` method with *number* executions.

2.6 新版功能.

timeit.**repeat**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *repeat=3*, *number=1000000*)
Create a `Timer` instance with the given statement, *setup* code and *timer* function and run its `repeat()` method with the given *repeat* count and *number* executions.

2.6 新版功能.

timeit.**default_timer**()
Define a default timer, in a platform-specific manner. On Windows, `time.clock()` has microsecond granularity, but `time.time()`'s granularity is 1/60th of a second. On Unix, `time.clock()` has 1/100th of a second granularity, and `time.time()` is much more precise. On either platform, `default_timer()` measures wall

clock time, not the CPU time. This means that other processes running on the same computer may interfere with the timing.

class timeit.**Timer**(*stmt='pass'*, *setup='pass'*, *timer=<timer function>*)
用于小代码片段的计数执行速度的类。

The constructor takes a statement to be timed, an additional statement used for setup, and a timer function. Both statements default to `'pass'`; the timer function is platform-dependent (see the module doc string). *stmt* and *setup* may also contain multiple statements separated by `;` or newlines, as long as they don't contain multi-line string literals.

To measure the execution time of the first statement, use the *timeit()* method. The *repeat()* method is a convenience to call *timeit()* multiple times and return a list of results.

在 2.6 版更改: The *stmt* and *setup* parameters can now also take objects that are callable without arguments. This will embed calls to them in a timer function that will then be executed by *timeit()*. Note that the timing overhead is a little larger in this case because of the extra function calls.

timeit(*number=1000000*)
执行 *number* 次主要语句。这将执行一次 setup 语句，然后返回执行主语句多次所需的时间，以秒为单位测量为浮点数。参数是通过循环的次数，默认为一百万。要使用的主语句、setup 语句和 timer 函数将传递给构造函数。

注解: By default, *timeit()* temporarily turns off *garbage collection* during the timing. The advantage of this approach is that it makes independent timings more comparable. This disadvantage is that GC may be an important component of the performance of the function being measured. If so, GC can be re-enabled as the first statement in the *setup* string. For example:

```
timeit.Timer('for i in xrange(10): oct(i)', 'gc.enable()').timeit()
```

---

repeat(*repeat=3*, *number=1000000*)
调用*timeit()* 几次。

这是一个方便的函数，它反复调用*timeit()* ，返回结果列表。第一个参数指定调用*timeit()* 的次数。第二个参数指定*timeit()* 的 *number* 参数。

注解: 从结果向量计算并报告平均值和标准差这些是很诱人的。但是，这不是很有用。在典型情况下，最低值给出了机器运行给定代码段的速度的下限；结果向量中较高的值通常不是由 Python 的速度变化引起的，而是由于其他过程干扰你的计时准确性。所以结果的*min()* 可能是你应该感兴趣的唯一数字。之后，你应该看看整个向量并应用常识而不是统计。

---

print_exc(*file=None*)
帮助程序从计时代码中打印回溯。

典型使用:

```
t = Timer(...)       # outside the try/except
try:
    t.timeit(...)    # or t.repeat(...)
except:
    t.print_exc()
```

The advantage over the standard traceback is that source lines in the compiled template will be displayed. The optional *file* argument directs where the traceback is sent; it defaults to *sys.stderr*.

26.6.3 命令行界面

从命令行调用程序时，使用以下表单:

```
python -m timeit [-n N] [-r N] [-s S] [-t] [-c] [-h] [statement ...]
```

如果了解以下选项:

-n N, **--number**=N
执行‘语句’多少次

-r N, **--repeat**=N
how many times to repeat the timer (default 3)

-s S, **--setup**=S
最初要执行一次的语句（默认为 `pass` ）

-t, **--time**
use *time.time()* (default on all platforms but Windows)

-c, **--clock**
use *time.clock()* (default on Windows)

-v, **--verbose**
打印原始计时结果；重复更多位数精度

-h, **--help**
打印一条简短的使用信息并退出

可以通过将每一行指定为单独的语句参数来给出多行语句；通过在引号中包含参数并使用前导空格可以缩进行。多个*-s* 选项的处理方式相似。

If *-n* is not given, a suitable number of loops is calculated by trying successive powers of 10 until the total time is at least 0.2 seconds.

default_timer() measurations can be affected by other programs running on the same machine, so the best thing to do when accurate timing is necessary is to repeat the timing a few times and use the best time. The *-r* option is good for this; the default of 3 repetitions is probably enough in most cases. On Unix, you can use *time.clock()* to measure CPU time.

注解: There is a certain baseline overhead associated with executing a pass statement. The code here doesn’t try to hide it, but you should be aware of it. The baseline overhead can be measured by invoking the program without arguments, and it might differ between Python versions. Also, to fairly compare older Python versions to Python 2.3, you may want to use Python’s `-O` option (see Optimizations) for the older versions to avoid timing `SET_LINENO` instructions.

26.6.4 例子

可以提供一个在开头只执行一次的 setup 语句:

```
$ python -m timeit -s 'text = "sample string"; char = "g"'  'char in text'
10000000 loops, best of 3: 0.0877 usec per loop
$ python -m timeit -s 'text = "sample string"; char = "g"'  'text.find(char)'
1000000 loops, best of 3: 0.342 usec per loop
```

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
```

使用 *Timer* 类及其方法可以完成同样的操作:

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40193588800002544, 0.3960157959998014, 0.39594301399984033]
```

以下示例显示如何计算包含多行的表达式。在这里我们对比使用 *hasattr()* 与 `try`/`except` 的开销来测试缺失与提供对象属性:

```
$ python -m timeit 'try:' '  str.__nonzero__' 'except AttributeError:' '  pass'
100000 loops, best of 3: 15.7 usec per loop
$ python -m timeit 'if hasattr(str, "__nonzero__"): pass'
100000 loops, best of 3: 4.26 usec per loop

$ python -m timeit 'try:' '  int.__nonzero__' 'except AttributeError:' '  pass'
1000000 loops, best of 3: 1.43 usec per loop
$ python -m timeit 'if hasattr(int, "__nonzero__"): pass'
100000 loops, best of 3: 2.23 usec per loop
```

```
>>> import timeit
>>> # attribute is missing
>>> s = """\
... try:
...     str.__nonzero__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__nonzero__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

要让 *timeit* 模块访问你定义的函数，你可以传递一个包含 import 语句的 *setup* 参数:

```
def test():
    """Stupid test function"""
    L = []
    for i in range(100):
        L.append(i)

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

26.7 trace —Trace or track Python statement execution

Source code: Lib/trace.py

The *trace* module allows you to trace program execution, generate annotated statement coverage listings, print caller/callee relationships and list functions executed during a program run. It can be used in another program or from the command line.

26.7.1 Command-Line Usage

The *trace* module can be invoked from the command line. It can be as simple as

```
python -m trace --count -C . somefile.py ...
```

The above will execute `somefile.py` and generate annotated listings of all Python modules imported during the execution into the current directory.

--help
Display usage and exit.

--version
Display the version of the module and exit.

Main options

At least one of the following options must be specified when invoking *trace*. The *--listfuncs* option is mutually exclusive with the *--trace* and *--count* options. When *--listfuncs* is provided, neither *--count* nor *--trace* are accepted, and vice versa.

-c, --count
Produce a set of annotated listing files upon program completion that shows how many times each statement was executed. See also *--coverdir*, *--file* and *--no-report* below.

-t, --trace
Display lines as they are executed.

-l, --listfuncs
Display the functions executed by running the program.

-r, --report
Produce an annotated list from an earlier program run that used the *--count* and *--file* option. This does not execute any code.

-T, --trackcalls
Display the calling relationships exposed by running the program.

Modifiers

-f, --file=<file>
Name of a file to accumulate counts over several tracing runs. Should be used with the *--count* option.

-C, --coverdir=<dir>
Directory where the report files go. The coverage report for `package.module` is written to file *dir/package/module*`.cover`.

-m, --missing
When generating annotated listings, mark lines which were not executed with >>>>>>.

-s, --summary
When using *--count* or *--report*, write a brief summary to stdout for each file processed.

-R, --no-report
Do not generate annotated listings. This is useful if you intend to make several runs with *--count*, and then produce a single set of annotated listings at the end.

-g, --timing
Prefix each line with the time since the program started. Only used while tracing.

Filters

These options may be repeated multiple times.

--ignore-module=<mod>
Ignore each of the given module names and its submodules (if it is a package). The argument can be a list of names separated by a comma.

--ignore-dir=<dir>
Ignore all modules and packages in the named directory and subdirectories. The argument can be a list of directories separated by *os.pathsep*.

26.7.2 编程接口

class trace.**Trace**([*count=1*[, *trace=1*[, *countfuncs=0*[, *countcallers=0*[, *ignoremods=()*[, *ignoredirs=()*[, *infile=None*[, *outfile=None*[, *timing=False*]]]]]]]]])
Create an object to trace execution of a single statement or expression. All parameters are optional. *count* enables counting of line numbers. *trace* enables line execution tracing. *countfuncs* enables listing of the functions called during the run. *countcallers* enables call relationship tracking. *ignoremods* is a list of modules or packages to ignore. *ignoredirs* is a list of directories whose modules or packages should be ignored. *infile* is the name of the file from which to read stored count information. *outfile* is the name of the file in which to write updated count information. *timing* enables a timestamp relative to when tracing was started to be displayed.

run(*cmd*)
Execute the command and gather statistics from the execution with the current tracing parameters. *cmd* must be a string or code object, suitable for passing into `exec()`.

runctx(*cmd*, *globals=None*, *locals=None*)
Execute the command and gather statistics from the execution with the current tracing parameters, in the defined global and local environments. If not defined, *globals* and *locals* default to empty dictionaries.

runfunc(*func*, **args*, ***kwds*)
Call *func* with the given arguments under control of the `Trace` object with the current tracing parameters.

results()
Return a `CoverageResults` object that contains the cumulative results of all previous calls to `run`, `runctx` and `runfunc` for the given `Trace` instance. Does not reset the accumulated trace results.

class trace.**CoverageResults**
A container for coverage results, created by `Trace.results()`. Should not be created directly by the user.

update(*other*)
Merge in data from another `CoverageResults` object.

write_results([*show_missing*=*True*[, *summary*=*False*[, *coverdir*=*None*]]])
Write coverage results. Set *show_missing* to show lines that had no hits. Set *summary* to include in the output the coverage summary per module. *coverdir* specifies the directory into which the coverage result files will be output. If `None`, the results for each source file are placed in its directory.

A simple example demonstrating the use of the programmatic interface:

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether to
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```

CHAPTER 27

软件打包和分发

这些库可帮助你发布和安装 Python 软件。虽然这些模块设计为与 'Python 包索引 <https://pypi.org>'__ 结合使用，但它们也可以与本地索引服务器一起使用，或者根本不使用任何索引服务器。

27.1 distutils —构建和安装 Python 模块

distutils 包为将待构建和安装的额外的模块，打包成 Python 安装包提供支持。新模块既可以是百分百的纯 Python，也可以是用 C 写的扩展模块，或者可以是一组包含了同时用 Python 和 C 编码的 Python 包。

大多数 Python 用户 不会想要直接使用这个包，而是使用 Python 包官方维护的跨版本工具。特别地，setuptools 是一个对于提供的*distutils* 的增强选项。

- 对声明项目依赖的支持。
- 额外的用于配置哪些文件包含在源代码发布中的装置（包括与版本控制系统集成需要的插件）
- 生成项目“进入点”的能力，进入点可用作应用插件系统的基础
- 自动在安装时间生成 Windows 命令行可执行文件的能力，而不是需要预编译它们
- 跨所有受支持的 Python 版本上的一致的表现

推荐的 pip 安装器用 `setuptools` 运行所有的 `setup.py` 脚本，即使脚本本身只引了 `distutils` 包。参考 Python Packaging User Guide 获得更多信息。

为了打包工具的作者和用户能更好理解当前的打包和分发系统，遗留的基于*distutils* 的用户文档和 API 参考保持可用：

- install-index
- distutils-index

27.2 ensurepip —Bootstrapping the pip installer

2.7.9 新版功能.

The *ensurepip* package provides support for bootstrapping the pip installer into an existing Python installation or virtual environment. This bootstrapping approach reflects the fact that pip is an independent project with its own release cycle, and the latest available stable version is bundled with maintenance and feature releases of the CPython reference interpreter.

In most cases, end users of Python shouldn't need to invoke this module directly (as pip should be bootstrapped by default), but it may be needed if installing pip was skipped when installing Python (or when creating a virtual environment) or after explicitly uninstalling pip.

注解: This module *does not* access the internet. All of the components needed to bootstrap pip are included as internal parts of the package.

参见:

installing-index The end user guide for installing Python packages

PEP 453: Explicit bootstrapping of pip in Python installations The original rationale and specification for this module.

PEP 477: Backport ensurepip (PEP 453) to Python 2.7 The rationale and specification for backporting PEP 453 to Python 2.7.

27.2.1 Command line interface

The command line interface is invoked using the interpreter's -m switch.

The simplest possible invocation is:

```
python -m ensurepip
```

This invocation will install pip if it is not already installed, but otherwise does nothing. To ensure the installed version of pip is at least as recent as the one bundled with ensurepip, pass the --upgrade option:

```
python -m ensurepip --upgrade
```

By default, pip is installed into the current virtual environment (if one is active) or into the system site packages (if there is no active virtual environment). The installation location can be controlled through two additional command line options:

- --root <dir>: Installs pip relative to the given root directory rather than the root of the currently active virtual environment (if any) or the default root for the current Python installation.
- --user: Installs pip into the user site packages directory rather than globally for the current Python installation (this option is not permitted inside an active virtual environment).

By default, the scripts pip, pipX, and pipX.Y will be installed (where X.Y stands for the version of Python used to invoke ensurepip). The scripts installed can be controlled through two additional command line options:

- --altinstall: if an alternate installation is requested, the pip and pipX script will *not* be installed.
- --no-default-pip: if a non-default installation is request, the pip script will *not* be installed.

在 2.7.15 版更改: The exit status is non-zero if the command fails.

27.2.2 Module API

ensurepip exposes two functions for programmatic use:

ensurepip.**version**()
Returns a string specifying the bundled version of pip that will be installed when bootstrapping an environment.

ensurepip.**bootstrap**(*root=None*, *upgrade=False*, *user=False*, *altinstall=False*, *default_pip=True*, *verbosity=0*)
Bootstraps pip into the current or designated environment.

root specifies an alternative root directory to install relative to. If *root* is None, then installation uses the default install location for the current environment.

upgrade indicates whether or not to upgrade an existing installation of an earlier version of pip to the bundled version.

user indicates whether to use the user scheme rather than installing globally.

By default, the scripts pip, pipX, and pipX.Y will be installed (where X.Y stands for the current version of Python).

If *altinstall* is set, then pip and pipX will *not* be installed.

If *default_pip* is set to False, then pip will *not* be installed.

Setting both *altinstall* and *default_pip* will trigger *ValueError*.

verbosity controls the level of output to *sys.stdout* from the bootstrapping operation.

注解： The bootstrapping process has side effects on both sys.path and os.environ. Invoking the command line interface in a subprocess instead allows these side effects to be avoided.

注解： The bootstrapping process may install additional modules required by pip, but other software should not assume those dependencies will always be present by default (as the dependencies may be removed in a future version of pip).

CHAPTER 28

Python 运行时服务

本章里描述的模块提供了和 Python 解释器及其环境交互相关的广泛服务。以下是综述：

28.1 sys —系统相关的参数和函数

该模块提供了一些变量和函数。这些变量可能被解释器使用，也可能由解释器提供。这些函数会影响解释器。本模块总是可用的。

sys.**argv**
一个列表，其中包含了被传递给 Python 脚本的命令行参数。argv[0] 为脚本的名称（是否是完整的路径名取决于操作系统）。如果是通过 Python 解释器的命令行参数 -c 来执行的，argv[0] 会被设置成字符串 '-c' 。如果没有脚本名被传递给 Python 解释器，argv[0] 为空字符串。

为了遍历标准输入，或者通过命令行传递的文件列表，参照 *fileinput* 模块

sys.**byteorder**
本地字节顺序的指示符。在大端序（最高有效位优先）操作系统上值为 'big' ，在小端序（最低有效位优先）操作系统上为 'little' 。

2.0 新版功能.

sys.**builtin_module_names**
一个元素为字符串的元组。包含了所有的被编译进 Python 解释器的模块。（这个信息无法通过其他的办法获取，modules.keys() 只包括被导入过的模块。）

sys.**call_tracing**(*func*, *args*)
在启用跟踪时调用 func(*args) 来保存跟踪状态，然后恢复跟踪状态。这将从检查点的调试器调用，以便递归地调试其他的一些代码。

sys.**copyright**
一个字符串，包含了 Python 解释器有关的版权信息

sys.**_clear_type_cache**()
清除内部的类型缓存。类型缓存是为了加速查找方法和属性的。在调试引用泄漏的时候调用这个函数只会清除不必要的引用。

这个函数应该只在内部为了一些特定的目的使用。

2.6 新版功能.

sys.**_current_frames**()
返回一个字典，将每个线程的标识符映射到调用函数时该线程中当前活动的最顶层堆栈帧。注意*traceback* 模块中的函数可以在给定帧的情况下构建调用堆栈。

这对于调试死锁最有用：本函数不需要死锁线程的配合，并且只要这些线程的调用栈保持死锁，它们就是冻结的。在调用本代码来检查栈顶的帧的那一刻，非死锁线程返回的帧可能与该线程当前活动的帧没有任何关系。

这个函数应该只在内部为了一些特定的目的使用。

2.5 新版功能.

sys.**dllhandle**
Integer specifying the handle of the Python DLL. Availability: Windows.

sys.**displayhook**(*value*)
If *value* is not `None`, this function prints it to `sys.stdout`, and saves it in `__builtin__._`.

`sys.displayhook` is called on the result of evaluating an *expression* entered in an interactive Python session. The display of these values can be customized by assigning another one-argument function to `sys.displayhook`.

sys.**dont_write_bytecode**
If this is true, Python won't try to write `.pyc` or `.pyo` files on the import of source modules. This value is initially set to `True` or `False` depending on the `-B` command line option and the `PYTHONDONTWRITEBYTECODE` environment variable, but you can set it yourself to control bytecode file generation.

2.6 新版功能.

sys.**excepthook**(*type*, *value*, *traceback*)
This function prints out a given traceback and exception to `sys.stderr`.

When an exception is raised and uncaught, the interpreter calls `sys.excepthook` with three arguments, the exception class, exception instance, and a traceback object. In an interactive session this happens just before control is returned to the prompt; in a Python program this happens just before the program exits. The handling of such top-level exceptions can be customized by assigning another three-argument function to `sys.excepthook`.

sys.**__displayhook__**
sys.**__excepthook__**
These objects contain the original values of `displayhook` and `excepthook` at the start of the program. They are saved so that `displayhook` and `excepthook` can be restored in case they happen to get replaced with broken objects.

sys.**exc_info**()
This function returns a tuple of three values that give information about the exception that is currently being handled. The information returned is specific both to the current thread and to the current stack frame. If the current stack frame is not handling an exception, the information is taken from the calling stack frame, or its caller, and so on until a stack frame is found that is handling an exception. Here, "handling an exception" is defined as "executing or having executed an except clause." For any stack frame, only information about the most recently handled exception is accessible.

If no exception is being handled anywhere on the stack, a tuple containing three `None` values is returned. Otherwise, the values returned are `(type, value, traceback)`. Their meaning is: *type* gets the exception type of the exception being handled (a class object); *value* gets the exception parameter (its *associated value* or the second argument to `raise`, which is always a class instance if the exception type is a class object); *traceback* gets a traceback object (see the Reference Manual) which encapsulates the call stack at the point where the exception originally occurred.

If *exc_clear()* is called, this function will return three None values until either another exception is raised in the current thread or the execution stack returns to a frame where another exception is being handled.

警告: Assigning the *traceback* return value to a local variable in a function that is handling an exception will cause a circular reference. This will prevent anything referenced by a local variable in the same function or by the traceback from being garbage collected. Since most functions don't need access to the traceback, the best solution is to use something like `exctype, value = sys.exc_info()[:2]` to extract only the exception type and value. If you do need the traceback, make sure to delete it after use (best done with a `try ... finally` statement) or to call *exc_info()* in a function that does not itself handle an exception.

注解: Beginning with Python 2.2, such cycles are automatically reclaimed when garbage collection is enabled and they become unreachable, but it remains more efficient to avoid creating cycles.

sys.**exc_clear**()
This function clears all information relating to the current or last exception that occurred in the current thread. After calling this function, *exc_info()* will return three None values until another exception is raised in the current thread or the execution stack returns to a frame where another exception is being handled.

This function is only needed in only a few obscure situations. These include logging and error handling systems that report information on the last or current exception. This function can also be used to try to free resources and trigger object finalization, though no guarantee is made as to what objects will be freed, if any.

2.3 新版功能.

sys.**exc_type**
sys.**exc_value**
sys.**exc_traceback**
1.5 版后已移除: Use *exc_info()* instead.

Since they are global variables, they are not specific to the current thread, so their use is not safe in a multi-threaded program. When no exception is being handled, exc_type is set to None and the other two are undefined.

sys.**exec_prefix**
A string giving the site-specific directory prefix where the platform-dependent Python files are installed; by default, this is also '/usr/local'. This can be set at build time with the --exec-prefix argument to the **configure** script. Specifically, all configuration files (e.g. the pyconfig.h header file) are installed in the directory *exec_prefix*/lib/python*X.Y*/config, and shared library modules are installed in *exec_prefix*/lib/python*X.Y*/lib-dynload, where *X.Y* is the version number of Python, for example 2.7.

sys.**executable**
A string giving the absolute path of the executable binary for the Python interpreter, on systems where this makes sense. If Python is unable to retrieve the real path to its executable, *sys.executable* will be an empty string or None.

sys.**exit**([*arg*])
从 Python 中退出。实现方式是抛出一个*SystemExit* 异常。异常抛出后 try 声明的 finally 分支语句的清除动作将被出发。此动作有可能打断更外层的退出尝试。

The optional argument *arg* can be an integer giving the exit status (defaulting to zero), or another type of object. If it is an integer, zero is considered "successful termination" and any nonzero value is considered "abnormal termination" by shells and the like. Most systems require it to be in the range 0–127, and produce undefined results otherwise. Some systems have a convention for assigning specific meanings to specific exit codes, but these are generally underdeveloped; Unix programs generally use 2 for command line syntax errors and 1 for all other kind of errors. If another type of object is passed, None is equivalent to passing zero, and any other object is printed

to *stderr* and results in an exit code of 1. In particular, `sys.exit("some error message")` is a quick way to exit a program when an error occurs.

由于*exit()* 最终“只是”抛出一个异常，因此当从主线程调用时，只会从进程退出；而异常不会因此被打断。

`sys.`**exitfunc**

This value is not actually defined by the module, but can be set by the user (or by a program) to specify a clean-up action at program exit. When set, it should be a parameterless function. This function will be called when the interpreter exits. Only one function may be installed in this way; to allow multiple functions which will be called at termination, use the *atexit* module.

注解： The exit function is not called when the program is killed by a signal, when a Python fatal internal error is detected, or when `os._exit()` is called.

2.4 版后已移除: Use *atexit* instead.

`sys.`**flags**

The struct sequence *flags* exposes the status of command line flags. The attributes are read only.

attribute –属性	标志
debug	-d
py3k_warning	-3
division_warning	-Q
division_new	-Qnew
inspect	-i
interactive	-i
optimize	-O 或 -OO
dont_write_bytecode	-B
no_user_site	-s
no_site	-S
ignore_environment	-E
tabcheck	-t or -tt
verbose	-v
unicode	-U
bytes_warning	-b
hash_randomization	-R

2.6 新版功能.

2.7.3 新版功能: `hash_randomization` 属性

`sys.`**float_info**

A structseq holding information about the float type. It contains low level information about the precision and internal representation. The values correspond to the various floating-point constants defined in the standard header file `float.h` for the ‘C’ programming language; see section 5.2.4.2.2 of the 1999 ISO/IEC C standard [C99], ‘Characteristics of floating types’, for details.

attribute –属性	float.h 宏	解释
epsilon	DBL_EPSILON	difference between 1 and the least value greater than 1 that is representable as a float
dig	DBL_DIG	maximum number of decimal digits that can be faithfully represented in a float; see below
mant_dig	DBL_MANT_DIG	float precision: the number of base-radix digits in the significand of a float
max	DBL_MAX	maximum representable finite float
max_exp	DBL_MAX_EXP	maximum integer e such that radix**(e-1) is a representable finite float
max_10_exp	DBL_MAX_10_EXP	maximum integer e such that 10**e is in the range of representable finite floats
min	DBL_MIN	minimum positive normalized float
min_exp	DBL_MIN_EXP	minimum integer e such that radix**(e-1) is a normalized float
min_10_exp	DBL_MIN_10_EXP	minimum integer e such that 10**e is a normalized float
radix	FLT_RADIX	radix of exponent representation
rounds	FLT_ROUNDS	integer constant representing the rounding mode used for arithmetic operations. This reflects the value of the system FLT_ROUNDS macro at interpreter startup time. See section 5.2.4.2.2 of the C99 standard for an explanation of the possible values and their meanings.

The attribute sys.float_info.dig needs further explanation. If s is any string representing a decimal number with at most sys.float_info.dig significant digits, then converting s to a float and back again will recover a string representing the same decimal value:

```
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'    # decimal string with 15 significant digits
>>> format(float(s), '.15g')  # convert to float and back -> same value
'3.14159265358979'
```

But for strings with more than sys.float_info.dig significant digits, this isn't always true:

```
>>> s = '9876543211234567'    # 16 significant digits is too many!
>>> format(float(s), '.16g')  # conversion changes value
'9876543211234568'
```

2.6 新版功能.

sys.**float_repr_style**

A string indicating how the *repr()* function behaves for floats. If the string has value 'short' then for a finite float x, repr(x) aims to produce a short string with the property that float(repr(x)) == x. This is the usual behaviour in Python 2.7 and later. Otherwise, float_repr_style has value 'legacy' and repr(x) behaves in the same way as it did in versions of Python prior to 2.7.

2.7 新版功能.

sys.**getcheckinterval**()

Return the interpreter's "check interval"; see *setcheckinterval()*.

2.3 新版功能.

sys.**getdefaultencoding**()
Return the name of the current default string encoding used by the Unicode implementation.

2.0 新版功能.

sys.**getdlopenflags**()
Return the current value of the flags that are used for `dlopen()` calls. The flag constants are defined in the *dl* and `DLFCN` modules. Availability: Unix.

2.2 新版功能.

sys.**getfilesystemencoding**()
Return the name of the encoding used to convert Unicode filenames into system file names, or `None` if the system default encoding is used. The result value depends on the operating system:

- On Mac OS X, the encoding is `'utf-8'`.
- On Unix, the encoding is the user's preference according to the result of nl_langinfo(CODESET), or `None` if the `nl_langinfo(CODESET)` failed.
- On Windows NT+, file names are Unicode natively, so no conversion is performed. *getfilesystemencoding()* still returns `'mbcs'`, as this is the encoding that applications should use when they explicitly want to convert Unicode strings to byte strings that are equivalent when used as file names.
- On Windows 9x, the encoding is `'mbcs'`.

2.3 新版功能.

sys.**getrefcount**(*object*)
Return the reference count of the *object*. The count returned is generally one higher than you might expect, because it includes the (temporary) reference as an argument to *getrefcount()*.

sys.**getrecursionlimit**()
Return the current value of the recursion limit, the maximum depth of the Python interpreter stack. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python. It can be set by *setrecursionlimit()*.

sys.**getsizeof**(*object*[, *default*])
Return the size of an object in bytes. The object can be any type of object. All built-in objects will return correct results, but this does not have to hold true for third-party extensions as it is implementation specific.

If given, *default* will be returned if the object does not provide means to retrieve the size. Otherwise a *TypeError* will be raised.

getsizeof() calls the object's `__sizeof__` method and adds an additional garbage collector overhead if the object is managed by the garbage collector.

2.6 新版功能.

sys.**_getframe**([*depth*])
Return a frame object from the call stack. If optional integer *depth* is given, return the frame object that many calls below the top of the stack. If that is deeper than the call stack, *ValueError* is raised. The default for *depth* is zero, returning the frame at the top of the call stack.

CPython implementation detail: This function should be used for internal and specialized purposes only. It is not guaranteed to exist in all implementations of Python.

sys.**getprofile**()
Get the profiler function as set by *setprofile()*.

2.6 新版功能.

sys.**gettrace**()
Get the trace function as set by *settrace()*.

CPython implementation detail: The *gettrace()* function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

2.6 新版功能.

sys.**getwindowsversion**()
Return a named tuple describing the Windows version currently running. The named elements are *major*, *minor*, *build*, *platform*, *service_pack*, *service_pack_minor*, *service_pack_major*, *suite_mask*, and *product_type*. *service_pack* contains a string while all other values are integers. The components can also be accessed by name, so `sys.getwindowsversion()[0]` is equivalent to `sys.getwindowsversion().major`. For compatibility with prior versions, only the first 5 elements are retrievable by indexing.

platform may be one of the following values:

常数	Platform
0 (VER_PLATFORM_WIN32s)	Win32s on Windows 3.1
1 (VER_PLATFORM_WIN32_WINDOWS)	Windows 95/98/ME
2 (VER_PLATFORM_WIN32_NT)	Windows NT/2000/XP/x64
3 (VER_PLATFORM_WIN32_CE)	Windows CE

product_type may be one of the following values:

常数	含义
1 (VER_NT_WORKSTATION)	系统是工作站。
2 (VER_NT_DOMAIN_CONTROLLER)	系统是域控制器。
3 (VER_NT_SERVER)	系统是服务器，但不是域控制器。

This function wraps the Win32 `GetVersionEx()` function; see the Microsoft documentation on `OSVERSIONINFOEX()` for more information about these fields.

Availability: Windows.

2.3 新版功能.

在 2.7 版更改: Changed to a named tuple and added *service_pack_minor*, *service_pack_major*, *suite_mask*, and *product_type*.

sys.**hexversion**
The version number encoded as a single integer. This is guaranteed to increase with each version, including proper support for non-production releases. For example, to test that the Python interpreter is at least version 1.5.2, use:

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

This is called `hexversion` since it only really looks meaningful when viewed as the result of passing it to the built-in *hex()* function. The `version_info` value may be used for a more human-friendly encoding of the same information.

The `hexversion` is a 32-bit number with the following layout:

Bits (big endian order)	含义
1-8	PY_MAJOR_VERSION (the 2 in 2.1.0a3)
9-16	PY_MINOR_VERSION (the 1 in 2.1.0a3)
17-24	PY_MICRO_VERSION (the 0 in 2.1.0a3)
25-28	PY_RELEASE_LEVEL (0xA for alpha, 0xB for beta, 0xC for release candidate and 0xF for final)
29-32	PY_RELEASE_SERIAL (the 3 in 2.1.0a3, zero for final releases)

Thus 2.1.0a3 is hexversion 0x020100a3.

1.5.2 新版功能.

sys.**long_info**
A struct sequence that holds information about Python's internal representation of integers. The attributes are read only.

属性	解释
bits_per_digit	number of bits held in each digit. Python integers are stored internally in base 2**long_info.bits_per_digit
sizeof_digit	用于表示数字的 C 类型的字节大小

2.7 新版功能.

sys.**last_type**
sys.**last_value**
sys.**last_traceback**
These three variables are not always defined; they are set when an exception is not handled and the interpreter prints an error message and a stack traceback. Their intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error. (Typical use is import pdb; pdb.pm() to enter the post-mortem debugger; see chapter *pdb —Python 的调试器* for more information.)

The meaning of the variables is the same as that of the return values from *exc_info()* above. (Since there is only one interactive thread, thread-safety is not a concern for these variables, unlike for exc_type etc.)

sys.**maxint**
The largest positive integer supported by Python's regular integer type. This is at least 2**31-1. The largest negative integer is -maxint-1 —the asymmetry results from the use of 2's complement binary arithmetic.

sys.**maxsize**
The largest positive integer supported by the platform's Py_ssize_t type, and thus the maximum size lists, strings, dicts, and many other containers can have.

sys.**maxunicode**
An integer giving the largest supported code point for a Unicode character. The value of this depends on the configuration option that specifies whether Unicode characters are stored as UCS-2 or UCS-4.

sys.**meta_path**
A list of *finder* objects that have their find_module() methods called to see if one of the objects can find the module to be imported. The find_module() method is called at least with the absolute name of the module being imported. If the module to be imported is contained in package then the parent package's __path__ attribute is passed in as a second argument. The method returns None if the module cannot be found, else returns a *loader*.

sys.meta_path is searched before any implicit default finders or *sys.path*.

See **PEP 302** for the original specification.

sys.**modules**
This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. Note that removing a module from this dictionary is *not* the same as calling *reload()* on the corresponding module object.

sys.**path**
A list of strings that specifies the search path for modules. Initialized from the environment variable PYTHONPATH, plus an installation-dependent default.

As initialized upon program startup, the first item of this list, path[0], is the directory containing the script that was used to invoke the Python interpreter. If the script directory is not available (e.g. if the interpreter is invoked interactively or if the script is read from standard input), path[0] is the empty string, which directs Python to search modules in the current directory first. Notice that the script directory is inserted *before* the entries inserted as a result of PYTHONPATH.

A program is free to modify this list for its own purposes.

在 2.3 版更改: Unicode strings are no longer ignored.

参见:

Module *site* This describes how to use .pth files to extend *sys.path*.

sys.**path_hooks**
A list of callables that take a path argument to try to create a *finder* for the path. If a finder can be created, it is to be returned by the callable, else raise *ImportError*.

Originally specified in **PEP 302**.

sys.**path_importer_cache**
A dictionary acting as a cache for *finder* objects. The keys are paths that have been passed to *sys.path_hooks* and the values are the finders that are found. If a path is a valid file system path but no explicit finder is found on *sys.path_hooks* then None is stored to represent the implicit default finder should be used. If the path is not an existing path then *imp.NullImporter* is set.

Originally specified in **PEP 302**.

sys.**platform**
This string contains a platform identifier that can be used to append platform-specific components to *sys.path*, for instance.

For most Unix systems, this is the lowercased OS name as returned by uname -s with the first part of the version as returned by uname -r appended, e.g. 'sunos5', *at the time when Python was built*. Unless you want to test for a specific system version, it is therefore recommended to use the following idiom:

```
if sys.platform.startswith('freebsd'):
    # FreeBSD-specific code here...
elif sys.platform.startswith('linux'):
    # Linux-specific code here...
```

在 2.7.3 版更改: Since lots of code check for sys.platform == 'linux2', and there is no essential change between Linux 2.x and 3.x, sys.platform is always set to 'linux2', even on Linux 3.x. In Python 3.3 and later, the value will always be set to 'linux', so it is recommended to always use the startswith idiom presented above.

对于其他系统，值是:

系统	*platform* value
Linux (2.x *and* 3.x)	`'linux2'`
Windows	`'win32'`
Windows/Cygwin	`'cygwin'`
Mac OS X	`'darwin'`
OS/2	`'os2'`
OS/2 EMX	`'os2emx'`
RiscOS	`'riscos'`
AtheOS	`'atheos'`

参见:

os.name has a coarser granularity. *os.uname()* gives system-dependent version information.

platform 模块对系统的标识有更详细的检查。

sys.**prefix**
A string giving the site-specific directory prefix where the platform independent Python files are installed; by default, this is the string `'/usr/local'`. This can be set at build time with the `--prefix` argument to the **configure** script. The main collection of Python library modules is installed in the directory *prefix*`/lib/python`*X.Y* while the platform independent header files (all except `pyconfig.h`) are stored in *prefix*`/include/python`*X.Y*, where *X.Y* is the version number of Python, for example `2.7`.

sys.**ps1**
sys.**ps2**
Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the interpreter is in interactive mode. Their initial values in this case are `'>>> '` and `'... '`. If a non-string object is assigned to either variable, its *str()* is re-evaluated each time the interpreter prepares to read a new interactive command; this can be used to implement a dynamic prompt.

sys.**py3kwarning**
Bool containing the status of the Python 3 warning flag. It's `True` when Python is started with the -3 option. (This should be considered read-only; setting it to a different value doesn't have an effect on Python 3 warnings.)

2.6 新版功能.

sys.**setcheckinterval**(*interval*)
Set the interpreter's "check interval". This integer value determines how often the interpreter checks for periodic things such as thread switches and signal handlers. The default is `100`, meaning the check is performed every 100 Python virtual instructions. Setting it to a larger value may increase performance for programs using threads. Setting it to a value <= 0 checks every virtual instruction, maximizing responsiveness as well as overhead.

sys.**setdefaultencoding**(*name*)
Set the current default string encoding used by the Unicode implementation. If *name* does not match any available encoding, *LookupError* is raised. This function is only intended to be used by the *site* module implementation and, where needed, by `sitecustomize`. Once used by the *site* module, it is removed from the *sys* module's namespace.

2.0 新版功能.

sys.**setdlopenflags**(*n*)
Set the flags used by the interpreter for `dlopen()` calls, such as when the interpreter loads extension modules. Among other things, this will enable a lazy resolving of symbols when importing a module, if called as `sys.setdlopenflags(0)`. To share symbols across extension modules, call as `sys.setdlopenflags(dl.RTLD_NOW | dl.RTLD_GLOBAL)`. Symbolic names for the flag modules can be either found in the *dl* module, or in the `DLFCN` module. If `DLFCN` is not available, it can be generated from `/usr/include/dlfcn.h` using the **h2py** script. Availability: Unix.

2.2 新版功能.

sys.**setprofile**(*profilefunc*)
Set the system's profile function, which allows you to implement a Python source code profiler in Python. See chapter *Python 分析器* for more information on the Python profiler. The system's profile function is called similarly to the system's trace function (see *settrace()*), but it is called with different events, for example it isn't called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set). The function is thread-specific, but there is no way for the profiler to know about context switches between threads, so it does not make sense to use this in the presence of multiple threads. Also, its return value is not used, so it can simply return `None`.

Profile functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: `'call'`, `'return'`, `'c_call'`, `'c_return'`, or `'c_exception'`. *arg* depends on the event type.

这些事件具有以下含义:

'call' A function is called (or some other code block entered). The profile function is called; *arg* is `None`.

'return' A function (or other code block) is about to return. The profile function is called; *arg* is the value that will be returned, or `None` if the event is caused by an exception being raised.

'c_call' A C function is about to be called. This may be an extension function or a built-in. *arg* is the C function object.

'c_return' A C function has returned. *arg* is the C function object.

'c_exception' A C function has raised an exception. *arg* is the C function object.

sys.**setrecursionlimit**(*limit*)
Set the maximum depth of the Python interpreter stack to *limit*. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.

The highest possible limit is platform-dependent. A user may need to set the limit higher when she has a program that requires deep recursion and a platform that supports a higher limit. This should be done with care, because a too-high limit can lead to a crash.

sys.**settrace**(*tracefunc*)
Set the system's trace function, which allows you to implement a Python source code debugger in Python. The function is thread-specific; for a debugger to support multiple threads, it must be registered using *settrace()* for each thread being debugged.

Trace functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: `'call'`, `'line'`, `'return'` or `'exception'`. *arg* depends on the event type.

The trace function is invoked (with *event* set to `'call'`) whenever a new local scope is entered; it should return a reference to a local trace function to be used that scope, or `None` if the scope shouldn't be traced.

The local trace function should return a reference to itself (or to another function for further tracing in that scope), or `None` to turn off tracing in that scope.

这些事件具有以下含义:

'call' A function is called (or some other code block entered). The global trace function is called; *arg* is `None`; the return value specifies the local trace function.

'line' The interpreter is about to execute a new line of code or re-execute the condition of a loop. The local trace function is called; *arg* is `None`; the return value specifies the new local trace function. See `Objects/lnotab_notes.txt` for a detailed explanation of how this works.

'return' A function (or other code block) is about to return. The local trace function is called; *arg* is the value that will be returned, or `None` if the event is caused by an exception being raised. The trace function's return value is ignored.

'exception' An exception has occurred. The local trace function is called; *arg* is a tuple `(exception, value, traceback)`; the return value specifies the new local trace function.

Note that as an exception is propagated down the chain of callers, an `'exception'` event is generated at each level.

For more information on code and frame objects, refer to types.

CPython implementation detail: The *settrace()* function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

sys.**settscdump**(*on_flag*)
Activate dumping of VM measurements using the Pentium timestamp counter, if *on_flag* is true. Deactivate these dumps if *on_flag* is off. The function is available only if Python was compiled with `--with-tsc`. To understand the output of this dump, read `Python/ceval.c` in the Python sources.

2.4 新版功能.

CPython implementation detail: This function is intimately bound to CPython implementation details and thus not likely to be implemented elsewhere.

sys.**stdin**
sys.**stdout**
sys.**stderr**
File objects corresponding to the interpreter's standard input, output and error streams. `stdin` is used for all interpreter input except for scripts but including calls to *input()* and *raw_input()*. `stdout` is used for the output of `print` and *expression* statements and for the prompts of *input()* and *raw_input()*. The interpreter's own prompts and (almost all of) its error messages go to `stderr`. `stdout` and `stderr` needn't be built-in file objects: any object is acceptable as long as it has a `write()` method that takes a string argument. (Changing these objects doesn't affect the standard I/O streams of processes executed by *os.popen()*, *os.system()* or the `exec*()` family of functions in the *os* module.)

sys.**__stdin__**
sys.**__stdout__**
sys.**__stderr__**
These objects contain the original values of `stdin`, `stderr` and `stdout` at the start of the program. They are used during finalization, and could be useful to print to the actual standard stream no matter if the `sys.std*` object has been redirected.

It can also be used to restore the actual files to known working file objects in case they have been overwritten with a broken object. However, the preferred way to do this is to explicitly save the previous stream before replacing it, and restore the saved object.

sys.**subversion**
A triple (repo, branch, version) representing the Subversion information of the Python interpreter. *repo* is the name of the repository, `'CPython'`. *branch* is a string of one of the forms `'trunk'`, `'branches/name'` or `'tags/name'`. *version* is the output of `svnversion`, if the interpreter was built from a Subversion checkout; it contains the revision number (range) and possibly a trailing 'M' if there were local modifications. If the tree was exported (or svnversion was not available), it is the revision of `Include/patchlevel.h` if the branch is a tag. Otherwise, it is `None`.

2.5 新版功能.

注解: Python is now developed using Git. In recent Python 2.7 bugfix releases, *subversion* therefore contains placeholder information. It is removed in Python 3.3.

---

sys.**tracebacklimit**
When this variable is set to an integer value, it determines the maximum number of levels of traceback information printed when an unhandled exception occurs. The default is `1000`. When set to `0` or less, all traceback information is suppressed and only the exception type and value are printed.

sys.**version**
A string containing the version number of the Python interpreter plus additional information on the build number and compiler used. This string is displayed when the interactive interpreter is started. Do not extract version information out of it, rather, use *version_info* and the functions provided by the *platform* module.

sys.**api_version**
The C API version for this interpreter. Programmers may find this useful when debugging version conflicts between Python and extension modules.

2.3 新版功能.

sys.**version_info**
A tuple containing the five components of the version number: *major*, *minor*, *micro*, *releaselevel*, and *serial*. All values except *releaselevel* are integers; the release level is `'alpha'`, `'beta'`, `'candidate'`, or `'final'`. The `version_info` value corresponding to the Python version 2.0 is `(2, 0, 0, 'final', 0)`. The components can also be accessed by name, so `sys.version_info[0]` is equivalent to `sys.version_info.major` and so on.

2.0 新版功能.

在 2.7 版更改: Added named component attributes

sys.**warnoptions**
This is an implementation detail of the warnings framework; do not modify this value. Refer to the *warnings* module for more information on the warnings framework.

sys.**winver**
The version number used to form registry keys on Windows platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the first three characters of *version*. It is provided in the *sys* module for informational purposes; modifying this value has no effect on the registry keys used by Python. Availability: Windows.

Citations

28.2 sysconfig —Provide access to Python's configuration information

2.7 新版功能.

源代码： Lib/sysconfig.py

The *sysconfig* module provides access to Python's configuration information like the list of installation paths and the configuration variables relevant for the current platform.

28.2.1 配置变量

A Python distribution contains a `Makefile` and a `pyconfig.h` header file that are necessary to build both the Python binary itself and third-party C extensions compiled using *distutils*.

sysconfig puts all variables found in these files in a dictionary that can be accessed using *get_config_vars()* or *get_config_var()*.

Notice that on Windows, it's a much smaller set.

`sysconfig.`**`get_config_vars`**`(`*`*args`*`)`
With no arguments, return a dictionary of all configuration variables relevant for the current platform.

With arguments, return a list of values that result from looking up each argument in the configuration variable dictionary.

For each argument, if the value is not found, return `None`.

`sysconfig.`**`get_config_var`**`(`*`name`*`)`
Return the value of a single variable *name*. Equivalent to `get_config_vars().get(name)`.

If *name* is not found, return `None`.

用法示例:

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```

28.2.2 安装路径

Python uses an installation scheme that differs depending on the platform and on the installation options. These schemes are stored in *sysconfig* under unique identifiers based on the value returned by *os.name*.

Every new component that is installed using *distutils* or a Distutils-based system will follow the same scheme to copy its file in the right places.

Python currently supports seven schemes:

- *posix_prefix*: scheme for Posix platforms like Linux or Mac OS X. This is the default scheme used when Python or a component is installed.
- *posix_home*: scheme for Posix platforms used when a *home* option is used upon installation. This scheme is used when a component is installed through Distutils with a specific home prefix.
- *posix_user*: scheme for Posix platforms used when a component is installed through Distutils and the *user* option is used. This scheme defines paths located under the user home directory.
- *nt*: scheme for NT platforms like Windows.
- *nt_user*: scheme for NT platforms, when the *user* option is used.
- *os2*: scheme for OS/2 platforms.
- *os2_home*: scheme for OS/2 platforms, when the *user* option is used.

Each scheme is itself composed of a series of paths and each path has a unique identifier. Python currently uses eight paths:

- *stdlib*: directory containing the standard Python library files that are not platform-specific.
- *platstdlib*: directory containing the standard Python library files that are platform-specific.
- *platlib*: directory for site-specific, platform-specific files.
- *purelib*: directory for site-specific, non-platform-specific files.
- *include*: directory for non-platform-specific header files.
- *platinclude*: directory for platform-specific header files.
- *scripts*: directory for script files.
- *data*: directory for data files.

sysconfig provides some functions to determine these paths.

sysconfig.**get_scheme_names**()
Return a tuple containing all schemes currently supported in *sysconfig*.

sysconfig.**get_path_names**()
Return a tuple containing all path names currently supported in *sysconfig*.

sysconfig.**get_path**(*name*[, *scheme*[, *vars*[, *expand*]]])
Return an installation path corresponding to the path *name*, from the install scheme named *scheme*.

name has to be a value from the list returned by *get_path_names()*.

sysconfig stores installation paths corresponding to each path name, for each platform, with variables to be expanded. For instance the *stdlib* path for the *nt* scheme is: `{base}/Lib`.

get_path() will use the variables returned by *get_config_vars()* to expand the path. All variables have default values for each platform so one may call this function and get the default value.

If *scheme* is provided, it must be a value from the list returned by *get_scheme_names()*. Otherwise, the default scheme for the current platform is used.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary return by *get_config_vars()*.

If *expand* is set to `False`, the path will not be expanded using the variables.

If *name* is not found, return `None`.

sysconfig.**get_paths**([*scheme*[, *vars*[, *expand*]]])
Return a dictionary containing all installation paths corresponding to an installation scheme. See *get_path()* for more information.

If *scheme* is not provided, will use the default scheme for the current platform.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary used to expand the paths.

If *expand* is set to false, the paths will not be expanded.

If *scheme* is not an existing scheme, *get_paths()* will raise a *KeyError*.

28.2.3 其他功能

sysconfig.**get_python_version**()
Return the MAJOR.MINOR Python version number as a string. Similar to sys.version[:3].

sysconfig.**get_platform**()
Return a string that identifies the current platform.

This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by *os.uname()*), although the exact information included depends on the OS; e.g. for IRIX the architecture isn't particularly important (IRIX only runs on SGI hardware), but for Linux the kernel version isn't particularly important.

返回值的示例:

- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u
- irix-5.3
- irix64-6.2

Windows 将返回以下之一:

- win-amd64 (在 AMD64, aka x86_64, Intel64, 和 EM64T 上的 64 位 Windows)
- win-ia64 (64bit Windows on Itanium)
- win32 (all others - specifically, sys.platform is returned)

Mac OS X 返回:

- macosx-10.6-ppc
- macosx-10.4-ppc64
- macosx-10.3-i386
- macosx-10.4-fat

对于其他非-POSIX 平台，目前只是返回*sys.platform* 。

sysconfig.**is_python_build**()
Return True if the current Python installation was built from source.

sysconfig.**parse_config_h**(*fp*[, *vars*])
Parse a config.h-style file.

fp is a file-like object pointing to the config.h-like file.

A dictionary containing name/value pairs is returned. If an optional dictionary is passed in as the second argument, it is used instead of a new dictionary, and updated with the values read in the file.

sysconfig.**get_config_h_filename**()
返回 pyconfig.h 的目录

sysconfig.**get_makefile_filename**()
返回 Makefile 的目录

28.3 __builtin__ —Built-in objects

This module provides direct access to all 'built-in' identifiers of Python; for example, `__builtin__.open` is the full name for the built-in function *open()*. See 内置函数 and 内置常量 for documentation.

This module is not normally accessed explicitly by most applications, but can be useful in modules that provide objects with the same name as a built-in value, but in which the built-in of that name is also needed. For example, in a module that wants to implement an *open()* function that wraps the built-in *open()*, this module can be used directly:

```
import __builtin__

def open(path):
    f = __builtin__.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to upper-case.'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ...
```

CPython implementation detail: Most modules have the name `__builtins__` (note the `'s'`) made available as part of their globals. The value of `__builtins__` is normally either this module or the value of this modules' s *__dict__* attribute. Since this is an implementation detail, it may not be used by alternate implementations of Python.

28.4 future_builtins —Python 3 builtins

2.6 新版功能.

This module provides functions that exist in 2.x, but have different behavior in Python 3, so they cannot be put into the 2.x builtins namespace.

Instead, if you want to write code compatible with Python 3 builtins, import them from this module, like this:

```
from future_builtins import map, filter

... code using Python 3-style map and filter ...
```

The *2to3* tool that ports Python 2 code to Python 3 will recognize this usage and leave the new builtins alone.

注解: The Python 3 *print()* function is already in the builtins, but cannot be accessed from Python 2 code unless you use the appropriate future statement:

```
from __future__ import print_function
```

Available builtins are:

`future_builtins.`**ascii**(*object*)
Returns the same as *repr()*. In Python 3, *repr()* will return printable Unicode characters unescaped, while

ascii() will always backslash-escape them. Using *future_builtins.ascii()* instead of *repr()* in 2.6 code makes it clear that you need a pure ASCII return value.

future_builtins.**filter**(*function*, *iterable*)
Works like *itertools.ifilter()*.

future_builtins.**hex**(*object*)
Works like the built-in *hex()*, but instead of __hex__() it will use the __index__() method on its argument to get an integer that is then converted to hexadecimal.

future_builtins.**map**(*function*, *iterable*, ...)
Works like *itertools.imap()*.

注解： In Python 3, *map()* does not accept None for the function argument.

future_builtins.**oct**(*object*)
Works like the built-in *oct()*, but instead of __oct__() it will use the __index__() method on its argument to get an integer that is then converted to octal.

future_builtins.**zip**(**iterables*)
Works like *itertools.izip()*.

28.5 __main__ —顶层脚本环境

This module represents the (otherwise anonymous) scope in which the interpreter's main program executes —commands read either from standard input, from a script file, or from an interactive prompt. It is this environment in which the idiomatic "conditional script" stanza causes a script to run:

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

28.6 warnings —Warning control

2.1 新版功能.

源代码： Lib/warnings.py

Warning messages are typically issued in situations where it is useful to alert the user of some condition in a program, where that condition (normally) doesn't warrant raising an exception and terminating the program. For example, one might want to issue a warning when a program uses an obsolete module.

Python programmers issue warnings by calling the *warn()* function defined in this module. (C programmers use PyErr_WarnEx(); see exceptionhandling for details).

Warning messages are normally written to sys.stderr, but their disposition can be changed flexibly, from ignoring all warnings to turning them into exceptions. The disposition of warnings can vary based on the warning category (see below), the text of the warning message, and the source location where it is issued. Repetitions of a particular warning for the same source location are typically suppressed.

There are two stages in warning control: first, each time a warning is issued, a determination is made whether a message should be issued or not; next, if a message is to be issued, it is formatted and printed using a user-settable hook.

The determination whether to issue a warning message is controlled by the warning filter, which is a sequence of matching rules and actions. Rules can be added to the filter by calling *filterwarnings()* and reset to its default state by calling *resetwarnings()*.

The printing of warning messages is done by calling *showwarning()*, which may be overridden; the default implementation of this function formats the message by calling *formatwarning()*, which is also available for use by custom implementations.

参见:

logging.captureWarnings() allows you to handle all warnings with the standard logging infrastructure.

28.6.1 警告类别

There are a number of built-in exceptions that represent warning categories. This categorization is useful to be able to filter out groups of warnings. The following warnings category classes are currently defined:

Class	描述
Warning	This is the base class of all warning category classes. It is a subclass of *Exception*.
UserWarning	The default category for *warn()*.
DeprecationWarning	Base category for warnings about deprecated features (ignored by default).
SyntaxWarning	Base category for warnings about dubious syntactic features.
RuntimeWarning	Base category for warnings about dubious runtime features.
FutureWarning	Base category for warnings about constructs that will change semantically in the future.
PendingDeprecationWarning	Base category for warnings about features that will be deprecated in the future (ignored by default).
ImportWarning	Base category for warnings triggered during the process of importing a module (ignored by default).
UnicodeWarning	Base category for warnings related to Unicode.
BytesWarning	Base category for warnings related to bytes and bytearray.

While these are technically built-in exceptions, they are documented here, because conceptually they belong to the warnings mechanism.

User code can define additional warning categories by subclassing one of the standard warning categories. A warning category must always be a subclass of the *Warning* class.

在 2.7 版更改: *DeprecationWarning* is ignored by default.

28.6.2 The Warnings Filter

The warnings filter controls whether warnings are ignored, displayed, or turned into errors (raising an exception).

Conceptually, the warnings filter maintains an ordered list of filter specifications; any specific warning is matched against each filter specification in the list in turn until a match is found; the match determines the disposition of the match. Each entry is a tuple of the form (*action*, *message*, *category*, *module*, *lineno*), where:

- *action* is one of the following strings:

值	处置
"error"	将匹配警告转换为异常
"ignore"	从不打印匹配的警告
"always"	总是打印匹配的警告
"default"	print the first occurrence of matching warnings for each location where the warning is issued
"module"	print the first occurrence of matching warnings for each module where the warning is issued
"once"	无论位置如何，仅打印第一次出现的匹配警告

- *message* is a string containing a regular expression that the start of the warning message must match. The expression is compiled to always be case-insensitive.
- *category* is a class (a subclass of *Warning*) of which the warning category must be a subclass in order to match.
- *module* is a string containing a regular expression that the module name must match. The expression is compiled to be case-sensitive.
- *lineno* is an integer that the line number where the warning occurred must match, or 0 to match all line numbers.

Since the *Warning* class is derived from the built-in *Exception* class, to turn a warning into an error we simply raise `category(message)`.

The warnings filter is initialized by -W options passed to the Python interpreter command line. The interpreter saves the arguments for all -W options without interpretation in `sys.warnoptions`; the *warnings* module parses these when it is first imported (invalid options are ignored, after printing a message to `sys.stderr`).

Default Warning Filters

By default, Python installs several warning filters, which can be overridden by the command-line options passed to -W and calls to *filterwarnings()*.

- *DeprecationWarning* and *PendingDeprecationWarning*, and *ImportWarning* are ignored.
- *BytesWarning* is ignored unless the -b option is given once or twice; in this case this warning is either printed (-b) or turned into an exception (-bb).

28.6.3 暂时禁止警告

If you are using code that you know will raise a warning, such as a deprecated function, but do not want to see the warning, then it is possible to suppress the warning using the *catch_warnings* context manager:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

While within the context manager all warnings will simply be ignored. This allows you to use known-deprecated code without having to see the warning while not suppressing the warning for other code that might not be aware of its use of deprecated code. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the *catch_warnings* context manager at the same time, the behavior is undefined.

28.6.4 测试警告

To test warnings raised by code, use the *catch_warnings* context manager. With it you can temporarily mutate the warnings filter to facilitate your testing. For instance, do the following to capture all raised warnings to check:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
    warnings.simplefilter("always")
    # Trigger a warning.
    fxn()
    # Verify some things
    assert len(w) == 1
    assert issubclass(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

One can also cause all warnings to be exceptions by using `error` instead of `always`. One thing to be aware of is that if a warning has already been raised because of a `once`/`default` rule, then no matter what filters are set the warning will not be seen again unless the warnings registry related to the warning has been cleared.

Once the context manager exits, the warnings filter is restored to its state when the context was entered. This prevents tests from changing the warnings filter in unexpected ways between tests and leading to indeterminate test results. The *showwarning()* function in the module is also restored to its original value. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the *catch_warnings* context manager at the same time, the behavior is undefined.

When testing multiple operations that raise the same kind of warning, it is important to test them in a manner that confirms each operation is raising a new warning (e.g. set warnings to be raised as exceptions and check the operations raise exceptions, check that the length of the warning list continues to increase after each operation, or else delete the previous entries from the warnings list before each new operation).

28.6.5 Updating Code For New Versions of Python

Warnings that are only of interest to the developer are ignored by default. As such you should make sure to test your code with typically ignored warnings made visible. You can do this from the command-line by passing `-Wd` to the interpreter (this is shorthand for `-W default`). This enables default handling for all warnings, including those that are ignored by default. To change what action is taken for encountered warnings you simply change what argument is passed to `-W`, e.g. `-W error`. See the `-W` flag for more details on what is possible.

To programmatically do the same as `-Wd`, use:

```
warnings.simplefilter('default')
```

Make sure to execute this code as soon as possible. This prevents the registering of what warnings have been raised from unexpectedly influencing how future warnings are treated.

Having certain warnings ignored by default is done to prevent a user from seeing warnings that are only of interest to the developer. As you do not necessarily have control over what interpreter a user uses to run their code, it is possible that a new version of Python will be released between your release cycles. The new interpreter release could trigger new warnings in your code that were not there in an older interpreter, e.g. *DeprecationWarning* for a module that you are using. While you as a developer want to be notified that your code is using a deprecated module, to a user this information is essentially noise and provides no benefit to them.

28.6.6 Available Functions

warnings.**warn**(*message*[, *category*[, *stacklevel*]])

Issue a warning, or maybe ignore it or raise an exception. The *category* argument, if given, must be a warning category class (see above); it defaults to *UserWarning*. Alternatively *message* can be a *Warning* instance, in which case *category* will be ignored and `message.__class__` will be used. In this case the message text will be `str(message)`. This function raises an exception if the particular warning issued is changed into an error by the warnings filter see above. The *stacklevel* argument can be used by wrapper functions written in Python, like this:

```
def deprecation(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

This makes the warning refer to `deprecation()`'s caller, rather than to the source of `deprecation()` itself (since the latter would defeat the purpose of the warning message).

warnings.**warn_explicit**(*message*, *category*, *filename*, *lineno*[, *module*[, *registry*[, *module_globals*]]])

This is a low-level interface to the functionality of *warn()*, passing in explicitly the message, category, filename and line number, and optionally the module name and the registry (which should be the `__warningregistry__` dictionary of the module). The module name defaults to the filename with `.py` stripped; if no registry is passed, the warning is never suppressed. *message* must be a string and *category* a subclass of *Warning* or *message* may be a *Warning* instance, in which case *category* will be ignored.

module_globals, if supplied, should be the global namespace in use by the code for which the warning is issued. (This argument is used to support displaying source for modules found in zipfiles or other non-filesystem import sources).

在 2.5 版更改: Added the *module_globals* parameter.

warnings.**warnpy3k**(*message*[, *category*[, *stacklevel*]])

Issue a warning related to Python 3.x deprecation. Warnings are only shown when Python is started with the -3 option. Like *warn()* *message* must be a string and *category* a subclass of *Warning*. *warnpy3k()* is using *DeprecationWarning* as default warning class.

2.6 新版功能.

warnings.**showwarning**(*message*, *category*, *filename*, *lineno*[, *file*[, *line*]])

Write a warning to a file. The default implementation calls `formatwarning(message, category, filename, lineno, line)` and writes the resulting string to *file*, which defaults to `sys.stderr`. You may replace this function with an alternative implementation by assigning to `warnings.showwarning`. *line* is a line of source code to be included in the warning message; if *line* is not supplied, *showwarning()* will try to read the line specified by *filename* and *lineno*.

在 2.7 版更改: The *line* argument is required to be supported.

warnings.**formatwarning**(*message*, *category*, *filename*, *lineno*[, *line*])

Format a warning the standard way. This returns a string which may contain embedded newlines and ends in a newline. *line* is a line of source code to be included in the warning message; if *line* is not supplied, *formatwarning()* will try to read the line specified by *filename* and *lineno*.

在 2.6 版更改: Added the *line* argument.

warnings.**filterwarnings**(*action*[, *message*[, *category*[, *module*[, *lineno*[, *append*]]]]])

Insert an entry into the list of *warnings filter specifications*. The entry is inserted at the front by default; if *append* is true, it is inserted at the end. This checks the types of the arguments, compiles the *message* and *module* regular expressions, and inserts them as a tuple in the list of warnings filters. Entries closer to the front of the list override entries later in the list, if both match a particular warning. Omitted arguments default to a value that matches everything.

warnings.**simplefilter**(*action*[, *category*[, *lineno*[, *append*]]])
Insert a simple entry into the list of *warnings filter specifications*. The meaning of the function parameters is as for *filterwarnings()*, but regular expressions are not needed as the filter inserted always matches any message in any module as long as the category and line number match.

warnings.**resetwarnings**()
Reset the warnings filter. This discards the effect of all previous calls to *filterwarnings()*, including that of the -W command line options and calls to *simplefilter()*.

28.6.7 Available Context Managers

class warnings.**catch_warnings**([*, *record=False*, *module=None*])
A context manager that copies and, upon exit, restores the warnings filter and the *showwarning()* function. If the *record* argument is *False* (the default) the context manager returns *None* on entry. If *record* is *True*, a list is returned that is progressively populated with objects as seen by a custom *showwarning()* function (which also suppresses output to sys.stdout). Each object in the list has attributes with the same names as the arguments to *showwarning()*.

The *module* argument takes a module that will be used instead of the module returned when you import *warnings* whose filter will be protected. This argument exists primarily for testing the *warnings* module itself.

注解: The *catch_warnings* manager works by replacing and then later restoring the module's *showwarning()* function and internal list of filter specifications. This means the context manager is modifying global state and therefore is not thread-safe.

注解: In Python 3, the arguments to the constructor for *catch_warnings* are keyword-only arguments.

2.6 新版功能.

28.7 contextlib —Utilities for with-statement contexts

2.5 新版功能.

源代码 Lib/contextlib.py

此模块为涉及 with 语句的常见任务提供了实用的程序。更多信息请参见上下文管理器类型 和 context-managers。

Functions provided:

contextlib.**contextmanager**(*func*)
This function is a *decorator* that can be used to define a factory function for with statement context managers, without needing to create a class or separate __enter__() and __exit__() methods.

While many objects natively support use in with statements, sometimes a resource needs to be managed that isn't a context manager in its own right, and doesn't implement a close() method for use with contextlib.closing

An abstract example would be the following to ensure correct resource management:

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwds):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwds)
    try:
        yield resource
    finally:
        # Code to release resource, e.g.:
        release_resource(resource)

>>> with managed_resource(timeout=3600) as resource:
...     # Resource is released at the end of this block,
...     # even if code in the block raises an exception
```

The function being decorated must return a *generator*-iterator when called. This iterator must yield exactly one value, which will be bound to the targets in the `with` statement's `as` clause, if any.

At the point where the generator yields, the block nested in the `with` statement is executed. The generator is then resumed after the block is exited. If an unhandled exception occurs in the block, it is reraised inside the generator at the point where the yield occurred. Thus, you can use a `try…except…finally` statement to trap the error (if any), or ensure that some cleanup takes place. If an exception is trapped merely in order to log it or to perform some action (rather than to suppress it entirely), the generator must reraise that exception. Otherwise the generator context manager will indicate to the `with` statement that the exception has been handled, and execution will resume with the statement immediately following the `with` statement.

`contextlib.`**`nested`**(*mgr1*[, *mgr2*[, ...]])

Combine multiple context managers into a single nested context manager.

This function has been deprecated in favour of the multiple manager form of the `with` statement.

The one advantage of this function over the multiple manager form of the `with` statement is that argument unpacking allows it to be used with a variable number of context managers as follows:

```
from contextlib import nested

with nested(*managers):
    do_something()
```

Note that if the `__exit__()` method of one of the nested context managers indicates an exception should be suppressed, no exception information will be passed to any remaining outer context managers. Similarly, if the `__exit__()` method of one of the nested managers raises an exception, any previous exception state will be lost; the new exception will be passed to the `__exit__()` methods of any remaining outer context managers. In general, `__exit__()` methods should avoid raising exceptions, and in particular they should not re-raise a passed-in exception.

This function has two major quirks that have led to it being deprecated. Firstly, as the context managers are all constructed before the function is invoked, the `__new__()` and `__init__()` methods of the inner context managers are not actually covered by the scope of the outer context managers. That means, for example, that using *`nested()`* to open two files is a programming error as the first file will not be closed promptly if an exception is thrown when opening the second file.

Secondly, if the `__enter__()` method of one of the inner context managers raises an exception that is caught and suppressed by the `__exit__()` method of one of the outer context managers, this construct will raise *`RuntimeError`* rather than skipping the body of the `with` statement.

Developers that need to support nesting of a variable number of context managers can either use the *`warnings`* module to suppress the DeprecationWarning raised by this function or else use this function as a model for an

application specific implementation.

2.7 版后已移除: The with-statement now supports this functionality directly (without the confusing error prone quirks).

contextlib.**closing**(*thing*)

Return a context manager that closes *thing* upon completion of the block. This is basically equivalent to:

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

And lets you write code like this:

```
from contextlib import closing
import urllib

with closing(urllib.urlopen('http://www.python.org')) as page:
    for line in page:
        print line
```

without needing to explicitly close `page`. Even if an error occurs, `page.close()` will be called when the `with` block is exited.

参见:

PEP 343 - "with" 语句 Python `with` 语句的规范描述、背景和示例。

28.8 abc —抽象基类

2.6 新版功能.

源代码: Lib/abc.py

该模块提供了在 Python 中定义抽象基类 (ABC) 的组件，在 **PEP 3119** 中已有概述。查看 PEP 文档了解为什么需要在 Python 中增加这个模块。（也可查看 **PEP 3141** 以及*numbers* 模块了解基于 ABC 的数字类型继承关系。）

The *collections* module has some concrete classes that derive from ABCs; these can, of course, be further derived. In addition, the *collections* module has some ABCs that can be used to test whether a class or instance provides a particular interface, for example, if it is hashable or if it is a mapping.

This module provides the following class:

class abc.**ABCMeta**

用于定义抽象基类（ABC）的元类。

使用该元类以创建抽象基类。抽象基类可以像 mix-in 类一样直接被子类继承。你也可以将不相关的具体类（包括内建类）和抽象基类注册为“抽象子类”——这些类以及它们的子类会被内建函数*issubclass()* 识别为对应的抽象基类的子类，但是该抽象基类不会出现在其 MRO（Method

Resolution Order，方法解析顺序）中，抽象基类中实现的方法也不可调用（即使通过*super()* 调用也不行）。[1]

使用*ABCMeta* 作为元类创建的类含有如下方法：

register(*subclass*)
将“子类”注册为该抽象基类的“抽象子类”，例如：

```
from abc import ABCMeta

class MyABC:
    __metaclass__ = ABCMeta

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

你也可以在虚基类中重载这个方法。

__subclasshook__(*subclass*)
（必须定义为类方法。）

检查 *subclass* 是否是该抽象基类的子类。也就是说对于那些你希望定义为该抽象基类的子类的类，你不用对每个类都调用*register()* 方法了，而是可以直接自定义 issubclass 的行为。（这个类方法是在抽象基类的 __subclasscheck__() 方法中调用的。）

该方法必须返回 True, False 或是 NotImplemented。如果返回 True，*subclass* 就会被认为是这个抽象基类的子类。如果返回 False，无论正常情况是否应该认为是其子类，统一视为不是。如果返回 NotImplemented，子类检查会按照正常机制继续执行。

为了对这些概念做一演示，请看以下定义 ABC 的示例：

```
class Foo(object):
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable:
    __metaclass__ = ABCMeta

    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented
```

(下页继续)

[1] C++ 程序员需要注意：Python 中虚基类的概念和 C++ 中的并不相同。

(续上页)

```
MyIterable.register(Foo)
```

ABC `MyIterable` 定义了标准的迭代方法*__iter__()* 作为一个抽象方法。这里给出的实现仍可在子类中被调用。`get_iterator()` 方法也是 `MyIterable` 抽象基类的一部分，但它并非必须被非抽象派生类所重载。

The *__subclasshook__()* class method defined here says that any class that has an *__iter__()* method in its *__dict__* (or in that of one of its base classes, accessed via the *__mro__* list) is considered a `MyIterable` too.

Finally, the last line makes `Foo` a virtual subclass of `MyIterable`, even though it does not define an *__iter__()* method (it uses the old-style iterable protocol, defined in terms of `__len__()` and `__getitem__()`). Note that this will not make `get_iterator` available as a method of `Foo`, so it is provided separately.

It also provides the following decorators:

`abc.`**`abstractmethod`**(*function*)
用于声明抽象方法的装饰器。

Using this decorator requires that the class's metaclass is *ABCMeta* or is derived from it. A class that has a metaclass derived from *ABCMeta* cannot be instantiated unless all of its abstract methods and properties are overridden. The abstract methods can be called using any of the normal 'super' call mechanisms.

Dynamically adding abstract methods to a class, or attempting to modify the abstraction status of a method or class once it is created, are not supported. The *abstractmethod()* only affects subclasses derived using regular inheritance; "virtual subclasses" registered with the ABC's `register()` method are not affected.

Usage:

```
class C:
    __metaclass__ = ABCMeta
    @abstractmethod
    def my_abstract_method(self, ...):
        ...
```

注解： Unlike Java abstract methods, these abstract methods may have an implementation. This implementation can be called via the *super()* mechanism from the class that overrides it. This could be useful as an end-point for a super-call in a framework that uses cooperative multiple-inheritance.

`abc.`**`abstractproperty`**([*fget*[, *fset*[, *fdel*[, *doc*]]]])
A subclass of the built-in *property()*, indicating an abstract property.

Using this function requires that the class's metaclass is *ABCMeta* or is derived from it. A class that has a metaclass derived from *ABCMeta* cannot be instantiated unless all of its abstract methods and properties are overridden. The abstract properties can be called using any of the normal 'super' call mechanisms.

Usage:

```
class C:
    __metaclass__ = ABCMeta
    @abstractproperty
    def my_abstract_property(self):
        ...
```

This defines a read-only property; you can also define a read-write abstract property using the 'long' form of property declaration:

```
class C:
    __metaclass__ = ABCMeta
    def getx(self): ...
    def setx(self, value): ...
    x = abstractproperty(getx, setx)
```

备注

28.9 atexit —退出处理器

2.0 新版功能.

Source code: Lib/atexit.py

The *atexit* module defines a single function to register cleanup functions. Functions thus registered are automatically executed upon normal interpreter termination. *atexit* runs these functions in the *reverse* order in which they were registered; if you register A, B, and C, at interpreter termination time they will be run in the order C, B, A.

注意: 通过该模块注册的函数, 在程序被未被 Python 捕获的信号杀死时并不会执行, 在检测到 Python 内部致命错误以及调用了*os._exit()* 时也不会执行.

This is an alternate interface to the functionality provided by the *sys.exitfunc()* variable.

Note: This module is unlikely to work correctly when used with other code that sets `sys.exitfunc`. In particular, other core Python modules are free to use *atexit* without the programmer's knowledge. Authors who use `sys.exitfunc` should convert their code to use *atexit* instead. The simplest way to convert code that sets `sys.exitfunc` is to import *atexit* and register the function that had been bound to `sys.exitfunc`.

`atexit.`**register**(*func*[, **args*[, ***kwargs*]])

将 *func* 注册为终止时执行的函数. 任何传给 *func* 的可选的参数都应当作为参数传给*register()*. 可以多次注册同样的函数及参数.

在正常的程序终止时 (举例来说, 当调用了*sys.exit()* 或是主模块的执行完成时), 所有注册过的函数都会以后进先出的顺序执行. 这样做是假定更底层的模块通常会比高层模块更早引入, 因此需要更晚清理.

如果在 exit 处理程序执行期间引发了异常，将会打印回溯信息 (除非引发的是*SystemExit*) 并且异常信息会被保存。在所有 exit 处理程序获得运行机会之后，所引发的最后一个异常会被重新引发。

在 2.6 版更改: This function now returns *func*, which makes it possible to use it as a decorator.

参见:

模块*readline* 使用*atexit* 读写*readline* 历史文件的有用的例子。

28.9.1 atexit 示例

以下简单例子演示了一个模块在被导入时如何从文件初始化一个计数器，并在程序终结时自动保存计数器的更新值，此操作不依赖于应用在终结时对此模块进行显式调用。:

```
try:
    _count = int(open("counter").read())
except IOError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    open("counter", "w").write("%d" % _count)

import atexit
atexit.register(savecounter)
```

位置和关键字参数也可传入*register()* 以便传递给被调用的已注册函数:

```
def goodbye(name, adjective):
    print 'Goodbye, %s, it was %s to meet you.' % (name, adjective)

import atexit
atexit.register(goodbye, 'Donny', 'nice')

# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

作为*decorator*: 使用:

```
import atexit

@atexit.register
def goodbye():
    print "You are now leaving the Python sector."
```

只有在函数不需要任何参数调用时才能工作.

28.10 traceback —打印或检索堆栈回溯

该模块提供了一个标准接口来提取、格式化和打印 Python 程序的堆栈跟踪结果。它完全模仿 Python 解释器在打印堆栈跟踪结果时的行为。当您想要在程序控制下打印堆栈跟踪结果时，例如在“封装”解释器时，这是非常有用的。

The module uses traceback objects —this is the object type that is stored in the variables *sys.exc_traceback* (deprecated) and *sys.last_traceback* and returned as the third item from *sys.exc_info()*.

这个模块定义了以下函数:

`traceback.`**`print_tb`**(*tb*[, *limit*[, *file*]])

Print up to *limit* stack trace entries from the traceback object *tb*. If *limit* is omitted or `None`, all entries are printed. If *file* is omitted or `None`, the output goes to `sys.stderr`; otherwise it should be an open file or file-like object to receive the output.

traceback.**print_exception**(*etype*, *value*, *tb*[, *limit*[, *file*]])
Print exception information and up to *limit* stack trace entries from the traceback *tb* to *file*. This differs from *print_tb()* in the following ways: (1) if *tb* is not None, it prints a header Traceback (most recent call last):; (2) it prints the exception *etype* and *value* after the stack trace; (3) if *etype* is *SyntaxError* and *value* has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.

traceback.**print_exc**([*limit*[, *file*]])
This is a shorthand for print_exception(sys.exc_type, sys.exc_value, sys.exc_traceback, limit, file). (In fact, it uses *sys.exc_info()* to retrieve the same information in a thread-safe way instead of using the deprecated variables.)

traceback.**format_exc**([*limit*])
This is like print_exc(limit) but returns a string instead of printing to a file.

2.4 新版功能.

traceback.**print_last**([*limit*[, *file*]])
This is a shorthand for print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file). In general it will work only after an exception has reached an interactive prompt (see *sys.last_type*).

traceback.**print_stack**([*f*[, *limit*[, *file*]]])
This function prints a stack trace from its invocation point. The optional *f* argument can be used to specify an alternate stack frame to start. The optional *limit* and *file* arguments have the same meaning as for *print_exception()*.

traceback.**extract_tb**(*tb*[, *limit*])
Return a list of up to *limit* "pre-processed" stack trace entries extracted from the traceback object *tb*. It is useful for alternate formatting of stack traces. If *limit* is omitted or None, all entries are extracted. A "pre-processed" stack trace entry is a 4-tuple (*filename*, *line number*, function name*, *text*) representing the information that is usually printed for a stack trace. The *text* is a string with leading and trailing whitespace stripped; if the source is not available it is None.

traceback.**extract_stack**([*f*[, *limit*]])
Extract the raw traceback from the current stack frame. The return value has the same format as for *extract_tb()*. The optional *f* and *limit* arguments have the same meaning as for *print_stack()*.

traceback.**format_list**(*extracted_list*)
Given a list of tuples as returned by *extract_tb()* or *extract_stack()*, return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline; the strings may contain internal newlines as well, for those items whose source text line is not None.

traceback.**format_exception_only**(*etype*, *value*)
Format the exception part of a traceback. The arguments are the exception type, *etype* and *value* such as given by sys.last_type and sys.last_value. The return value is a list of strings, each ending in a newline. Normally, the list contains a single string; however, for *SyntaxError* exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. The message indicating which exception occurred is the always last string in the list.

traceback.**format_exception**(*etype*, *value*, *tb*[, *limit*])
Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to *print_exception()*. The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does *print_exception()*.

traceback.**format_tb**(*tb*[, *limit*])
A shorthand for format_list(extract_tb(tb, limit)).

traceback.**format_stack**([*f*[, *limit*]])
A shorthand for format_list(extract_stack(f, limit)).

traceback.**tb_lineno**(*tb*)
This function returns the current line number set in the traceback object. This function was necessary because in versions of Python prior to 2.3 when the -O flag was passed to Python the tb.tb_lineno was not updated correctly. This function has no use in versions past 2.3.

28.10.1 Traceback Examples

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the *code* module.

```
import sys, traceback

def run_user_code(envdir):
    source = raw_input(">>> ")
    try:
        exec source in envdir
    except:
        print "Exception in user code:"
        print '-'*60
        traceback.print_exc(file=sys.stdout)
        print '-'*60

envdir = {}
while 1:
    run_user_code(envdir)
```

The following example demonstrates the different ways to print and format the exception and traceback:

```
import sys, traceback

def lumberjack():
    bright_side_of_death()

def bright_side_of_death():
    return tuple()[0]

try:
    lumberjack()
except IndexError:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print "*** print_tb:"
    traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
    print "*** print_exception:"
    traceback.print_exception(exc_type, exc_value, exc_traceback,
                              limit=2, file=sys.stdout)
    print "*** print_exc:"
    traceback.print_exc()
    print "*** format_exc, first and last line:"
    formatted_lines = traceback.format_exc().splitlines()
    print formatted_lines[0]
    print formatted_lines[-1]
    print "*** format_exception:"
    print repr(traceback.format_exception(exc_type, exc_value,
```

(下页继续)

(续上页)

```
                                          exc_traceback))
    print "*** extract_tb:"
    print repr(traceback.extract_tb(exc_traceback))
    print "*** format_tb:"
    print repr(traceback.format_tb(exc_traceback))
    print "*** tb_lineno:", exc_traceback.tb_lineno
```

The output for the example would look similar to this:

```
*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n',
 'IndexError: tuple index out of range\n']
*** extract_tb:
[('<doctest...>', 10, '<module>', 'lumberjack()'),
 ('<doctest...>', 4, 'lumberjack', 'bright_side_of_death()'),
 ('<doctest...>', 7, 'bright_side_of_death', 'return tuple()[0]')]
*** format_tb:
['  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n']
*** tb_lineno: 10
```

The following example shows the different ways to print and format the stack:

```
>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print repr(traceback.extract_stack())
...     print repr(traceback.format_stack())
...
```

(下页继续)

(续上页)

```
>>> another_function()
  File "<doctest>", line 10, in <module>
    another_function()
  File "<doctest>", line 3, in another_function
    lumberstack()
  File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print repr(traceback.extract_stack())')]
['  File "<doctest>", line 10, in <module>\n    another_function()\n',
 '  File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n    print repr(traceback.format_
↪stack())\n']
```

This last example demonstrates the final few formatting functions:

```
>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
...                        ('eggs.py', 42, 'eggs', 'return "bacon"')])
['  File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 '  File "eggs.py", line 42, in eggs\n    return "bacon"\n']
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(type(an_error), an_error)
['IndexError: tuple index out of range\n']
```

28.11 __future__ —Future 语句定义

源代码： Lib/__future__.py

__future__ 是一个真正的模块，这主要有 3 个原因：

- 避免混淆已有的分析 import 语句并查找 import 的模块的工具。
- 确保 future 语句在 2.1 之前的版本运行时至少能抛出 runtime 异常（import *__future__* 会失败，因为 2.1 版本之前没有这个模块）。
- 当引入不兼容的修改时，可以记录其引入的时间以及强制使用的时间。这是一种可执行的文档，并且可以通过 import *__future__* 来做程序性的检查。

`__future__.py` 中的每一条语句都是以下格式的：

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                       CompilerFlag)
```

where, normally, *OptionalRelease* is less than *MandatoryRelease*, and both are 5-tuples of the same form as `sys.version_info`:

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
 PY_MINOR_VERSION, # the 1; an int
 PY_MICRO_VERSION, # the 0; an int
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string
 PY_RELEASE_SERIAL # the 3; an int
)
```

OptionalRelease 记录了一个特性首次发布时的 Python 版本。

在 *MandatoryRelases* 还没有发布时，*MandatoryRelease* 表示该特性会变成语言的一部分的预测时间。

其他情况下，*MandatoryRelease* 用来记录这个特性是何时成为语言的一部分的。从该版本往后，使用该特性将不需要 future 语句，不过很多人还是会加上对应的 import。

MandatoryRelease 也可能是 `None`, 表示这个特性已经被撤销。

`_Feature` 类的实例有两个对应的方法，`getOptionalRelease()` 和 `getMandatoryRelease()`。

CompilerFlag 是一个（位）标记，对于动态编译的代码，需要将这个标记作为第四个参数传入内建函数*compile()* 中以开启对应的特性。这个标记存储在 `_Feature` 类实例的 `compiler_flag` 属性中。

__future__ 中不会删除特性的描述。从 Python 2.1 中首次加入以来，通过这种方式引入了以下特性：

特性	可选版本	强制加入版本	效果
nested_scopes	2.1.0b1	2.2	**PEP 227**: 静态嵌套作用域
generators	2.2.0a1	2.3	**PEP 255**: 简单生成器
division	2.2.0a2	3.0	**PEP 238**: 修改除法运算符
absolute_import	2.5.0a1	3.0	**PEP 328**: 导入：多行与绝对/相对
with_statement	2.5.0a1	2.6	**PEP 343**: * “with” 语句 *
print_function	2.6.0a2	3.0	**PEP 3105**: *print* 改为函数
unicode_literals	2.6.0a2	3.0	**PEP 3112**: *Python 3000* 中的字节字面值

参见:

future 编译器怎样处理 future import。

28.12 gc —垃圾回收器接口

此模块提供可选的垃圾回收器的接口，提供的功能包括：关闭收集器、调整收集频率、设置调试选项。它同时提供对回收器找到但是无法释放的不可达对象的访问。由于 Python 使用了带有引用计数的回收器，如果你确定你的程序不会产生循环引用，你可以关闭回收器。可以通过调用 `gc.disable()` 关闭自动垃圾回收。若要调试一个存在内存泄漏的程序，调用 `gc.set_debug(gc.DEBUG_LEAK)`；需要注意的是，它包含 `gc.DEBUG_SAVEALL`，使得被垃圾回收的对象会被存放在 gc.garbage 中以待检查。

gc 模块提供下列函数：

`gc.`**`enable`**`()`
启用自动垃圾回收

`gc.`**`disable`**`()`
停用自动垃圾回收

`gc.`**`isenabled`**`()`
Returns true if automatic collection is enabled.

`gc.`**`collect`**`([`*generation*`])`
若被调用时不包含参数，则启动完全的垃圾回收。可选的参数 *generation* 可以是一个整数，指明需要回收哪一代（从 0 到 2）的垃圾。当参数 *generation* 无效时，会引发*ValueError* 异常。返回发现的不可达对象的数目。

在 2.5 版更改: The optional *generation* argument was added.

在 2.6 版更改: The free lists maintained for a number of built-in types are cleared whenever a full collection or collection of the highest generation (2) is run. Not all items in some free lists may be freed due to the particular implementation, in particular *int* and *float*.

gc.**set_debug**(*flags*)
设置垃圾回收器的调试标识位。调试信息会被写入 sys.stderr 。此文档末尾列出了各个标志位及其含义；可以使用位操作对多个标志位进行设置以控制调试器。

gc.**get_debug**()
返回当前调试标识位。

gc.**get_objects**()
返回一个含有所有被收集器跟踪的对象的列表。返回的列表中不包括该函数返回的对象。

2.2 新版功能.

gc.**set_threshold**(*threshold0*[, *threshold1*[, *threshold2*]])
设置垃圾回收阈值（收集频率）。将 *threshold0* 设为零会禁用回收。

垃圾回收器把所有对象分类为三代，取决于对象幸存于多少次垃圾回收。新创建的对象会被放在最年轻代（第 0 代）。如果一个对象幸存于一次垃圾回收，则该对象会被放入下一代。第 2 代是最老的一代，因此这一代的对象幸存于垃圾回收后，仍会留在第 2 代。为了判定何时需要进行垃圾回收，垃圾回收器会跟踪上一次回收后，分配和释放的对象的数目。当分配对象的数量减去释放对象的数量大于阈值 *threshold0* 时，回收器开始进行垃圾回收。起初只有第 0 代会被检查。当上一次第 1 代被检查后，第 0 代被检查的次数多于阈值 *threshold1* 时，第 1 代也会被检查。相似的，*threshold2* 设置了触发第 2 代被垃圾回收的第 1 代被垃圾回收的次数。

gc.**get_count**()
将当前回收计数以形为 (count0, count1, count2) 的元组返回。

2.5 新版功能.

gc.**get_threshold**()
将当前回收阈值以形为 (threshold0, threshold1, threshold2) 的元组返回。

gc.**get_referrers**(**objs*)
返回直接引用任意一个 *ojbs* 的对象列表。这个函数只定位支持垃圾回收的容器；引用了其它对象但不支持垃圾回收的扩展类型不会被找到。

需要注意的是，已经解除对 *objs* 引用的对象，但仍存在于循环引用中未被回收时，仍然会被作为引用者出现在返回的列表当中。若要获取当前正在引用 *objs* 的对象，需要调用*collect()* 然后再调用*get_referrers()* 。

在使用*get_referrers()* 返回的对象时必须要小心，因为其中一些对象可能仍在构造中因此处于暂时的无效状态。不要把*get_referrers()* 用于调试以外的其它目的。

2.2 新版功能.

gc.**get_referents**(**objs*)
返回被任意一个参数中的对象直接引用的对象的列表。返回的被引用对象是被参数中的对象的 C 语言级别方法（若存在）tp_traverse 访问到的对象，可能不是所有的实际直接可达对象。只有支持垃圾回收的对象支持 tp_traverse 方法，并且此方法只会在需要访问涉及循环引用的对象时使用。因此，可以有以下例子：一个整数对其中一个参数是直接可达的，这个整数有可能出现或不出现在返回的结果列表当中。

2.3 新版功能.

gc.**is_tracked**(*obj*)
当对象正在被垃圾回收器监控时返回 True ，否则返回 False 。一般来说，原子类的实例不会被监控，

而非原子类（如容器、用户自定义的对象）会被监控。然而，会有一些特定类型的优化以便减少垃圾回收器在简单实例（如只含有原子性的键和值的字典）上的消耗。

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
False
>>> gc.is_tracked({"a": []})
True
```

2.7 新版功能.

The following variable is provided for read-only access (you can mutate its value but should not rebind it):

gc.**garbage**
A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects). By default, this list contains only objects with `__del__()` methods.[1] Objects that have `__del__()` methods and are part of a reference cycle cause the entire reference cycle to be uncollectable, including objects not necessarily in the cycle but reachable only from it. Python doesn't collect such cycles automatically because, in general, it isn't possible for Python to guess a safe order in which to run the `__del__()` methods. If you know a safe order, you can force the issue by examining the *garbage* list, and explicitly breaking cycles due to your objects within the list. Note that these objects are kept alive even so by virtue of being in the *garbage* list, so they should be removed from *garbage* too. For example, after breaking cycles, do `del gc.garbage[:]` to empty the list. It's generally better to avoid the issue by not creating cycles containing objects with `__del__()` methods, and *garbage* can be examined in that case to verify that no such cycles are being created.

如果设置了*DEBUG_SAVEALL*，则所有不可访问对象将被添加至该列表而不会被释放。

以下常量被用于*set_debug()*：

gc.**DEBUG_STATS**
在回收完成后打印统计信息。当回收频率设置较高时，这些信息会比较有用。

gc.**DEBUG_COLLECTABLE**
当发现可回收对象时打印信息。

gc.**DEBUG_UNCOLLECTABLE**
打印找到的不可回收对象的信息（指不能被回收器回收的不可达对象）。这些对象会被添加到 `garbage` 列表中。

gc.**DEBUG_INSTANCES**
When *DEBUG_COLLECTABLE* or *DEBUG_UNCOLLECTABLE* is set, print information about instance objects found.

gc.**DEBUG_OBJECTS**
When *DEBUG_COLLECTABLE* or *DEBUG_UNCOLLECTABLE* is set, print information about objects other than instance objects found.

gc.**DEBUG_SAVEALL**
设置后，所有回收器找到的不可达对象会被添加进 *garbage* 而不是直接被释放。这在调试一个内存泄漏的程序时会很有用。

[1] Prior to Python 2.2, the list contained all instance objects in unreachable cycles, not only those with `__del__()` methods.

gc.**DEBUG_LEAK**
The debugging flags necessary for the collector to print information about a leaking program (equal to DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_INSTANCES | DEBUG_OBJECTS | DEBUG_SAVEALL).

28.13 inspect —检查对象

2.1 新版功能.

源代码: Lib/inspect.py

---

inspect 模块提供了一些有用的函数帮助获取对象的信息，例如模块、类、方法、函数、回溯、帧对象以及代码对象。例如它可以帮助你检查类的内容，获取某个方法的源代码，取得并格式化某个函数的参数列表，或者获取你需要显示的回溯的详细信息。

该模块提供了 4 种主要的功能：类型检查、获取源代码、检查类与函数、检查解释器的调用堆栈。

28.13.1 类型和成员

The *getmembers()* function retrieves the members of an object such as a class or module. The sixteen functions whose names begin with "is" are mainly provided as convenient choices for the second argument to *getmembers()*. They also help you determine when you can expect to find the following special attributes:

类型	属性	描述
module 模块	__doc__	文档字符串
	__file__	文件名 (内置模块没有文件名)
class –类	__doc__	文档字符串
	__module__	该类型被定义时所在的模块的名称
method 方法	__doc__	文档字符串
	__name__	该方法定义时所使用的名称
	im_class	class object that asked for this method
	im_func or __func__	实现该方法的函数对象
	im_self or __self__	该方法被绑定的实例，若没有绑定则为 None
function –函数	__doc__	文档字符串
	__name__	用于定义此函数的名称
	func_code	包含已编译函数的代码对象*bytecode*
	func_defaults	tuple of any default values for arguments
	func_doc	(same as __doc__)
	func_globals	global namespace in which this function was defined
	func_name	(same as __name__)
generator –生成器	__iter__	defined to support iteration over container
	close	raises new GeneratorExit exception inside the generator to terminate the iteration
	gi_code	code object
	gi_frame	frame object or possibly None once the generator has been exhausted
	gi_running	set to 1 when generator is executing, 0 otherwise
	next	return the next item from the container
	send	resumes the generator and "sends" a value that becomes the result of the current yield-ex
	throw	used to raise an exception inside the generator
回溯	tb_frame	此级别的框架对象

表 1 – 续上页

类型	属性	描述
	tb_lasti	index of last attempted instruction in bytecode
	tb_lineno	current line number in Python source code
	tb_next	next inner traceback object (called by this level)
框架	f_back	next outer frame object (this frame’s caller)
	f_builtins	builtins namespace seen by this frame
	f_code	code object being executed in this frame
	f_exc_traceback	traceback if raised in this frame, or `None`
	f_exc_type	exception type if raised in this frame, or `None`
	f_exc_value	exception value if raised in this frame, or `None`
	f_globals	global namespace seen by this frame
	f_lasti	index of last attempted instruction in bytecode
	f_lineno	current line number in Python source code
	f_locals	local namespace seen by this frame
	f_restricted	0 or 1 if frame is in restricted execution mode
	f_trace	tracing function for this frame, or `None`
code	co_argcount	number of arguments (not including * or ** args)
	co_code	原始编译字节码的字符串
	co_consts	字节码中使用的常量元组
	co_filename	创建此代码对象的文件的名称
	co_firstlineno	number of first line in Python source code
	co_flags	bitmap: 1=optimized | 2=newlocals | 4=*arg | 8=**arg
	co_lnotab	编码的行号到字节码索引的映射
	co_name	定义此代码对象的名称
	co_names	局部变量名称的元组
	co_nlocals	局部变量的数量
	co_stacksize	需要虚拟机堆栈空间
	co_varnames	参数名和局部变量的元组
builtin	__doc__	文档字符串
	__name__	此函数或方法的原始名称
	__self__	instance to which a method is bound, or `None`

Note:

(1) 在 2.2 版更改: `im_class` used to refer to the class that defined the method.

`inspect.`**`getmembers`**`(`*object*`[`, *predicate*`])`

Return all the members of an object in a list of (name, value) pairs sorted by name. If the optional *predicate* argument is supplied, only members for which the predicate returns a true value are included.

注解: *`getmembers()`* does not return metaclass attributes when the argument is a class (this behavior is inherited from the *`dir()`* function).

`inspect.`**`getmoduleinfo`**`(`*path*`)`

Return a tuple of values that describe how Python will interpret the file identified by *path* if it is a module, or `None` if it would not be identified as a module. The return tuple is `(name, suffix, mode, module_type)`, where *name* is the name of the module without the name of any enclosing package, *suffix* is the trailing part of the file name (which may not be a dot-delimited extension), *mode* is the *`open()`* mode that would be used (`'r'` or `'rb'`), and *module_type* is an integer giving the type of the module. *module_type* will have a value which can be compared to the constants defined in the *`imp`* module; see the documentation for that module for more information on module types.

在 2.6 版更改: Returns a *named tuple* `ModuleInfo(name, suffix, mode, module_type)`.

`inspect.`**`getmodulename`**(*path*)
Return the name of the module named by the file *path*, without including the names of enclosing packages. This uses the same algorithm as the interpreter uses when searching for modules. If the name cannot be matched according to the interpreter's rules, `None` is returned.

`inspect.`**`ismodule`**(*object*)
Return true if the object is a module.

`inspect.`**`isclass`**(*object*)
Return true if the object is a class, whether built-in or created in Python code.

`inspect.`**`ismethod`**(*object*)
Return true if the object is a bound or unbound method written in Python.

`inspect.`**`isfunction`**(*object*)
Return true if the object is a Python function, which includes functions created by a *lambda* expression.

`inspect.`**`isgeneratorfunction`**(*object*)
Return true if the object is a Python generator function.

2.6 新版功能.

`inspect.`**`isgenerator`**(*object*)
Return true if the object is a generator.

2.6 新版功能.

`inspect.`**`istraceback`**(*object*)
Return true if the object is a traceback.

`inspect.`**`isframe`**(*object*)
Return true if the object is a frame.

`inspect.`**`iscode`**(*object*)
Return true if the object is a code.

`inspect.`**`isbuiltin`**(*object*)
Return true if the object is a built-in function or a bound built-in method.

`inspect.`**`isroutine`**(*object*)
Return true if the object is a user-defined or built-in function or method.

`inspect.`**`isabstract`**(*object*)
Return true if the object is an abstract base class.

2.6 新版功能.

`inspect.`**`ismethoddescriptor`**(*object*)
Return true if the object is a method descriptor, but not if *ismethod()*, *isclass()*, *isfunction()* or *isbuiltin()* are true.

This is new as of Python 2.2, and, for example, is true of `int.__add__`. An object passing this test has a `__get__()` method but not a `__set__()` method, but beyond that the set of attributes varies. A *__name__* attribute is usually sensible, and `__doc__` often is.

Methods implemented via descriptors that also pass one of the other tests return false from the *ismethoddescriptor()* test, simply because the other tests promise more –you can, e.g., count on having the `im_func` attribute (etc) when an object passes *ismethod()*.

`inspect.`**`isdatadescriptor`**(*object*)
Return true if the object is a data descriptor.

Data descriptors have both a `__get__` and a `__set__` method. Examples are properties (defined in Python), getsets, and members. The latter two are defined in C and there are more specific tests available for those types, which is robust across Python implementations. Typically, data descriptors will also have *__name__* and `__doc__` attributes (properties, getsets, and members have both of these attributes), but this is not guaranteed.

2.3 新版功能.

`inspect.`**`isgetsetdescriptor`**(*object*)
Return true if the object is a getset descriptor.

CPython implementation detail: getsets are attributes defined in extension modules via `PyGetSetDef` structures. For Python implementations without such types, this method will always return `False`.

2.5 新版功能.

`inspect.`**`ismemberdescriptor`**(*object*)
Return true if the object is a member descriptor.

CPython implementation detail: Member descriptors are attributes defined in extension modules via `PyMemberDef` structures. For Python implementations without such types, this method will always return `False`.

2.5 新版功能.

28.13.2 Retrieving source code

`inspect.`**`getdoc`**(*object*)
Get the documentation string for an object, cleaned up with *cleandoc()*.

`inspect.`**`getcomments`**(*object*)
Return in a single string any lines of comments immediately preceding the object's source code (for a class, function, or method), or at the top of the Python source file (if the object is a module).

`inspect.`**`getfile`**(*object*)
Return the name of the (text or binary) file in which an object was defined. This will fail with a *TypeError* if the object is a built-in module, class, or function.

`inspect.`**`getmodule`**(*object*)
Try to guess which module an object was defined in.

`inspect.`**`getsourcefile`**(*object*)
Return the name of the Python source file in which an object was defined. This will fail with a *TypeError* if the object is a built-in module, class, or function.

`inspect.`**`getsourcelines`**(*object*)
Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An *IOError* is raised if the source code cannot be retrieved.

`inspect.`**`getsource`**(*object*)
Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An *IOError* is raised if the source code cannot be retrieved.

`inspect.`**`cleandoc`**(*doc*)
Clean up indentation from docstrings that are indented to line up with blocks of code.

All leading whitespace is removed from the first line. Any leading whitespace that can be uniformly removed from the second line onwards is removed. Empty lines at the beginning and end are subsequently removed. Also, all tabs are expanded to spaces.

2.6 新版功能.

28.13.3 类与函数

inspect.**getclasstree**(*classes*[, *unique*])

Arrange the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list. Each entry is a 2-tuple containing a class and a tuple of its base classes. If the *unique* argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

inspect.**getargspec**(*func*)

Get the names and default values of a Python function's arguments. A tuple of four things is returned: `(args, varargs, keywords, defaults)`. *args* is a list of the argument names (it may contain nested lists). *varargs* and *keywords* are the names of the `*` and `**` arguments or `None`. *defaults* is a tuple of default argument values or `None` if there are no default arguments; if this tuple has *n* elements, they correspond to the last *n* elements listed in *args*.

在 2.6 版更改: Returns a *named tuple* `ArgSpec(args, varargs, keywords, defaults)`.

inspect.**getargvalues**(*frame*)

Get information about arguments passed into a particular frame. A tuple of four things is returned: `(args, varargs, keywords, locals)`. *args* is a list of the argument names (it may contain nested lists). *varargs* and *keywords* are the names of the `*` and `**` arguments or `None`. *locals* is the locals dictionary of the given frame.

在 2.6 版更改: Returns a *named tuple* `ArgInfo(args, varargs, keywords, locals)`.

inspect.**formatargspec**(*args*[, *varargs*, *varkw*, *defaults*, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*, *join*])

Format a pretty argument spec from the four values returned by *getargspec()*. The format* arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

inspect.**formatargvalues**(*args*[, *varargs*, *varkw*, *locals*, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*, *join*])

Format a pretty argument spec from the four values returned by *getargvalues()*. The format* arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

inspect.**getmro**(*cls*)

Return a tuple of class cls's base classes, including cls, in method resolution order. No class appears more than once in this tuple. Note that the method resolution order depends on cls's type. Unless a very peculiar user-defined metatype is in use, cls will be the first element of the tuple.

inspect.**getcallargs**(*func[, *args][, **kwds]*)

Bind the *args* and *kwds* to the argument names of the Python function or method *func*, as if it was called with them. For bound methods, bind also the first argument (typically named `self`) to the associated instance. A dict is returned, mapping the argument names (including the names of the `*` and `**` arguments, if any) to their values from *args* and *kwds*. In case of invoking *func* incorrectly, i.e. whenever `func(*args, **kwds)` would raise an exception because of incompatible signature, an exception of the same type and the same or similar message is raised. For example:

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
```

(下页继续)

(续上页)

```
>>> getcallargs(f, 1, 2, 3)
{'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
>>> getcallargs(f, a=2, x=4)
{'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() takes at least 1 argument (0 given)
```

2.7 新版功能.

28.13.4 The interpreter stack

When the following functions return "frame records," each record is a tuple of six items: the frame object, the filename, the line number of the current line, the function name, a list of lines of context from the source code, and the index of the current line within that list.

注解: Keeping references to frame objects, as found in the first element of the frame records these functions return, can cause your program to create reference cycles. Once a reference cycle has been created, the lifespan of all objects which can be accessed from the objects which form the cycle can become much longer even if Python's optional cycle detector is enabled. If such cycles must be created, it is important to ensure they are explicitly broken to avoid the delayed destruction of objects and increased memory consumption which occurs.

Though the cycle detector will catch these, destruction of the frames (and local variables) can be made deterministic by removing the cycle in a `finally` clause. This is also important if the cycle detector was disabled when Python was compiled or using *gc.disable()*. For example:

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

The optional *context* argument supported by most of these functions specifies the number of lines of context to return, which are centered around the current line.

`inspect.`**`getframeinfo`**(*frame*[, *context*])

Get information about a frame or traceback object. A 5-tuple is returned, the last five elements of the frame's frame record.

在 2.6 版更改: Returns a *named tuple* `Traceback(filename, lineno, function, code_context, index)`.

`inspect.`**`getouterframes`**(*frame*[, *context*])

Get a list of frame records for a frame and all outer frames. These frames represent the calls that lead to the creation of *frame*. The first entry in the returned list represents *frame*; the last entry represents the outermost call on *frame*'s stack.

`inspect.`**`getinnerframes`**(*traceback*[, *context*])

Get a list of frame records for a traceback's frame and all inner frames. These frames represent calls made as a consequence of *frame*. The first entry in the list represents *traceback*; the last entry represents where the exception was raised.

inspect.**currentframe**()
Return the frame object for the caller's stack frame.

CPython implementation detail: This function relies on Python stack frame support in the interpreter, which isn't guaranteed to exist in all implementations of Python. If running in an implementation without Python stack frame support this function returns `None`.

inspect.**stack**([*context*])
Return a list of frame records for the caller's stack. The first entry in the returned list represents the caller; the last entry represents the outermost call on the stack.

inspect.**trace**([*context*])
Return a list of frame records for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller; the last entry represents where the exception was raised.

28.14 site ——指定 Site 的配置钩子

源代码: Lib/site.py

这个模块将在初始化时被自动导入。此自动导入可以通过使用解释器的 `-S` 选项来屏蔽。

Importing this module will append site-specific paths to the module search path and add a few builtins.

It starts by constructing up to four directories from a head and a tail part. For the head part, it uses `sys.prefix` and `sys.exec_prefix`; empty heads are skipped. For the tail part, it uses the empty string and then `lib/site-packages` (on Windows) or `lib/python`*X.Y*`/site-packages` and then `lib/site-python` (on Unix and Macintosh). For each of the distinct head-tail combinations, it sees if it refers to an existing directory, and if so, adds it to `sys.path` and also inspects the newly added path for configuration files.

一个路径配置文件是具有 *name*`.pth` 命名格式的文件，并且存在上面提到的四个目录之一中；它的内容是要添加到 `sys.path` 中的额外项目（每行一个）。不存在的项目不会添加到 `sys.path`，并且不会检查项目指向的是目录还是文件。项目不会被添加到 `sys.path` 超过一次。空行和由 `#` 起始的行会被跳过。以 `import` 开始的行（跟着空格或 TAB）会被执行。

在 2.6 版更改: A space or tab is now required after the import keyword.

例如，假设 `sys.prefix` 和 `sys.exec_prefix` 已经被设置为 `/usr/local`。Python X.Y 的库之后被安装为 `/usr/local/lib/python`*X.Y*。假设有一个拥有三个孙目录 `foo`, `bar` 和 `spam` 的子目录 `/usr/local/lib/python`*X.Y*`/site-packages`，并且有两个路径配置文件 `foot.pth` 和 `bar.pth`。假定 `foo.pth` 内容如下:

```
# foo package configuration

foo
bar
bletch
```

并且 `bar.pth` 包含:

```
# bar package configuration

bar
```

则下面特定版目录将以如下顺序被添加到 `sys.path`。

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

Note that `bletch` is omitted because it doesn't exist; the `bar` directory precedes the `foo` directory because `bar.pth` comes alphabetically before `foo.pth`; and `spam` is omitted because it is not mentioned in either path configuration file.

After these path manipulations, an attempt is made to import a module named `sitecustomize`, which can perform arbitrary site-specific customizations. It is typically created by a system administrator in the site-packages directory. If this import fails with an *ImportError* exception, it is silently ignored. If Python is started without output streams available, as with `pythonw.exe` on Windows (which is used by default to start IDLE), attempted output from `sitecustomize` is ignored. Any exception other than *ImportError* causes a silent and perhaps mysterious failure of the process.

After this, an attempt is made to import a module named `usercustomize`, which can perform arbitrary user-specific customizations, if *ENABLE_USER_SITE* is true. This file is intended to be created in the user site-packages directory (see below), which is part of `sys.path` unless disabled by `-s`. An *ImportError* will be silently ignored.

Note that for some non-Unix systems, `sys.prefix` and `sys.exec_prefix` are empty, and the path manipulations are skipped; however the import of `sitecustomize` and `usercustomize` is still attempted.

`site.`**PREFIXES**
A list of prefixes for site-packages directories.

2.6 新版功能.

`site.`**ENABLE_USER_SITE**
Flag showing the status of the user site-packages directory. `True` means that it is enabled and was added to `sys.path`. `False` means that it was disabled by user request (with `-s` or `PYTHONNOUSERSITE`). `None` means it was disabled for security reasons (mismatch between user or group id and effective id) or by an administrator.

2.6 新版功能.

`site.`**USER_SITE**
Path to the user site-packages for the running Python. Can be `None` if *getusersitepackages()* hasn't been called yet. Default value is `~/.local/lib/pythonX.Y/site-packages` for UNIX and non-framework Mac OS X builds, `~/Library/Python/X.Y/lib/python/site-packages` for Mac framework builds, and *%APPDATA%*`\Python\PythonXY\site-packages` on Windows. This directory is a site directory, which means that `.pth` files in it will be processed.

2.6 新版功能.

`site.`**USER_BASE**
Path to the base directory for the user site-packages. Can be `None` if *getuserbase()* hasn't been called yet. Default value is `~/.local` for UNIX and Mac OS X non-framework builds, `~/Library/Python/X.Y` for Mac framework builds, and *%APPDATA%*`\Python` for Windows. This value is used by Distutils to compute the installation directories for scripts, data files, Python modules, etc. for the user installation scheme. See also `PYTHONUSERBASE`.

2.6 新版功能.

`site.`**addsitedir**(*sitedir*, *known_paths=None*)
Add a directory to sys.path and process its `.pth` files. Typically used in `sitecustomize` or `usercustomize` (see above).

`site.`**getsitepackages**()
Return a list containing all global site-packages directories (and possibly site-python).

2.7 新版功能.

`site.`**getuserbase**()
Return the path of the user base directory, *USER_BASE*. If it is not initialized yet, this function will also set it, respecting `PYTHONUSERBASE`.

2.7 新版功能.

site.**getusersitepackages**()
Return the path of the user-specific site-packages directory, *USER_SITE*. If it is not initialized yet, this function will also set it, respecting PYTHONNOUSERSITE and *USER_BASE*.

2.7 新版功能.

The *site* module also provides a way to get the user directories from the command line:

```
$ python -m site --user-site
/home/user/.local/lib/python2.7/site-packages
```

If it is called without arguments, it will print the contents of *sys.path* on the standard output, followed by the value of *USER_BASE* and whether the directory exists, then the same thing for *USER_SITE*, and finally the value of *ENABLE_USER_SITE*.

--user-base
Print the path to the user base directory.

--user-site
Print the path to the user site-packages directory.

If both options are given, user base and user site will be printed (always in this order), separated by *os.pathsep*.

If any option is given, the script will exit with one of these values: 0 if the user site-packages directory is enabled, 1 if it was disabled by the user, 2 if it is disabled for security reasons or by an administrator, and a value greater than 2 if there is an error.

参见:

PEP 370 –分用户的 site-packages 目录

28.15 user —User-specific configuration hook

2.6 版后已移除: The *user* module has been removed in Python 3.

As a policy, Python doesn't run user-specified code on startup of Python programs. (Only interactive sessions execute the script specified in the PYTHONSTARTUP environment variable if it exists).

However, some programs or sites may find it convenient to allow users to have a standard customization file, which gets run when a program requests it. This module implements such a mechanism. A program that wishes to use the mechanism must execute the statement

```
import user
```

The *user* module looks for a file .pythonrc.py in the user's home directory and if it can be opened, executes it (using *execfile()*) in its own (the module *user*'s) global namespace. Errors during this phase are not caught; that's up to the program that imports the *user* module, if it wishes. The home directory is assumed to be named by the HOME environment variable; if this is not set, the current directory is used.

The user's .pythonrc.py could conceivably test for sys.version if it wishes to do different things depending on the Python version.

A warning to users: be very conservative in what you place in your .pythonrc.py file. Since you don't know which programs will use it, changing the behavior of standard modules or functions is generally not a good idea.

A suggestion for programmers who wish to use this mechanism: a simple way to let users specify options for your package is to have them define variables in their .pythonrc.py file that you test in your module. For example, a module spam that has a verbosity level can look for a variable user.spam_verbose, as follows:

```
import user

verbose = bool(getattr(user, "spam_verbose", 0))
```

(The three-argument form of *getattr()* is used in case the user has not defined `spam_verbose` in their `.pythonrc.py` file.)

Programs with extensive customization needs are better off reading a program-specific customization file.

Programs with security or privacy concerns should *not* import this module; a user can easily break into a program by placing arbitrary code in the `.pythonrc.py` file.

Modules for general use should *not* import this module; it may interfere with the operation of the importing program.

参见:

Module ***site*** Site-wide customization mechanism.

28.16 `fpectl` —Floating point exception control

注解: The *fpectl* module is not built by default, and its usage is discouraged and may be dangerous except in the hands of experts. See also the section *Limitations and other considerations* on limitations for more details.

Most computers carry out floating point operations in conformance with the so-called IEEE-754 standard. On any real computer, some floating point operations produce results that cannot be expressed as a normal floating point value. For example, try

```
>>> import math
>>> math.exp(1000)
inf
>>> math.exp(1000) / math.exp(1000)
nan
```

(The example above will work on many platforms. DEC Alpha may be one exception.) "Inf" is a special, non-numeric value in IEEE-754 that stands for "infinity", and "nan" means "not a number." Note that, other than the non-numeric results, nothing special happened when you asked Python to carry out those calculations. That is in fact the default behaviour prescribed in the IEEE-754 standard, and if it works for you, stop reading now.

In some circumstances, it would be better to raise an exception and stop processing at the point where the faulty operation was attempted. The *fpectl* module is for use in that situation. It provides control over floating point units from several hardware manufacturers, allowing the user to turn on the generation of `SIGFPE` whenever any of the IEEE-754 exceptions Division by Zero, Overflow, or Invalid Operation occurs. In tandem with a pair of wrapper macros that are inserted into the C code comprising your python system, `SIGFPE` is trapped and converted into the Python *FloatingPointError* exception.

The *fpectl* module defines the following functions and may raise the given exception:

`fpectl.`**`turnon_sigfpe`**`()`
Turn on the generation of `SIGFPE`, and set up an appropriate signal handler.

`fpectl.`**`turnoff_sigfpe`**`()`
Reset default handling of floating point exceptions.

exception `fpectl.`**`FloatingPointError`**
After *turnon_sigfpe()* has been executed, a floating point operation that raises one of the IEEE-754 exceptions Division by Zero, Overflow, or Invalid operation will in turn raise this standard Python exception.

28.16.1 Example

The following example demonstrates how to start up and test operation of the *fpectl* module.

```
>>> import fpectl
>>> import fpetest
>>> fpectl.turnon_sigfpe()
>>> fpetest.test()
overflow        PASS
FloatingPointError: Overflow

div by 0        PASS
FloatingPointError: Division by zero
  [ more output from test elided ]
>>> import math
>>> math.exp(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FloatingPointError: in math_1
```

28.16.2 Limitations and other considerations

Setting up a given processor to trap IEEE-754 floating point errors currently requires custom code on a per-architecture basis. You may have to modify *fpectl* to control your particular hardware.

Conversion of an IEEE-754 exception to a Python exception requires that the wrapper macros `PyFPE_START_PROTECT` and `PyFPE_END_PROTECT` be inserted into your code in an appropriate fashion. Python itself has been modified to support the *fpectl* module, but many other codes of interest to numerical analysts have not.

The *fpectl* module is not thread-safe.

参见:

Some files in the source distribution may be interesting in learning more about how this module operates. The include file Include/pyfpe.h discusses the implementation of this module at some length. Modules/fpetestmodule.c gives several examples of use. Many additional examples can be found in Objects/floatobject.c.

CHAPTER 29

自定义 Python 解释器

本章中描述的模块允许编写类似于 Python 的交互式解释器的接口。如果你想要一个支持附加一些特殊功能到 Python 语言的 Python 解释器，你应该看看*code* 模块。(*codeop* 模块是低层级的，用于支持编译可能不完整的 Python 代码块。)

本章描述的完整模块列表如下：

29.1 code —解释器基础类

code 模块提供了在 Python 中实现 read-eval-print 循环的功能。它包含两个类和一些快捷功能，可用于构建提供交互式解释器的应用程序。

class code.**InteractiveInterpreter**([*locals*])
这个类处理解析器和解释器状态（用户命名空间的)；它不处理缓冲器、终端提示区或着输入文件名(文件名总是显示地传递)。可选的 *locals* 参数指定一个字典，字典里面包含将在此类执行的代码；它默认创建新的字典，其键 '__name__' 设置为 '__console__'，键 '__doc__' 设置为 None 。

class code.**InteractiveConsole**([*locals*[, *filename*]])
尽可能模拟交互式 Python 解释器的行为。此类建立在*InteractiveInterpreter* 的基础上，使用熟悉的 sys.ps1 和 sys.ps2 作为输入提示符，并有输入缓冲。

code.**interact**([*banner*[, *readfunc*[, *local*]]])
Convenience function to run a read-eval-print loop. This creates a new instance of *InteractiveConsole* and sets *readfunc* to be used as the *InteractiveConsole.raw_input()* method, if provided. If *local* is provided, it is passed to the *InteractiveConsole* constructor for use as the default namespace for the interpreter loop. The *interact()* method of the instance is then run with *banner* passed as the banner to use, if provided. The console object is discarded after use.

code.**compile_command**(*source*[, *filename*[, *symbol*]])
这个函数主要用来模拟 Python 解释器的主循环（即 read-eval-print 循环)。难点的部分是当用户输入不完整命令时，判断能否通过之后的输入来完成（要么成为完整的命令，要么语法错误)。该函数 几乎和实际的解释器主循环的判断是相同的。

source is the source string; *filename* is the optional filename from which source was read, defaulting to `'<input>'`; and *symbol* is the optional grammar start symbol, which should be either `'single'` (the default) or `'eval'`.

如果命令完整且有效则返回一个代码对象(等价于 `compile(source, filename, symbol)`)；如果命令不完整则返回 `None`；如果命令完整但包含语法错误则会引发*SyntaxError* 或*OverflowError* 而如果命令包含无效字面值则将引发*ValueError*。

29.1.1 交互解释器对象

`InteractiveInterpreter.`**`runsource`**(*source*[, *filename*[, *symbol*]])

Compile and run some source in the interpreter. Arguments are the same as for *compile_command()*; the default for *filename* is `'<input>'`, and for *symbol* is `'single'`. One several things can happen:

- 输入不正确；*compile_command()* 引发了一个异常 (*SyntaxError* 或*OverflowError*)。将通过调用*showsyntaxerror()* 方法打印语法回溯信息。*runsource()* 返回 `False`。
- 输入不完整，需要更多输入；函数*compile_command()* 返回 `None` 。方法*runsource()* 返回 `True` 。
- 输入完整；*compile_command()* 返回了一个代码对象。将通过调用*runcode()* 执行代码（该方法也会处理运行时异常，*SystemExit* 除外)。*runsource()* 返回 `False`。

该返回值用于决定使用 `sys.ps1` 还是 `sys.ps2` 来作为下一行的输入提示符。

`InteractiveInterpreter.`**`runcode`**(*code*)

执行一个代码对象。当发生异常时，调用*showtraceback()* 来显示回溯。除*SystemExit* （允许传播）以外的所有异常都会被捕获。

有关*KeyboardInterrupt* 的说明，该异常可能发生于此代码的其他位置，并且并不总能被捕获。调用者应当准备好处理它。

`InteractiveInterpreter.`**`showsyntaxerror`**([*filename*])

显示刚发生的语法错误。这不会显示堆栈回溯因为语法错误并无此种信息。如果给出了 *filename*，它会被放入异常来替代 Python 解析器所提供的默认文件名，因为它在从一个字符串读取时总是会使用 `'<string>'`。输出将由*write()* 方法来写入。

`InteractiveInterpreter.`**`showtraceback`**()

显示刚发生的异常。我们移除了第一个堆栈条目因为它从属于解释器对象的实现。输出将由*write()* 方法来写入。

`InteractiveInterpreter.`**`write`**(*data*)

将一个字符串写入到标准错误流 (`sys.stderr`)。所有派生类都应重载此方法以提供必要的正确输出处理。

29.1.2 交互式控制台对象

InteractiveConsole 类是*InteractiveInterpreter* 的子类，因此它提供了解释器对象的所有方法，还有以下的额外方法。

`InteractiveConsole.`**`interact`**([*banner*])

Closely emulate the interactive Python console. The optional banner argument specify the banner to print before the first interaction; by default it prints a banner similar to the one printed by the standard Python interpreter, followed by the class name of the console object in parentheses (so as not to confuse this with the real interpreter –since it’s so close!).

`InteractiveConsole.`**`push`**(*line*)

将一行源文本推入解释器。行内容不应带有末尾换行符；它可以有内部换行符。行内容会被添加到一个缓冲区并且会调用解释器的 `runsource()` 方法，附带缓冲区内容的拼接结果作为源文本。如果显示

命令已执行或不合法，缓冲区将被重置；否则，则命令尚未结束，缓冲区将在添加行后保持原样。如果要求更多输入则返回值为 `True`，如果行已按某种方式被处理则返回值为 `False` (这与 `runsource()` 相同)。

`InteractiveConsole.`**`resetbuffer`**`()`
从输入缓冲区中删除所有未处理的内容。

`InteractiveConsole.`**`raw_input`**`(`[*prompt*]`)`
Write a prompt and read a line. The returned line does not include the trailing newline. When the user enters the EOF key sequence, *EOFError* is raised. The base implementation uses the built-in function *raw_input()*; a subclass may replace this with a different implementation.

29.2 codeop —编译 Python 代码

codeop 模块提供了可以模拟 Python 读取-执行-打印循环的实用程序，就像在*code* 模块中一样。因此，您可能不希望直接使用该模块；如果你想在程序中包含这样一个循环，你可能需要使用*code* 模块。

这个任务有两个部分:

1. 能够判断一行输入是否完成了一个 Python 语句：简而言之，告诉我们是否要打印‘`>>>`’或‘`...`’。
2. 记住用户已输入了哪些 future 语句，这样后续的输入可以在这些语句被启用的状态下被编译。

codeop 模块提供了分别以及同时执行这两个部分的方式。

只执行前一部分:

`codeop.`**`compile_command`**`(`*source*[, *filename*[, *symbol*]]`)`
尝试编译 *source*，这应当是一个 Python 代码字符串，并且在 *source* 是有效的 Python 代码时返回一个代码对象。在此情况下，代码对象的 filename 属性将为 *filename*，其默认值为 `'<input>'`。如果 *source* 不是有效的 Python 代码而是有效的 Python 代码的一个前缀时将返回 `None`。

如果 *source* 存在问题，将引发异常。如果存在无效的 Python 语法将引发*SyntaxError*，而如果存在无效的字面值则将引发*OverflowError* 或*ValueError*。

The *symbol* argument determines whether *source* is compiled as a statement (`'single'`, the default) or as an *expression* (`'eval'`). Any other value will cause *ValueError* to be raised.

注解: 解析器有可能（但很不常见）会在到达源码结尾之前停止解析并成功输出结果；在这种情况下，末尾的符号可能会被忽略而不是引发错误。例如，一个反斜杠加两个换行符之后可以跟随任何无意义的符号。一旦解析器 API 得到改进将修正这个问题。

---

class codeop.**Compile**
这个类的实例具有 `__call__()` 方法，其签名与内置函数*compile()* 相似，区别在于如果该实例编译了包含*__future__* 语句的程序文本，则实例会‘记住’并使用已生效的语句编译所有后续程序文本。

class codeop.**CommandCompiler**
这个类的实例具有 `__call__()` 方法，其签名与*compile_command()* 相似；区别在于如果该实例编译了包含 `__future__` 语句的程序文本，则实例会‘记住’并使用已生效的语句编译编译所有后续程序文本。

A note on version compatibility: the *Compile* and *CommandCompiler* are new in Python 2.2. If you want to enable the future-tracking features of 2.2 but also retain compatibility with 2.1 and earlier versions of Python you can either write

```
try:
    from codeop import CommandCompiler
    compile_command = CommandCompiler()
    del CommandCompiler
except ImportError:
    from codeop import compile_command
```

which is a low-impact change, but introduces possibly unwanted global state into your program, or you can write:

```
try:
    from codeop import CommandCompiler
except ImportError:
    def CommandCompiler():
        from codeop import compile_command
        return compile_command
```

and then call `CommandCompiler` every time you need a fresh compiler object.

CHAPTER 30

Restricted Execution

警告： In Python 2.3 these modules have been disabled due to various known and not readily fixable security holes. The modules are still documented here to help in reading old code that uses the *rexec* and *Bastion* modules.

Restricted execution is the basic framework in Python that allows for the segregation of trusted and untrusted code. The framework is based on the notion that trusted Python code (a *supervisor*) can create a "padded cell" (or environment) with limited permissions, and run the untrusted code within this cell. The untrusted code cannot break out of its cell, and can only interact with sensitive system resources through interfaces defined and managed by the trusted code. The term "restricted execution" is favored over "safe-Python" since true safety is hard to define, and is determined by the way the restricted environment is created. Note that the restricted environments can be nested, with inner cells creating subcells of lesser, but never greater, privilege.

An interesting aspect of Python's restricted execution model is that the interfaces presented to untrusted code usually have the same names as those presented to trusted code. Therefore no special interfaces need to be learned to write code designed to run in a restricted environment. And because the exact nature of the padded cell is determined by the supervisor, different restrictions can be imposed, depending on the application. For example, it might be deemed "safe" for untrusted code to read any file within a specified directory, but never to write a file. In this case, the supervisor may redefine the built-in *open()* function so that it raises an exception whenever the *mode* parameter is `'w'`. It might also perform a `chroot()`-like operation on the *filename* parameter, such that root is always relative to some safe "sandbox" area of the filesystem. In this case, the untrusted code would still see a built-in *open()* function in its environment, with the same calling interface. The semantics would be identical too, with *IOError*s being raised when the supervisor determined that an unallowable parameter is being used.

The Python run-time determines whether a particular code block is executing in restricted execution mode based on the identity of the `__builtins__` object in its global variables: if this is (the dictionary of) the standard *__builtin__* module, the code is deemed to be unrestricted, else it is deemed to be restricted.

Python code executing in restricted mode faces a number of limitations that are designed to prevent it from escaping from the padded cell. For instance, the function object attribute `func_globals` and the class and instance object attribute *__dict__* are unavailable.

Two modules provide the framework for setting up restricted execution environments:

30.1 rexec —Restricted execution framework

2.6 版后已移除: The *rexec* module has been removed in Python 3.

在 2.3 版更改: Disabled module.

警告: The documentation has been left in place to help in reading old code that uses the module.

This module contains the *RExec* class, which supports `r_eval()`, `r_execfile()`, `r_exec()`, and `r_import()` methods, which are restricted versions of the standard Python functions *eval()*, *execfile()* and the `exec` and `import` statements. Code executed in this restricted environment will only have access to modules and functions that are deemed safe; you can subclass *RExec* to add or remove capabilities as desired.

警告: While the *rexec* module is designed to perform as described below, it does have a few known vulnerabilities which could be exploited by carefully written code. Thus it should not be relied upon in situations requiring "production ready" security. In such situations, execution via sub-processes or very careful "cleansing" of both code and data to be processed may be necessary. Alternatively, help in patching known *rexec* vulnerabilities would be welcomed.

注解: The *RExec* class can prevent code from performing unsafe operations like reading or writing disk files, or using TCP/IP sockets. However, it does not protect against code using extremely large amounts of memory or processor time.

class rexec.**RExec**([*hooks*[, *verbose*]])

Returns an instance of the *RExec* class.

hooks is an instance of the `RHooks` class or a subclass of it. If it is omitted or `None`, the default `RHooks` class is instantiated. Whenever the *rexec* module searches for a module (even a built-in one) or reads a module's code, it doesn't actually go out to the file system itself. Rather, it calls methods of an `RHooks` instance that was passed to or created by its constructor. (Actually, the *RExec* object doesn't make these calls —they are made by a module loader object that's part of the *RExec* object. This allows another level of flexibility, which can be useful when changing the mechanics of `import` within the restricted environment.)

By providing an alternate `RHooks` object, we can control the file system accesses made to import a module, without changing the actual algorithm that controls the order in which those accesses are made. For instance, we could substitute an `RHooks` object that passes all filesystem requests to a file server elsewhere, via some RPC mechanism such as ILU. Grail's applet loader uses this to support importing applets from a URL for a directory.

If *verbose* is true, additional debugging output may be sent to standard output.

It is important to be aware that code running in a restricted environment can still call the *sys.exit()* function. To disallow restricted code from exiting the interpreter, always protect calls that cause restricted code to run with a `try`/`except` statement that catches the *SystemExit* exception. Removing the *sys.exit()* function from the restricted environment is not sufficient —the restricted code could still use `raise SystemExit`. Removing *SystemExit* is not a reasonable option; some library code makes use of this and would break were it not available.

参见:

Grail Home Page Grail is a Web browser written entirely in Python. It uses the *rexec* module as a foundation for supporting Python applets, and can be used as an example usage of this module.

30.1.1 RExec Objects

RExec instances support the following methods:

RExec.**r_eval**(*code*)
code must either be a string containing a Python expression, or a compiled code object, which will be evaluated in the restricted environment's *__main__* module. The value of the expression or code object will be returned.

RExec.**r_exec**(*code*)
code must either be a string containing one or more lines of Python code, or a compiled code object, which will be executed in the restricted environment's *__main__* module.

RExec.**r_execfile**(*filename*)
Execute the Python code contained in the file *filename* in the restricted environment's *__main__* module.

Methods whose names begin with `s_` are similar to the functions beginning with `r_`, but the code will be granted access to restricted versions of the standard I/O streams `sys.stdin`, `sys.stderr`, and `sys.stdout`.

RExec.**s_eval**(*code*)
code must be a string containing a Python expression, which will be evaluated in the restricted environment.

RExec.**s_exec**(*code*)
code must be a string containing one or more lines of Python code, which will be executed in the restricted environment.

RExec.**s_execfile**(*code*)
Execute the Python code contained in the file *filename* in the restricted environment.

RExec objects must also support various methods which will be implicitly called by code executing in the restricted environment. Overriding these methods in a subclass is used to change the policies enforced by a restricted environment.

RExec.**r_import**(*modulename*[, *globals*[, *locals*[, *fromlist*]]])
Import the module *modulename*, raising an *ImportError* exception if the module is considered unsafe.

RExec.**r_open**(*filename*[, *mode*[, *bufsize*]])
Method called when *open()* is called in the restricted environment. The arguments are identical to those of *open()*, and a file object (or a class instance compatible with file objects) should be returned. *RExec*'s default behaviour is allow opening any file for reading, but forbidding any attempt to write a file. See the example below for an implementation of a less restrictive *r_open()*.

RExec.**r_reload**(*module*)
Reload the module object *module*, re-parsing and re-initializing it.

RExec.**r_unload**(*module*)
Unload the module object *module* (remove it from the restricted environment's `sys.modules` dictionary).

And their equivalents with access to restricted standard I/O streams:

RExec.**s_import**(*modulename*[, *globals*[, *locals*[, *fromlist*]]])
Import the module *modulename*, raising an *ImportError* exception if the module is considered unsafe.

RExec.**s_reload**(*module*)
Reload the module object *module*, re-parsing and re-initializing it.

RExec.**s_unload**(*module*)
Unload the module object *module*.

30.1.2 Defining restricted environments

The *RExec* class has the following class attributes, which are used by the `__init__()` method. Changing them on an existing instance won't have any effect; instead, create a subclass of *RExec* and assign them new values in the class definition. Instances of the new class will then use those new values. All these attributes are tuples of strings.

`RExec.`**nok_builtin_names**
Contains the names of built-in functions which will *not* be available to programs running in the restricted environment. The value for *RExec* is `('open', 'reload', '__import__')`. (This gives the exceptions, because by far the majority of built-in functions are harmless. A subclass that wants to override this variable should probably start with the value from the base class and concatenate additional forbidden functions —when new dangerous built-in functions are added to Python, they will also be added to this module.)

`RExec.`**ok_builtin_modules**
Contains the names of built-in modules which can be safely imported. The value for *RExec* is `('audioop', 'array', 'binascii', 'cmath', 'errno', 'imageop', 'marshal', 'math', 'md5', 'operator', 'parser', 'regex', 'select', 'sha', '_sre', 'strop', 'struct', 'time')`. A similar remark about overriding this variable applies —use the value from the base class as a starting point.

`RExec.`**ok_path**
Contains the directories which will be searched when an `import` is performed in the restricted environment. The value for *RExec* is the same as `sys.path` (at the time the module is loaded) for unrestricted code.

`RExec.`**ok_posix_names**
Contains the names of the functions in the *os* module which will be available to programs running in the restricted environment. The value for *RExec* is `('error', 'fstat', 'listdir', 'lstat', 'readlink', 'stat', 'times', 'uname', 'getpid', 'getppid', 'getcwd', 'getuid', 'getgid', 'geteuid', 'getegid')`.

`RExec.`**ok_sys_names**
Contains the names of the functions and variables in the *sys* module which will be available to programs running in the restricted environment. The value for *RExec* is `('ps1', 'ps2', 'copyright', 'version', 'platform', 'exit', 'maxint')`.

`RExec.`**ok_file_types**
Contains the file types from which modules are allowed to be loaded. Each file type is an integer constant defined in the *imp* module. The meaningful values are `PY_SOURCE`, `PY_COMPILED`, and `C_EXTENSION`. The value for *RExec* is `(C_EXTENSION, PY_SOURCE)`. Adding `PY_COMPILED` in subclasses is not recommended; an attacker could exit the restricted execution mode by putting a forged byte-compiled file (`.pyc`) anywhere in your file system, for example by writing it to `/tmp` or uploading it to the `/incoming` directory of your public FTP server.

30.1.3 An example

Let us say that we want a slightly more relaxed policy than the standard *RExec* class. For example, if we're willing to allow files in `/tmp` to be written, we can subclass the *RExec* class:

```
class TmpWriterRExec(rexec.RExec):
    def r_open(self, file, mode='r', buf=-1):
        if mode in ('r', 'rb'):
            pass
        elif mode in ('w', 'wb', 'a', 'ab'):
            # check filename: must begin with /tmp/
            if file[:5]!='/tmp/':
                raise IOError("can't write outside /tmp")
```

(下页继续)

(续上页)

```
            elif (string.find(file, '/../') >= 0 or
                  file[:3] == '../' or file[-3:] == '/..'):
                raise IOError("'..' in filename forbidden")
        else: raise IOError("Illegal open() mode")
        return open(file, mode, buf)
```

Notice that the above code will occasionally forbid a perfectly valid filename; for example, code in the restricted environment won't be able to open a file called `/tmp/foo/../bar`. To fix this, the `r_open()` method would have to simplify the filename to `/tmp/bar`, which would require splitting apart the filename and performing various operations on it. In cases where security is at stake, it may be preferable to write simple code which is sometimes overly restrictive, instead of more general code that is also more complex and may harbor a subtle security hole.

30.2 `Bastion` —Restricting access to objects

2.6 版后已移除: The *Bastion* module has been removed in Python 3.

在 2.3 版更改: Disabled module.

注解: The documentation has been left in place to help in reading old code that uses the module.

According to the dictionary, a bastion is "a fortified area or position", or "something that is considered a stronghold." It's a suitable name for this module, which provides a way to forbid access to certain attributes of an object. It must always be used with the *rexec* module, in order to allow restricted-mode programs access to certain safe attributes of an object, while denying access to other, unsafe attributes.

`Bastion.`**`Bastion`**(*object*[, *filter*[, *name*[, *class*]]])

Protect the object *object*, returning a bastion for the object. Any attempt to access one of the object's attributes will have to be approved by the *filter* function; if the access is denied an *AttributeError* exception will be raised.

If present, *filter* must be a function that accepts a string containing an attribute name, and returns true if access to that attribute will be permitted; if *filter* returns false, the access is denied. The default filter denies access to any function beginning with an underscore (`'_'`). The bastion's string representation will be `<Bastion for name>` if a value for *name* is provided; otherwise, `repr(object)` will be used.

class, if present, should be a subclass of *BastionClass*; see the code in `bastion.py` for the details. Overriding the default *BastionClass* will rarely be required.

`class` `Bastion.`**`BastionClass`**(*getfunc*, *name*)

Class which actually implements bastion objects. This is the default class used by *Bastion()*. The *getfunc* parameter is a function which returns the value of an attribute which should be exposed to the restricted execution environment when called with the name of the attribute as the only parameter. *name* is used to construct the *repr()* of the *BastionClass* instance.

参见:

Grail Home Page Grail, an Internet browser written in Python, uses these modules to support Python applets. More information on the use of Python's restricted execution mode in Grail is available on the Web site.

CHAPTER 31

导入模块

本章中介绍的模块提供了导入其他 Python 模块和挂钩以自定义导入过程的新方法。

本章描述的完整模块列表如下：

31.1 imp —Access the import internals

This module provides an interface to the mechanisms used to implement the `import` statement. It defines the following constants and functions:

`imp.`**`get_magic`**`()`
Return the magic string value used to recognize byte-compiled code files (`.pyc` files). (This value may be different for each Python version.)

`imp.`**`get_suffixes`**`()`
Return a list of 3-element tuples, each describing a particular type of module. Each triple has the form `(suffix, mode, type)`, where *suffix* is a string to be appended to the module name to form the filename to search for, *mode* is the mode string to pass to the built-in *open()* function to open the file (this can be `'r'` for text files or `'rb'` for binary files), and *type* is the file type, which has one of the values *PY_SOURCE*, *PY_COMPILED*, or *C_EXTENSION*, described below.

`imp.`**`find_module`**`(`*name*[, *path*]`)`
Try to find the module *name*. If *path* is omitted or `None`, the list of directory names given by `sys.path` is searched, but first a few special places are searched: the function tries to find a built-in module with the given name (*C_BUILTIN*), then a frozen module (*PY_FROZEN*), and on some systems some other places are looked in as well (on Windows, it looks in the registry which may point to a specific file).

Otherwise, *path* must be a list of directory names; each directory is searched for files with any of the suffixes returned by *get_suffixes()* above. Invalid names in the list are silently ignored (but all list items must be strings).

If search is successful, the return value is a 3-element tuple `(file, pathname, description)`:

file is an open file object positioned at the beginning, *pathname* is the pathname of the file found, and *description* is a 3-element tuple as contained in the list returned by *get_suffixes()* describing the kind of module found.

If the module does not live in a file, the returned *file* is None, *pathname* is the empty string, and the *description* tuple contains empty strings for its suffix and mode; the module type is indicated as given in parentheses above. If the search is unsuccessful, *ImportError* is raised. Other exceptions indicate problems with the arguments or environment.

If the module is a package, *file* is None, *pathname* is the package path and the last item in the *description* tuple is *PKG_DIRECTORY*.

This function does not handle hierarchical module names (names containing dots). In order to find *P.M*, that is, submodule *M* of package *P*, use *find_module()* and *load_module()* to find and load package *P*, and then use *find_module()* with the *path* argument set to `P.__path__`. When *P* itself has a dotted name, apply this recipe recursively.

imp.**load_module**(*name*, *file*, *pathname*, *description*)

Load a module that was previously found by *find_module()* (or by an otherwise conducted search yielding compatible results). This function does more than importing the module: if the module was already imported, it is equivalent to a *reload()*! The *name* argument indicates the full module name (including the package name, if this is a submodule of a package). The *file* argument is an open file, and *pathname* is the corresponding file name; these can be None and '', respectively, when the module is a package or not being loaded from a file. The *description* argument is a tuple, as would be returned by *get_suffixes()*, describing what kind of module must be loaded.

If the load is successful, the return value is the module object; otherwise, an exception (usually *ImportError*) is raised.

Important: the caller is responsible for closing the *file* argument, if it was not None, even when an exception is raised. This is best done using a `try ··· finally` statement.

imp.**new_module**(*name*)

Return a new empty module object called *name*. This object is *not* inserted in `sys.modules`.

imp.**lock_held**()

Return True if the import lock is currently held, else False. On platforms without threads, always return False.

On platforms with threads, a thread executing an import holds an internal lock until the import is complete. This lock blocks other threads from doing an import until the original import completes, which in turn prevents other threads from seeing incomplete module objects constructed by the original thread while in the process of completing its import (and the imports, if any, triggered by that).

imp.**acquire_lock**()

Acquire the interpreter's import lock for the current thread. This lock should be used by import hooks to ensure thread-safety when importing modules.

Once a thread has acquired the import lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

On platforms without threads, this function does nothing.

2.3 新版功能.

imp.**release_lock**()

Release the interpreter's import lock. On platforms without threads, this function does nothing.

2.3 新版功能.

The following constants with integer values, defined in this module, are used to indicate the search result of *find_module()*.

imp.**PY_SOURCE**

The module was found as a source file.

imp.**PY_COMPILED**
The module was found as a compiled code object file.

imp.**C_EXTENSION**
The module was found as dynamically loadable shared library.

imp.**PKG_DIRECTORY**
The module was found as a package directory.

imp.**C_BUILTIN**
The module was found as a built-in module.

imp.**PY_FROZEN**
The module was found as a frozen module (see *init_frozen()*).

The following constant and functions are obsolete; their functionality is available through *find_module()* or *load_module()*. They are kept around for backward compatibility:

imp.**SEARCH_ERROR**
Unused.

imp.**init_builtin**(*name*)
Initialize the built-in module called *name* and return its module object along with storing it in `sys.modules`. If the module was already initialized, it will be initialized *again*. Re-initialization involves the copying of the built-in module's `__dict__` from the cached module over the module's entry in `sys.modules`. If there is no built-in module called *name*, `None` is returned.

imp.**init_frozen**(*name*)
Initialize the frozen module called *name* and return its module object. If the module was already initialized, it will be initialized *again*. If there is no frozen module called *name*, `None` is returned. (Frozen modules are modules written in Python whose compiled byte-code object is incorporated into a custom-built Python interpreter by Python's **freeze** utility. See `Tools/freeze/` for now.)

imp.**is_builtin**(*name*)
Return `1` if there is a built-in module called *name* which can be initialized again. Return `-1` if there is a built-in module called *name* which cannot be initialized again (see *init_builtin()*). Return `0` if there is no built-in module called *name*.

imp.**is_frozen**(*name*)
Return `True` if there is a frozen module (see *init_frozen()*) called *name*, or `False` if there is no such module.

imp.**load_compiled**(*name*, *pathname*[, *file*])
Load and initialize a module implemented as a byte-compiled code file and return its module object. If the module was already initialized, it will be initialized *again*. The *name* argument is used to create or access a module object. The *pathname* argument points to the byte-compiled code file. The *file* argument is the byte-compiled code file, open for reading in binary mode, from the beginning. It must currently be a real file object, not a user-defined class emulating a file.

imp.**load_dynamic**(*name*, *pathname*[, *file*])
Load and initialize a module implemented as a dynamically loadable shared library and return its module object. If the module was already initialized, it will be initialized *again*. Re-initialization involves copying the `__dict__` attribute of the cached instance of the module over the value used in the module cached in `sys.modules`. The *pathname* argument must point to the shared library. The *name* argument is used to construct the name of the initialization function: an external C function called `initname()` in the shared library is called. The optional *file* argument is ignored. (Note: using shared libraries is highly system dependent, and not all systems support it.)

CPython implementation detail: The import internals identify extension modules by filename, so doing `foo = load_dynamic("foo", "mod.so")` and `bar = load_dynamic("bar", "mod.so")` will result in both foo and bar referring to the same module, regardless of whether or not `mod.so` exports an `initbar`

function. On systems which support them, symlinks can be used to import multiple modules from the same shared library, as each reference to the module will use a different file name.

imp.**load_source**(*name*, *pathname*[, *file*])
Load and initialize a module implemented as a Python source file and return its module object. If the module was already initialized, it will be initialized *again*. The *name* argument is used to create or access a module object. The *pathname* argument points to the source file. The *file* argument is the source file, open for reading as text, from the beginning. It must currently be a real file object, not a user-defined class emulating a file. Note that if a properly matching byte-compiled file (with suffix `.pyc` or `.pyo`) exists, it will be used instead of parsing the given source file.

class imp.**NullImporter**(*path_string*)
The *NullImporter* type is a **PEP 302** import hook that handles non-directory path strings by failing to find any modules. Calling this type with an existing directory or empty string raises *ImportError*. Otherwise, a *NullImporter* instance is returned.

Python adds instances of this type to `sys.path_importer_cache` for any path entries that are not directories and are not handled by any other path hooks on `sys.path_hooks`. Instances have only one method:

find_module(*fullname*[, *path*])
This method always returns `None`, indicating that the requested module could not be found.

2.5 新版功能.

31.1.1 例子

The following function emulates what was the standard import statement up to Python 1.4 (no hierarchical module names). (This *implementation* wouldn't work in that version, since *find_module()* has been extended and *load_module()* has been added in 1.4.)

```
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    try:
        return sys.modules[name]
    except KeyError:
        pass

    # If any of the following calls raises an exception,
    # there's a problem we can't handle -- let the caller handle it.

    fp, pathname, description = imp.find_module(name)

    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        # Since we may exit via an exception, close fp explicitly.
        if fp:
            fp.close()
```

A more complete example that implements hierarchical module names and includes a *reload()* function can be found in the module `knee`. The `knee` module can be found in `Demo/imputil/` in the Python source distribution.

31.2 importlib —Convenience wrappers for __import__()

2.7 新版功能.

This module is a minor subset of what is available in the more full-featured package of the same name from Python 3.1 that provides a complete implementation of `import`. What is here has been provided to help ease in transitioning from 2.7 to 3.1.

`importlib.`**`import_module`**(*name*, *package=None*)

Import a module. The *name* argument specifies what module to import in absolute or relative terms (e.g. either `pkg.mod` or `..mod`). If the name is specified in relative terms, then the *package* argument must be specified to the package which is to act as the anchor for resolving the package name (e.g. `import_module('..mod', 'pkg.subpkg')` will import `pkg.mod`). The specified module will be inserted into *sys.modules* and returned.

31.3 imputil —Import utilities

2.6 版后已移除: The *imputil* module has been removed in Python 3.

This module provides a very handy and useful mechanism for custom `import` hooks. Compared to the older `ihooks` module, *imputil* takes a dramatically simpler and more straight-forward approach to custom `import` functions.

class `imputil.`**`ImportManager`**([*fs_imp*])

Manage the import process.

`install`([*namespace*])

Install this ImportManager into the specified namespace.

`uninstall`()

Restore the previous import mechanism.

`add_suffix`(*suffix*, *importFunc*)

Undocumented.

class `imputil.`**`Importer`**

Base class for replacing standard import functions.

`import_top`(*name*)

Import a top-level module.

`get_code`(*parent*, *modname*, *fqname*)

Find and retrieve the code for the given module.

parent specifies a parent module to define a context for importing. It may be `None`, indicating no particular context for the search.

modname specifies a single module (not dotted) within the parent.

fqname specifies the fully-qualified module name. This is a (potentially) dotted name from the "root" of the module namespace down to the modname.

If there is no parent, then modname==fqname.

This method should return `None`, or a 3-tuple.

- If the module was not found, then `None` should be returned.
- The first item of the 2- or 3-tuple should be the integer 0 or 1, specifying whether the module that was found is a package or not.

- The second item is the code object for the module (it will be executed within the new module's namespace). This item can also be a fully-loaded module object (e.g. loaded from a shared lib).
- The third item is a dictionary of name/value pairs that will be inserted into new module before the code object is executed. This is provided in case the module's code expects certain values (such as where the module was found). When the second item is a module object, then these names/values will be inserted *after* the module has been loaded/initialized.

class imputil.**BuiltinImporter**
Emulate the import mechanism for built-in and frozen modules. This is a sub-class of the *Importer* class.

get_code(*parent*, *modname*, *fqname*)
Undocumented.

imputil.**py_suffix_importer**(*filename*, *finfo*, *fqname*)
Undocumented.

class imputil.**DynLoadSuffixImporter**([*desc*])
Undocumented.

import_file(*filename*, *finfo*, *fqname*)
Undocumented.

31.3.1 Examples

This is a re-implementation of hierarchical module import.

This code is intended to be read, not executed. However, it does work –all you need to do to enable it is "import knee".

(The name is a pun on the clunkier predecessor of this module, "ni".)

```
import sys, imp, __builtin__

# Replacement for __import__()
def import_hook(name, globals=None, locals=None, fromlist=None):
    parent = determine_parent(globals)
    q, tail = find_head_package(parent, name)
    m = load_tail(q, tail)
    if not fromlist:
        return q
    if hasattr(m, "__path__"):
        ensure_fromlist(m, fromlist)
    return m

def determine_parent(globals):
    if not globals or  not globals.has_key("__name__"):
        return None
    pname = globals['__name__']
    if globals.has_key("__path__"):
        parent = sys.modules[pname]
        assert globals is parent.__dict__
        return parent
    if '.' in pname:
        i = pname.rfind('.')
        pname = pname[:i]
        parent = sys.modules[pname]
        assert parent.__name__ == pname
        return parent
    return None
```

(下页继续)

(续上页)

```
def find_head_package(parent, name):
    if '.' in name:
        i = name.find('.')
        head = name[:i]
        tail = name[i+1:]
    else:
        head = name
        tail = ""
    if parent:
        qname = "%s.%s" % (parent.__name__, head)
    else:
        qname = head
    q = import_module(head, qname, parent)
    if q: return q, tail
    if parent:
        qname = head
        parent = None
        q = import_module(head, qname, parent)
        if q: return q, tail
    raise ImportError("No module named " + qname)

def load_tail(q, tail):
    m = q
    while tail:
        i = tail.find('.')
        if i < 0: i = len(tail)
        head, tail = tail[:i], tail[i+1:]
        mname = "%s.%s" % (m.__name__, head)
        m = import_module(head, mname, m)
        if not m:
            raise ImportError("No module named " + mname)
    return m

def ensure_fromlist(m, fromlist, recursive=0):
    for sub in fromlist:
        if sub == "*":
            if not recursive:
                try:
                    all = m.__all__
                except AttributeError:
                    pass
                else:
                    ensure_fromlist(m, all, 1)
            continue
        if sub != "*" and not hasattr(m, sub):
            subname = "%s.%s" % (m.__name__, sub)
            submod = import_module(sub, subname, m)
            if not submod:
                raise ImportError("No module named " + subname)

def import_module(partname, fqname, parent):
    try:
        return sys.modules[fqname]
    except KeyError:
        pass
```

(下页继续)

(续上页)

```
    try:
        fp, pathname, stuff = imp.find_module(partname,
                                              parent and parent.__path__)
    except ImportError:
        return None
    try:
        m = imp.load_module(fqname, fp, pathname, stuff)
    finally:
        if fp: fp.close()
    if parent:
        setattr(parent, partname, m)
    return m


# Replacement for reload()
def reload_hook(module):
    name = module.__name__
    if '.' not in name:
        return import_module(name, name, None)
    i = name.rfind('.')
    pname = name[:i]
    parent = sys.modules[pname]
    return import_module(name[i+1:], name, parent)


# Save the original hooks
original_import = __builtin__.__import__
original_reload = __builtin__.reload

# Now install our hooks
__builtin__.__import__ = import_hook
__builtin__.reload = reload_hook
```

Also see the `importers` module (which can be found in `Demo/imputil/` in the Python source distribution) for additional examples.

31.4 zipimport —从 Zip 存档中导入模块

2.3 新版功能.

This module adds the ability to import Python modules (`*.py`, `*.py[co]`) and packages from ZIP-format archives. It is usually not needed to use the *zipimport* module explicitly; it is automatically used by the built-in `import` mechanism for *sys.path* items that are paths to ZIP archives.

通常，*sys.path* 是字符串的目录名称列表。此模块同样允许*sys.path* 的一项成为命名 ZIP 文件档案的字符串。ZIP 档案可以容纳子目录结构去支持包的导入，并且可以将归档文件中的路径指定为仅从子目录导入。比如说，路径 `example.zip/lib/` 将只会从档案中的 `lib/` 子目录导入。

Any files may be present in the ZIP archive, but only files `.py` and `.py[co]` are available for import. ZIP import of dynamic modules (`.pyd`, `.so`) is disallowed. Note that if an archive only contains `.py` files, Python will not attempt to modify the archive by adding the corresponding `.pyc` or `.pyo` file, meaning that if a ZIP archive doesn't contain `.pyc` files, importing may be rather slow.

Using the built-in *reload()* function will fail if called on a module loaded from a ZIP archive; it is unlikely that *reload()* would be needed, since this would imply that the ZIP has been altered during runtime.

目前不支持带有档案注释的 ZIP 归档。

参见:

PKZIP Application Note Phil Katz 编写的 ZIP 文件格式文档，此格式和使用的算法的创建者。

PEP 273 - 从 ZIP 压缩包导入模块 由 James C. Ahlstrom 编写，他还提供了一个具体实现。Python 2.3 遵循 PEP 273 的规范，但使用了一个由 van Rossum 本人所编写的实现，该实现使用了 PEP 302 所描述的导入钩子。

PEP 302 - 新导入钩 PEP 添加导入钩来有助于模块运作。

此模块定义了一个异常:

exception zipimport.**ZipImportError**
异常由 zipimporter 对象引发。这是*ImportError* 的子类，因此，也可以捕获为*ImportError* 。

31.4.1 zipimporter 对象

zipimporter 是用于导入 ZIP 文件的类。

class zipimport.**zipimporter**(*archivepath*)
创建新的 zipimporter 实例。*archivepath* 必须是指向 ZIP 文件的路径，或者 ZIP 文件中的特定路径。例如，`foo/bar.zip/lib` 的 *archivepath* 将在 ZIP 文件 `foo/bar.zip` 中的 `lib` 目录中查找模块。

如果 *archivepath* 没有指向一个有效的 ZIP 档案，引发*ZipImportError* 。

find_module(*fullname*[, *path*])
搜索由 *fullname* 指定的模块。*fullname* 必须是完全合格的（含加点作分割的拓展名）模块名。它返回 zipimporter 实例本身如果模块被找到，或者返回*None* 如果没找到指定模块。可选的 *path* 被忽略，这是为了与导入器协议兼容。

get_code(*fullname*)
返回指定模块的代码对象。如果不能找到模块会引发*ZipImportError* 错误。

get_data(*pathname*)
Return the data associated with *pathname*. Raise *IOError* if the file wasn't found.

get_filename(*fullname*)
如果导入了指定的模块 `__file__` ，则返回为该模块设置的值。如果未找到模块则引发*ZipImportError* 错误。

2.7 新版功能.

get_source(*fullname*)
返回指定模块的源代码。如果没有找到模块则引发*ZipImportError* ，如果档案包含模块但是没有源代码，返回*None* 。

is_package(*fullname*)
如果由 *fullname* 指定的模块是一个包则返回 `True` 。如果不能找到模块则引发*ZipImportError* 错误。

load_module(*fullname*)
加载由 *fullname* 指定的模块。*fullname* 必须是完全合格的（含加点作为分割的拓展名）模块名。它返回已加载模块，或者当找不到模块时引发*ZipImportError* 错误。

archive
导入器关联的 ZIP 文件的文件名，不含可能的子路径。

prefix
ZIP 文件中搜索的模块的子路径。这是一个指向 ZIP 文件根目录的 zipimporter 对象的空字符串。

当与斜杠结合使用时，*archive* 和*prefix* 属性等价于赋予*zipimporter* 构造器的原始 *archivepath* 参数。

31.4.2 例子

这是一个从 ZIP 档案中导入模块的例子 - 请注意*zipimport* 不需要明确地使用。

```
$ unzip -l example.zip
Archive:  example.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
     8467  11-26-02 22:30   jwzthreading.py
 --------                   -------
     8467                   1 file
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip')  # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'example.zip/jwzthreading.py'
```

31.5 pkgutil —包扩展模块工具

2.3 新版功能.

源代码: Lib/pkgutil.py

该模块为导入系统提供了工具，尤其是在包支持方面。

pkgutil.**extend_path**(*path*, *name*)

Extend the search path for the modules which comprise a package. Intended use is to place the following code in a package's `__init__.py`:

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

This will add to the package's `__path__` all subdirectories of directories on `sys.path` named after the package. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

It also looks for `*.pkg` files beginning where `*` matches the *name* argument. This feature is similar to `*.pth` files (see the *site* module for more information), except that it doesn't special-case lines starting with `import`. A `*.pkg` file is trusted at face value: apart from checking for duplicates, all entries found in a `*.pkg` file are added to the path, regardless of whether they exist on the filesystem. (This is a feature.)

If the input path is not a list (as is the case for frozen packages) it is returned unchanged. The input path is not modified; an extended copy is returned. Items are only appended to the copy at the end.

It is assumed that *sys.path* is a sequence. Items of *sys.path* that are not (Unicode or 8-bit) strings referring to existing directories are ignored. Unicode items on *sys.path* that cause errors when used as filenames may cause this function to raise an exception (in line with *os.path.isdir()* behavior).

class pkgutil.**ImpImporter**(*dirname=None*)
PEP 302 Importer that wraps Python's "classic" import algorithm.

If *dirname* is a string, a PEP 302 importer is created that searches that directory. If *dirname* is None, a PEP 302 importer is created that searches the current *sys.path*, plus any modules that are frozen or built-in.

Note that *ImpImporter* does not currently support being used by placement on *sys.meta_path*.

class pkgutil.**ImpLoader**(*fullname*, *file*, *filename*, *etc*)
PEP 302 Loader that wraps Python's "classic" import algorithm.

pkgutil.**find_loader**(*fullname*)
Find a PEP 302 "loader" object for *fullname*.

If *fullname* contains dots, path must be the containing package's __path__. Returns None if the module cannot be found or imported. This function uses *iter_importers()*, and is thus subject to the same limitations regarding platform-specific special import locations such as the Windows registry.

pkgutil.**get_importer**(*path_item*)
Retrieve a PEP 302 importer for the given *path_item*.

The returned importer is cached in *sys.path_importer_cache* if it was newly created by a path hook.

If there is no importer, a wrapper around the basic import machinery is returned. This wrapper is never inserted into the importer cache (None is inserted instead).

The cache (or part of it) can be cleared manually if a rescan of *sys.path_hooks* is necessary.

pkgutil.**get_loader**(*module_or_name*)
Get a PEP 302 "loader" object for *module_or_name*.

If the module or package is accessible via the normal import mechanism, a wrapper around the relevant part of that machinery is returned. Returns None if the module cannot be found or imported. If the named module is not already imported, its containing package (if any) is imported, in order to establish the package __path__.

This function uses *iter_importers()*, and is thus subject to the same limitations regarding platform-specific special import locations such as the Windows registry.

pkgutil.**iter_importers**(*fullname=''*)
Yield PEP 302 importers for the given module name.

If fullname contains a '.', the importers will be for the package containing fullname, otherwise they will be importers for *sys.meta_path*, *sys.path*, and Python's "classic" import machinery, in that order. If the named module is in a package, that package is imported as a side effect of invoking this function.

Non-PEP 302 mechanisms (e.g. the Windows registry) used by the standard import machinery to find files in alternative locations are partially supported, but are searched *after* *sys.path*. Normally, these locations are searched *before* *sys.path*, preventing *sys.path* entries from shadowing them.

For this to cause a visible difference in behaviour, there must be a module or package name that is accessible via both *sys.path* and one of the non-PEP 302 file system mechanisms. In this case, the emulation will find the former version, while the builtin import mechanism will find the latter.

Items of the following types can be affected by this discrepancy: imp.C_EXTENSION, imp.PY_SOURCE, imp.PY_COMPILED, imp.PKG_DIRECTORY.

pkgutil.**iter_modules**(*path=None*, *prefix=''*)
Yields (module_loader, name, ispkg) for all submodules on *path*, or, if path is None, all top-level modules on sys.path.

path should be either None or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

pkgutil.**walk_packages**(*path=None*, *prefix=''*, *onerror=None*)
Yields `(module_loader, name, ispkg)` for all modules recursively on *path*, or, if path is `None`, all accessible modules.

path should be either `None` or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

Note that this function must import all *packages* (*not* all modules!) on the given *path*, in order to access the `__path__` attribute to find submodules.

onerror is a function which gets called with one argument (the name of the package which was being imported) if any exception occurs while trying to import a package. If no *onerror* function is supplied, *ImportError*s are caught and ignored, while all other exceptions are propagated, terminating the search.

示例:

```
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

pkgutil.**get_data**(*package*, *resource*)
从包中获取一个资源。

This is a wrapper for the **PEP 302** loader *get_data()* API. The *package* argument should be the name of a package, in standard module format (`foo.bar`). The *resource* argument should be in the form of a relative filename, using `/` as the path separator. The parent directory name `..` is not allowed, and nor is a rooted name (starting with a `/`).

The function returns a binary string that is the contents of the specified resource.

For packages located in the filesystem, which have already been imported, this is the rough equivalent of:

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```

If the package cannot be located or loaded, or it uses a **PEP 302** loader which does not support *get_data()*, then `None` is returned.

2.6 新版功能.

31.6 modulefinder —查找脚本使用的模块

2.3 新版功能.

源码: Lib/modulefinder.py

该模块提供了一个*ModuleFinder* 类，可用于确定脚本导入的模块集。`modulefinder.py` 也可以作为脚本运行，给出 Python 脚本的文件名作为参数，之后将打印导入模块的报告。

modulefinder.**AddPackagePath**(*pkg_name*, *path*)
记录名为 *pkg_name* 的包可以在指定的 *path* 中找到。

modulefinder.**ReplacePackage**(*oldname*, *newname*)
Allows specifying that the module named *oldname* is in fact the package named *newname*. The most common usage would be to handle how the `_xmlplus` package replaces the *xml* package.

class modulefinder.**ModuleFinder**([*path=None*, *debug=0*, *excludes=[]*, *replace_paths=[]*])
This class provides *run_script()* and *report()* methods to determine the set of modules imported by a script. *path* can be a list of directories to search for modules; if not specified, `sys.path` is used. *debug* sets the debugging level; higher values make the class print debugging messages about what it's doing. *excludes* is a list of module names to exclude from the analysis. *replace_paths* is a list of `(oldpath, newpath)` tuples that will be replaced in module paths.

report()
将报告打印到标准输出，列出脚本导入的模块及其路径，以及缺少或似乎缺失的模块。

run_script(*pathname*)
分析 *pathname* 文件的内容，该文件必须包含 Python 代码。

modules
一个将模块名称映射到模块的字典。请参阅*ModuleFinder 的示例用法*。

31.6.1 ModuleFinder 的示例用法

稍后将分析的脚本（bacon.py）：

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

将输出 bacon.py 报告的脚本:

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print 'Loaded modules:'
for name, mod in finder.modules.iteritems():
    print '%s: ' % name,
    print ','.join(mod.globalnames.keys()[:3])

print '-'*50
print 'Modules not imported:'
print '\n'.join(finder.badmodules.iterkeys())
```

输出样例（可能因架构而异）：

```
Loaded modules:
_types:
copy_reg:  _inverted_registry,_slotnames,__all__
sre_compile:  isstring,_sre,_optimize_unicode
_sre:
sre_constants:  REPEAT_ONE,makedict,AT_END_LINE
sys:
```

(下页继续)

(续上页)

```
re:  __module__,finditer,_expand
itertools:
__main__:  re,itertools,baconhameggs
sre_parse:  __getslice__,_PATTERNENDERS,SRE_FLAG_UNICODE
array:
types:  __module__,IntType,TypeType
------------------------------------------------------
Modules not imported:
guido.python.ham
baconhameggs
```

31.7 runpy —Locating and executing Python modules

2.5 新版功能.

Source code: Lib/runpy.py

The *runpy* module is used to locate and run Python modules without importing them first. Its main use is to implement the `-m` command line switch that allows scripts to be located using the Python module namespace rather than the filesystem.

The *runpy* module provides two functions:

runpy.**run_module**(*mod_name*, *init_globals=None*, *run_name=None*, *alter_sys=False*)

Execute the code of the specified module and return the resulting module globals dictionary. The module's code is first located using the standard import mechanism (refer to **PEP 302** for details) and then executed in a fresh module namespace.

If the supplied module name refers to a package rather than a normal module, then that package is imported and the `__main__` submodule within that package is then executed and the resulting module globals dictionary returned.

The optional dictionary argument *init_globals* may be used to pre-populate the module's globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by *run_module()*.

The special global variables `__name__`, `__file__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

`__name__` is set to *run_name* if this optional argument is not *None*, to `mod_name + '.__main__'` if the named module is a package and to the *mod_name* argument otherwise.

`__file__` is set to the name provided by the module loader. If the loader does not make filename information available, this variable is set to *None*.

`__loader__` is set to the **PEP 302** module loader used to retrieve the code for the module (This loader may be a wrapper around the standard import mechanism).

`__package__` is set to *mod_name* if the named module is a package and to `mod_name.rpartition('.')[0]` otherwise.

If the argument *alter_sys* is supplied and evaluates to *True*, then `sys.argv[0]` is updated with the value of `__file__` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. Both `sys.argv[0]` and `sys.modules[__name__]` are restored to their original values before the function returns.

Note that this manipulation of *sys* is not thread-safe. Other threads may see the partially initialised module, as well as the altered list of arguments. It is recommended that the *sys* module be left alone when invoking this function from threaded code.

参见:

The `-m` option offering equivalent functionality from the command line.

在 2.7 版更改: Added ability to execute packages by looking for a `__main__` submodule

`runpy.`**`run_path`**(*file_path*, *init_globals=None*, *run_name=None*)

Execute the code at the named filesystem location and return the resulting module globals dictionary. As with a script name supplied to the CPython command line, the supplied path may refer to a Python source file, a compiled bytecode file or a valid sys.path entry containing a `__main__` module (e.g. a zipfile containing a top-level `__main__.py` file).

For a simple script, the specified code is simply executed in a fresh module namespace. For a valid sys.path entry (typically a zipfile or directory), the entry is first added to the beginning of `sys.path`. The function then looks for and executes a *__main__* module using the updated path. Note that there is no special protection against invoking an existing *__main__* entry located elsewhere on `sys.path` if there is no such module at the specified location.

The optional dictionary argument *init_globals* may be used to pre-populate the module's globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by *run_path()*.

The special global variables `__name__`, `__file__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

`__name__` is set to *run_name* if this optional argument is not *None* and to `'<run_path>'` otherwise.

`__file__` is set to the name provided by the module loader. If the loader does not make filename information available, this variable is set to *None*. For a simple script, this will be set to `file_path`.

`__loader__` is set to the **PEP 302** module loader used to retrieve the code for the module (This loader may be a wrapper around the standard import mechanism). For a simple script, this will be set to *None*.

`__package__` is set to `__name__.rpartition('.')[0]`.

A number of alterations are also made to the *sys* module. Firstly, `sys.path` may be altered as described above. `sys.argv[0]` is updated with the value of `file_path` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. All modifications to items in *sys* are reverted before the function returns.

Note that, unlike *run_module()*, the alterations made to *sys* are not optional in this function as these adjustments are essential to allowing the execution of sys.path entries. As the thread-safety limitations still apply, use of this function in threaded code should be either serialised with the import lock or delegated to a separate process.

参见:

using-on-interface-options for equivalent functionality on the command line (`python path/to/script`).

2.7 新版功能.

参见:

PEP 338 –**将模块作为脚本执行** PEP 由 Nick Coghlan 撰写并实现。

PEP 366 –**Main module explicit relative imports** PEP 由 Nick Coghlan 撰写并实现。

using-on-general - CPython command line details

CHAPTER 32

Python 语言服务

Python 提供了许多模块来帮助使用 Python 语言。这些模块支持标记化、解析、语法分析、字节码反汇编以及各种其他工具。

这些模块包括：

32.1 `parser` —Access Python parse trees

The *parser* module provides an interface to Python's internal parser and byte-code compiler. The primary purpose for this interface is to allow Python code to edit the parse tree of a Python expression and create executable code from this. This is better than trying to parse and modify an arbitrary Python code fragment as a string because parsing is performed in a manner identical to the code forming the application. It is also faster.

注解： From Python 2.5 onward, it's much more convenient to cut in at the Abstract Syntax Tree (AST) generation and compilation stage, using the *ast* module.

The *parser* module exports the names documented here also with "st" replaced by "ast"; this is a legacy from the time when there was no other AST and has nothing to do with the AST found in Python 2.5. This is also the reason for the functions' keyword arguments being called *ast*, not *st*. The "ast" functions have been removed in Python 3.

There are a few things to note about this module which are important to making use of the data structures created. This is not a tutorial on editing the parse trees for Python code, but some examples of using the *parser* module are presented.

Most importantly, a good understanding of the Python grammar processed by the internal parser is required. For full information on the language syntax, refer to reference-index. The parser itself is created from a grammar specification defined in the file `Grammar/Grammar` in the standard Python distribution. The parse trees stored in the ST objects created by this module are the actual output from the internal parser when created by the *expr()* or *suite()* functions, described below. The ST objects created by *sequence2st()* faithfully simulate those structures. Be aware that the values of the sequences which are considered "correct" will vary from one version of Python to another as the formal grammar for the language is revised. However, transporting code from one Python version to another as source text will always allow correct parse trees to be created in the target version, with the only restriction being that migrating to an older

version of the interpreter will not support more recent language constructs. The parse trees are not typically compatible from one version to another, whereas source code has always been forward-compatible.

Each element of the sequences returned by *st2list()* or *st2tuple()* has a simple form. Sequences representing non-terminal elements in the grammar always have a length greater than one. The first element is an integer which identifies a production in the grammar. These integers are given symbolic names in the C header file `Include/graminit.h` and the Python module *symbol*. Each additional element of the sequence represents a component of the production as recognized in the input string: these are always sequences which have the same form as the parent. An important aspect of this structure which should be noted is that keywords used to identify the parent node type, such as the keyword `if` in an `if_stmt`, are included in the node tree without any special treatment. For example, the `if` keyword is represented by the tuple `(1, 'if')`, where `1` is the numeric value associated with all `NAME` tokens, including variable and function names defined by the user. In an alternate form returned when line number information is requested, the same token might be represented as `(1, 'if', 12)`, where the `12` represents the line number at which the terminal symbol was found.

Terminal elements are represented in much the same way, but without any child elements and the addition of the source text which was identified. The example of the `if` keyword above is representative. The various types of terminal symbols are defined in the C header file `Include/token.h` and the Python module *token*.

The ST objects are not required to support the functionality of this module, but are provided for three purposes: to allow an application to amortize the cost of processing complex parse trees, to provide a parse tree representation which conserves memory space when compared to the Python list or tuple representation, and to ease the creation of additional modules in C which manipulate parse trees. A simple "wrapper" class may be created in Python to hide the use of ST objects.

The *parser* module defines functions for a few distinct purposes. The most important purposes are to create ST objects and to convert ST objects to other representations such as parse trees and compiled code objects, but there are also functions which serve to query the type of parse tree represented by an ST object.

参见:

Module *symbol* Useful constants representing internal nodes of the parse tree.

Module *token* Useful constants representing leaf nodes of the parse tree and functions for testing node values.

32.1.1 Creating ST Objects

ST objects may be created from source code or from a parse tree. When creating an ST object from source, different functions are used to create the `'eval'` and `'exec'` forms.

`parser.`**`expr`**(*source*)
The *expr()* function parses the parameter *source* as if it were an input to `compile(source, 'file.py', 'eval')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

`parser.`**`suite`**(*source*)
The *suite()* function parses the parameter *source* as if it were an input to `compile(source, 'file.py', 'exec')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

`parser.`**`sequence2st`**(*sequence*)
This function accepts a parse tree represented as a sequence and builds an internal representation if possible. If it can validate that the tree conforms to the Python grammar and all nodes are valid node types in the host version of Python, an ST object is created from the internal representation and returned to the called. If there is a problem creating the internal representation, or if the tree cannot be validated, a *ParserError* exception is raised. An ST object created this way should not be assumed to compile correctly; normal exceptions raised by compilation may still be initiated when the ST object is passed to *compilest()*. This may indicate problems not related to syntax (such as a *MemoryError* exception), but may also be due to constructs such as the result of parsing `del f(0)`, which escapes the Python parser but is checked by the bytecode compiler.

Sequences representing terminal tokens may be represented as either two-element lists of the form `(1, 'name')` or as three-element lists of the form `(1, 'name', 56)`. If the third element is present, it is assumed to be a valid line number. The line number may be specified for any subset of the terminal symbols in the input tree.

`parser.`**`tuple2st`**(*sequence*)
This is the same function as *sequence2st()*. This entry point is maintained for backward compatibility.

32.1.2 Converting ST Objects

ST objects, regardless of the input used to create them, may be converted to parse trees represented as list- or tuple-trees, or may be compiled into executable code objects. Parse trees may be extracted with or without line numbering information.

`parser.`**`st2list`**(*ast*[, *line_info*])
This function accepts an ST object from the caller in *ast* and returns a Python list representing the equivalent parse tree. The resulting list representation can be used for inspection or the creation of a new parse tree in list form. This function does not fail so long as memory is available to build the list representation. If the parse tree will only be used for inspection, *st2tuple()* should be used instead to reduce memory consumption and fragmentation. When the list representation is required, this function is significantly faster than retrieving a tuple representation and converting that to nested lists.

If *line_info* is true, line number information will be included for all terminal tokens as a third element of the list representing the token. Note that the line number provided specifies the line on which the token *ends*. This information is omitted if the flag is false or omitted.

`parser.`**`st2tuple`**(*ast*[, *line_info*])
This function accepts an ST object from the caller in *ast* and returns a Python tuple representing the equivalent parse tree. Other than returning a tuple instead of a list, this function is identical to *st2list()*.

If *line_info* is true, line number information will be included for all terminal tokens as a third element of the list representing the token. This information is omitted if the flag is false or omitted.

`parser.`**`compilest`**(*ast*, *filename='<syntax-tree>'*)
The Python byte compiler can be invoked on an ST object to produce code objects which can be used as part of an `exec` statement or a call to the built-in *eval()* function. This function provides the interface to the compiler, passing the internal parse tree from *ast* to the parser, using the source file name specified by the *filename* parameter. The default value supplied for *filename* indicates that the source was an ST object.

Compiling an ST object may result in exceptions related to compilation; an example would be a *SyntaxError* caused by the parse tree for `del f(0)`: this statement is considered legal within the formal grammar for Python but is not a legal language construct. The *SyntaxError* raised for this condition is actually generated by the Python byte-compiler normally, which is why it can be raised at this point by the *parser* module. Most causes of compilation failure can be diagnosed programmatically by inspection of the parse tree.

32.1.3 Queries on ST Objects

Two functions are provided which allow an application to determine if an ST was created as an expression or a suite. Neither of these functions can be used to determine if an ST was created from source code via *expr()* or *suite()* or from a parse tree via *sequence2st()*.

`parser.`**`isexpr`**(*ast*)
When *ast* represents an `'eval'` form, this function returns true, otherwise it returns false. This is useful, since code objects normally cannot be queried for this information using existing built-in functions. Note that the code objects created by *compilest()* cannot be queried like this either, and are identical to those created by the built-in *compile()* function.

parser.**issuite**(*ast*)
This function mirrors *isexpr()* in that it reports whether an ST object represents an `'exec'` form, commonly known as a "suite." It is not safe to assume that this function is equivalent to `not isexpr(ast)`, as additional syntactic fragments may be supported in the future.

32.1.4 Exceptions and Error Handling

The parser module defines a single exception, but may also pass other built-in exceptions from other portions of the Python runtime environment. See each function for information about the exceptions it can raise.

exception parser.**ParserError**
Exception raised when a failure occurs within the parser module. This is generally produced for validation failures rather than the built-in *SyntaxError* raised during normal parsing. The exception argument is either a string describing the reason of the failure or a tuple containing a sequence causing the failure from a parse tree passed to *sequence2st()* and an explanatory string. Calls to *sequence2st()* need to be able to handle either type of exception, while calls to other functions in the module will only need to be aware of the simple string values.

Note that the functions *compilest()*, *expr()*, and *suite()* may raise exceptions which are normally raised by the parsing and compilation process. These include the built in exceptions *MemoryError*, *OverflowError*, *SyntaxError*, and *SystemError*. In these cases, these exceptions carry all the meaning normally associated with them. Refer to the descriptions of each function for detailed information.

32.1.5 ST Objects

Ordered and equality comparisons are supported between ST objects. Pickling of ST objects (using the *pickle* module) is also supported.

parser.**STType**
The type of the objects returned by *expr()*, *suite()* and *sequence2st()*.

ST objects have the following methods:

ST.**compile**([*filename*])
Same as `compilest(st, filename)`.

ST.**isexpr**()
Same as `isexpr(st)`.

ST.**issuite**()
Same as `issuite(st)`.

ST.**tolist**([*line_info*])
Same as `st2list(st, line_info)`.

ST.**totuple**([*line_info*])
Same as `st2tuple(st, line_info)`.

32.1.6 Example: Emulation of `compile()`

While many useful operations may take place between parsing and bytecode generation, the simplest operation is to do nothing. For this purpose, using the *parser* module to produce an intermediate data structure is equivalent to the code

```
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

The equivalent operation using the *parser* module is somewhat longer, and allows the intermediate internal parse tree to be retained as an ST object:

```
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

An application which needs both ST and code objects can package this code into readily available functions:

```
import parser

def load_suite(source_string):
    st = parser.suite(source_string)
    return st, st.compile()

def load_expression(source_string):
    st = parser.expr(source_string)
    return st, st.compile()
```

32.2 `ast` —抽象语法树

2.5 新版功能: The low-level `_ast` module containing only the node classes.

2.6 新版功能: The high-level `ast` module containing all helpers.

源代码： Lib/ast.py

---

ast 模块帮助 Python 程序处理 Python 语法的抽象语法树。抽象语法或许会随着 Python 的更新发布而改变；该模块能够帮助理解当前语法在编程层面的样貌。

抽象语法树可通过将 `ast.PyCF_ONLY_AST` 作为旗标传递给*compile()* 内置函数来生成，或是使用此模块中提供的*parse()* 辅助函数。返回结果将是一个对象树，，其中的类都继承自*ast.AST*。抽象语法树可被内置的*compile()* 函数编译为一个 Python 代码对象。

32.2.1 节点类

class ast.**AST**

这是所有 AST 节点类的基类。实际上，这些节点类派生自 `Parser/Python.asdl` 文件，其中定义的语法树示例如下。它们在 C 语言模块 `_ast` 中定义，并被导出至 *ast* 模块。

抽象语法定义的每个左侧符号 (比方说，`ast.stmt` 或者 `ast.expr`) 定义了一个类。另外，在抽象语法定义的右侧，对每一个构造器也定义了一个类；这些类继承自树左侧的类。比如，`ast.BinOp` 继承自 `ast.expr`。对于多分支产生式 (也就是”和规则”)，树右侧的类是抽象的；只有特定构造器结点的实例能被构造。

_fields

每个具体类都有个属性 *_fields*, 用来给出所有子节点的名字。

每个具体类的实例对它每个子节点都有一个属性，对应类型如文法中所定义。比如，`ast.BinOp` 的实例有个属性 `left`，类型是 `ast.expr`.

如果这些属性在文法中标记为可选（使用问号），对应值可能会是 `None`。如果这些属性有零或多个（用星号标记），对应值会用 Python 的列表来表示。所有可能的属性必须在用 *compile()* 编译得到 AST 时给出，且是有效的值。

lineno

col_offset

`ast.expr` 和 `ast.stmt` 子类的实例有 *lineno* 和 *col_offset* 属性。*lineno* 是源代码的行数 (从 1 开始, 所以第一行行数是 1)，而 *col_offset* 是该生成节点第一个 token 的 UTF-8 字节偏移量。记录下 UTF-8 偏移量的原因是 parser 内部使用 UTF-8。

一个类的构造器 `ast.T` 像下面这样 parse 它的参数。

- 如果有位置参数，它们必须和 `T._fields` 中的元素一样多；他们会像这些名字的属性一样被赋值。
- 如果有关键字参数，它们必须被设为和给定值同名的属性。

比方说，要创建和填充节点 `ast.UnaryOp`，你得用

```
node = ast.UnaryOp()
node.op = ast.USub()
node.operand = ast.Num()
node.operand.n = 5
node.operand.lineno = 0
node.operand.col_offset = 0
node.lineno = 0
node.col_offset = 0
```

或者更紧凑点

```
node = ast.UnaryOp(ast.USub(), ast.Num(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

2.6 新版功能: The constructor as explained above was added. In Python 2.5 nodes had to be created by calling the class constructor without arguments and setting the attributes afterwards.

32.2.2 抽象文法

The module defines a string constant `__version__` which is the decimal Subversion revision number of the file shown below.

抽象文法目前定义如下

```
-- ASDL's five builtin types are identifier, int, string, object, bool

module Python version "$Revision$"
{
        mod = Module(stmt* body)
            | Interactive(stmt* body)
            | Expression(expr body)

            -- not really an actual node but useful in Jython's typesystem.
            | Suite(stmt* body)

        stmt = FunctionDef(identifier name, arguments args,
                           stmt* body, expr* decorator_list)
              | ClassDef(identifier name, expr* bases, stmt* body, expr* decorator_
↪list)
              | Return(expr? value)

              | Delete(expr* targets)
              | Assign(expr* targets, expr value)
              | AugAssign(expr target, operator op, expr value)

              -- not sure if bool is allowed, can always use int
               | Print(expr? dest, expr* values, bool nl)

              -- use 'orelse' because else is a keyword in target languages
              | For(expr target, expr iter, stmt* body, stmt* orelse)
              | While(expr test, stmt* body, stmt* orelse)
              | If(expr test, stmt* body, stmt* orelse)
              | With(expr context_expr, expr? optional_vars, stmt* body)

              -- 'type' is a bad name
              | Raise(expr? type, expr? inst, expr? tback)
              | TryExcept(stmt* body, excepthandler* handlers, stmt* orelse)
              | TryFinally(stmt* body, stmt* finalbody)
              | Assert(expr test, expr? msg)

              | Import(alias* names)
              | ImportFrom(identifier? module, alias* names, int? level)

              -- Doesn't capture requirement that locals must be
              -- defined if globals is
              -- still supports use as a function!
              | Exec(expr body, expr? globals, expr? locals)

              | Global(identifier* names)
              | Expr(expr value)
              | Pass | Break | Continue

              -- XXX Jython will be different
              -- col_offset is the byte offset in the utf8 string the parser uses
              attributes (int lineno, int col_offset)
```

(下页继续)

(续上页)

```
      -- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Dict(expr* keys, expr* values)
     | Set(expr* elts)
     | ListComp(expr elt, comprehension* generators)
     | SetComp(expr elt, comprehension* generators)
     | DictComp(expr key, expr value, comprehension* generators)
     | GeneratorExp(expr elt, comprehension* generators)
     -- the grammar constrains where yield expressions can occur
     | Yield(expr? value)
     -- need sequences for compare to distinguish between
     -- x < 4 < 3 and (x < 4) < 3
     | Compare(expr left, cmpop* ops, expr* comparators)
     | Call(expr func, expr* args, keyword* keywords,
                 expr? starargs, expr? kwargs)
     | Repr(expr value)
     | Num(object n) -- a number as a PyObject.
     | Str(string s) -- need to specify raw, unicode, etc?
     -- other literals? bools?

     -- the following expression can appear in assignment context
     | Attribute(expr value, identifier attr, expr_context ctx)
     | Subscript(expr value, slice slice, expr_context ctx)
     | Name(identifier id, expr_context ctx)
     | List(expr* elts, expr_context ctx)
     | Tuple(expr* elts, expr_context ctx)

      -- col_offset is the byte offset in the utf8 string the parser uses
      attributes (int lineno, int col_offset)

expr_context = Load | Store | Del | AugLoad | AugStore | Param

slice = Ellipsis | Slice(expr? lower, expr? upper, expr? step)
      | ExtSlice(slice* dims)
      | Index(expr value)

boolop = And | Or

operator = Add | Sub | Mult | Div | Mod | Pow | LShift
         | RShift | BitOr | BitXor | BitAnd | FloorDiv

unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

comprehension = (expr target, expr iter, expr* ifs)

-- not sure what to call the first argument for raise and except
excepthandler = ExceptHandler(expr? type, expr? name, stmt* body)
                attributes (int lineno, int col_offset)
```

(下页继续)

(续上页)

```
        arguments = (expr* args, identifier? vararg,
                     identifier? kwarg, expr* defaults)

        -- keyword arguments supplied to call
        keyword = (identifier arg, expr value)

        -- import name with optional 'as' alias.
        alias = (identifier name, identifier? asname)
}
```

32.2.3 ast 中的辅助函数

2.6 新版功能.

Apart from the node classes, *ast* module defines these utility functions and classes for traversing abstract syntax trees:

ast.**parse**(*source*, *filename='<unknown>'*, *mode='exec'*)
把源码解析为 AST 节点。和 compile(source, filename, mode,ast.PyCF_ONLY_AST) 等价。

> **警告:** 足够复杂或是巨大的字符串可能导致 Python 解释器的崩溃，因为 Python 的 AST 编译器是有栈深限制的。

ast.**literal_eval**(*node_or_string*)
Safely evaluate an expression node or a Unicode or *Latin-1* encoded string containing a Python literal or container display. The string or node provided may only consist of the following Python literal structures: strings, numbers, tuples, lists, dicts, booleans, and None.

This can be used for safely evaluating strings containing Python values from untrusted sources without the need to parse the values oneself. It is not capable of evaluating arbitrarily complex expressions, for example involving operators or indexing.

> **警告:** 足够复杂或是巨大的字符串可能导致 Python 解释器的崩溃，因为 Python 的 AST 编译器是有栈深限制的。

ast.**get_docstring**(*node*, *clean=True*)
Return the docstring of the given *node* (which must be a FunctionDef, ClassDef or Module node), or None if it has no docstring. If *clean* is true, clean up the docstring's indentation with *inspect.cleandoc()*.

ast.**fix_missing_locations**(*node*)
When you compile a node tree with *compile()*, the compiler expects lineno and col_offset attributes for every node that supports them. This is rather tedious to fill in for generated nodes, so this helper adds these attributes recursively where not already set, by setting them to the values of the parent node. It works recursively starting at *node*.

ast.**increment_lineno**(*node*, *n=1*)
Increment the line number of each node in the tree starting at *node* by *n*. This is useful to "move code" to a different location in a file.

ast.**copy_location**(*new_node*, *old_node*)
Copy source location (lineno and col_offset) from *old_node* to *new_node* if possible, and return *new_node*.

ast.**iter_fields**(*node*)
Yield a tuple of (fieldname, value) for each field in node._fields that is present on *node*.

ast.**iter_child_nodes**(*node*)
Yield all direct child nodes of *node*, that is, all fields that are nodes and all items of fields that are lists of nodes.

ast.**walk**(*node*)
Recursively yield all descendant nodes in the tree starting at *node* (including *node* itself), in no specified order. This is useful if you only want to modify nodes in place and don't care about the context.

class ast.**NodeVisitor**
A node visitor base class that walks the abstract syntax tree and calls a visitor function for every node found. This function may return a value which is forwarded by the *visit()* method.

This class is meant to be subclassed, with the subclass adding visitor methods.

visit(*node*)
Visit a node. The default implementation calls the method called `self.visit_classname` where *classname* is the name of the node class, or *generic_visit()* if that method doesn't exist.

generic_visit(*node*)
This visitor calls *visit()* on all children of the node.

Note that child nodes of nodes that have a custom visitor method won't be visited unless the visitor calls *generic_visit()* or visits them itself.

Don't use the *NodeVisitor* if you want to apply changes to nodes during traversal. For this a special visitor exists (*NodeTransformer*) that allows modifications.

class ast.**NodeTransformer**
子类*NodeVisitor* 用于遍历抽象语法树，并允许修改节点。

NodeTransformer 将遍历抽象语法树并使用 visitor 方法的返回值去替换或移除旧节点。如果 visitor 方法的返回值为 `None`，则该节点将从其位置移除，否则将替换为返回值。当返回值是原始节点时，无需替换。

如下是一个转换器示例，它将所有出现的名称 (`foo`) 重写为 `data['foo']`:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return copy_location(Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Str(s=node.id)),
            ctx=node.ctx
        ), node)
```

请记住，如果您正在操作的节点具有子节点，则必须先转换其子节点或为该节点调用 `generic_visit()` 方法。

对于属于语句集合（适用于所有语句节点）的节点，访问者还可以返回节点列表而不仅仅是单个节点。

通常你可以像这样使用转换器:

```
node = YourTransformer().visit(node)
```

ast.**dump**(*node*, *annotate_fields=True*, *include_attributes=False*)
Return a formatted dump of the tree in *node*. This is mainly useful for debugging purposes. The returned string will show the names and the values for fields. This makes the code impossible to evaluate, so if evaluation is wanted *annotate_fields* must be set to `False`. Attributes such as line numbers and column offsets are not dumped by default. If this is wanted, *include_attributes* can be set to `True`.

32.3 symtable —Access to the compiler's symbol tables

Source code: Lib/symtable.py

Symbol tables are generated by the compiler from AST just before bytecode is generated. The symbol table is responsible for calculating the scope of every identifier in the code. *symtable* provides an interface to examine these tables.

32.3.1 Generating Symbol Tables

symtable.**symtable**(*code*, *filename*, *compile_type*)
Return the toplevel *SymbolTable* for the Python source *code*. *filename* is the name of the file containing the code. *compile_type* is like the *mode* argument to *compile()*.

32.3.2 Examining Symbol Tables

class symtable.**SymbolTable**
A namespace table for a block. The constructor is not public.

get_type()
Return the type of the symbol table. Possible values are `'class'`, `'module'`, and `'function'`.

get_id()
Return the table's identifier.

get_name()
Return the table's name. This is the name of the class if the table is for a class, the name of the function if the table is for a function, or `'top'` if the table is global (*get_type()* returns `'module'`).

get_lineno()
Return the number of the first line in the block this table represents.

is_optimized()
Return `True` if the locals in this table can be optimized.

is_nested()
Return `True` if the block is a nested class or function.

has_children()
Return `True` if the block has nested namespaces within it. These can be obtained with *get_children()*.

has_exec()
Return `True` if the block uses `exec`.

has_import_star()
Return `True` if the block uses a starred from-import.

get_identifiers()
Return a list of names of symbols in this table.

lookup(*name*)
Lookup *name* in the table and return a *Symbol* instance.

get_symbols()
Return a list of *Symbol* instances for names in the table.

get_children()
Return a list of the nested symbol tables.

class symtable.**Function**
A namespace for a function or method. This class inherits *SymbolTable*.

get_parameters()
Return a tuple containing names of parameters to this function.

get_locals()
Return a tuple containing names of locals in this function.

get_globals()
Return a tuple containing names of globals in this function.

get_frees()
Return a tuple containing names of free variables in this function.

class symtable.**Class**
A namespace of a class. This class inherits *SymbolTable*.

get_methods()
Return a tuple containing the names of methods declared in the class.

class symtable.**Symbol**
An entry in a *SymbolTable* corresponding to an identifier in the source. The constructor is not public.

get_name()
Return the symbol's name.

is_referenced()
Return `True` if the symbol is used in its block.

is_imported()
Return `True` if the symbol is created from an import statement.

is_parameter()
Return `True` if the symbol is a parameter.

is_global()
Return `True` if the symbol is global.

is_declared_global()
Return `True` if the symbol is declared global with a global statement.

is_local()
Return `True` if the symbol is local to its block.

is_free()
Return `True` if the symbol is referenced in its block, but not assigned to.

is_assigned()
Return `True` if the symbol is assigned to in its block.

is_namespace()
Return `True` if name binding introduces new namespace.

If the name is used as the target of a function or class statement, this will be true.

例如

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

Note that a single name can be bound to multiple objects. If the result is `True`, the name may also be bound to other objects, like an int or list, that does not introduce a new namespace.

get_namespaces()
Return a list of namespaces bound to this name.

get_namespace()
Return the namespace bound to this name. If more than one namespace is bound, a *ValueError* is raised.

32.4 symbol —与 Python 解析树一起使用的常量

源代码: Lib/symbol.py

此模块提供用于表示解析树内部节点数值的常量。与大多数 Python 不同，这些常量使用小写字符名称。请参阅 Python 发行版中的 Grammar/Grammar 文件来获取该语言语法上下文中对这些名称的定义。这些名称所映射的特定数字值可能会在 Python 版本之间更改。

此模块还提供了一个额外的数据对象:

symbol.**sym_name**
将此模块中定义的常量的数值映射回名称字符串的字典，允许生成更加人类可读的解析树表示。

32.5 token —与 Python 解析树一起使用的常量

源码: Lib/token.py

此模块提供表示解析树（终端令牌）的叶节点的数值的常量。请参阅 Python 发行版中的文件 Grammar/Grammar，以获取语言语法上下文中名称的定义。名称映射到的特定数值可能会在 Python 版本之间更改。

该模块还提供从数字代码到名称和一些函数的映射。这些函数镜像了 Python C 头文件中的定义。

token.**tok_name**
将此模块中定义的常量的数值映射回名称字符串的字典，允许生成更加人类可读的解析树表示。

token.**ISTERMINAL**(*x*)
Return true for terminal token values.

token.**ISNONTERMINAL**(*x*)
Return true for non-terminal token values.

token.**ISEOF**(*x*)
Return true if *x* is the marker indicating the end of input.

标记常量是:

token.**ENDMARKER**
token.**NAME**
token.**NUMBER**
token.**STRING**
token.**NEWLINE**
token.**INDENT**
token.**DEDENT**
token.**LPAR**
token.**RPAR**
token.**LSQB**

token.**RSQB**
token.**COLON**
token.**COMMA**
token.**SEMI**
token.**PLUS**
token.**MINUS**
token.**STAR**
token.**SLASH**
token.**VBAR**
token.**AMPER**
token.**LESS**
token.**GREATER**
token.**EQUAL**
token.**DOT**
token.**PERCENT**
token.**BACKQUOTE**
token.**LBRACE**
token.**RBRACE**
token.**EQEQUAL**
token.**NOTEQUAL**
token.**LESSEQUAL**
token.**GREATEREQUAL**
token.**TILDE**
token.**CIRCUMFLEX**
token.**LEFTSHIFT**
token.**RIGHTSHIFT**
token.**DOUBLESTAR**
token.**PLUSEQUAL**
token.**MINEQUAL**
token.**STAREQUAL**
token.**SLASHEQUAL**
token.**PERCENTEQUAL**
token.**AMPEREQUAL**
token.**VBAREQUAL**
token.**CIRCUMFLEXEQUAL**
token.**LEFTSHIFTEQUAL**
token.**RIGHTSHIFTEQUAL**
token.**DOUBLESTAREQUAL**
token.**DOUBLESLASH**
token.**DOUBLESLASHEQUAL**
token.**AT**
token.**OP**
token.**ERRORTOKEN**
token.**N_TOKENS**
token.**NT_OFFSET**

参见:

Module *parser* The second example for the *parser* module shows how to use the *symbol* module.

32.6 keyword —检验 Python 关键字

源码： Lib/keyword.py

This module allows a Python program to determine if a string is a keyword.

keyword.**iskeyword**(*s*)
Return true if *s* is a Python keyword.

keyword.**kwlist**
Sequence containing all the keywords defined for the interpreter. If any keywords are defined to only be active when particular *__future__* statements are in effect, these will be included as well.

32.7 tokenize –对 Python 代码使用的标记解析器

源码： Lib/tokenize.py

The *tokenize* module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing "pretty-printers," including colorizers for on-screen displays.

To simplify token stream handling, all operators and delimiters tokens are returned using the generic *token.OP* token type. The exact type can be determined by checking the second field (containing the actual token string matched) of the tuple returned from *tokenize.generate_tokens()* for the character sequence that identifies a specific operator token.

主要的入口是一个*generator*:

tokenize.**generate_tokens**(*readline*)
The *generate_tokens()* generator requires one argument, *readline*, which must be a callable object which provides the same interface as the *readline()* method of built-in file objects (see section *File Objects*). Each call to the function should return one line of input as a string. Alternately, *readline* may be a callable object that signals completion by raising *StopIteration*.

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple `(srow, scol)` of ints specifying the row and column where the token begins in the source; a 2-tuple `(erow, ecol)` of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed (the last tuple item) is the *logical* line; continuation lines are included.

2.2 新版功能.

An older entry point is retained for backward compatibility:

tokenize.**tokenize**(*readline*[, *tokeneater*])
The *tokenize()* function accepts two parameters: one representing the input stream, and one providing an output mechanism for *tokenize()*.

The first parameter, *readline*, must be a callable object which provides the same interface as the *readline()* method of built-in file objects (see section *File Objects*). Each call to the function should return one line of input as a string. Alternately, *readline* may be a callable object that signals completion by raising *StopIteration*.

在 2.5 版更改: Added *StopIteration* support.

The second parameter, *tokeneater*, must also be a callable object. It is called once for each token, with five arguments, corresponding to the tuples generated by *generate_tokens()*.

All constants from the *token* module are also exported from *tokenize*, as are two additional token type values that might be passed to the *tokeneater* function by *tokenize()*:

tokenize.**COMMENT**
Token value used to indicate a comment.

tokenize.**NL**
Token value used to indicate a non-terminating newline. The NEWLINE token indicates the end of a logical line of Python code; NL tokens are generated when a logical line of code is continued over multiple physical lines.

Another function is provided to reverse the tokenization process. This is useful for creating tools that tokenize a script, modify the token stream, and write back the modified script.

tokenize.**untokenize**(*iterable*)
Converts tokens back into Python source code. The *iterable* must return sequences with at least two elements, the token type and the token string. Any additional sequence elements are ignored.

The reconstructed script is returned as a single string. The result is guaranteed to tokenize back to match the input so that the conversion is lossless and round-trips are assured. The guarantee applies only to the token type and token string as the spacing between tokens (column positions) may change.

2.5 新版功能.

exception tokenize.**TokenError**
Raised when either a docstring or expression that may be split over several lines is not completed anywhere in the file, for example:

```
"""Beginning of
docstring
```

或者:

```
[1,
 2,
 3
```

Note that unclosed single-quoted strings do not cause an error to be raised. They are tokenized as ERRORTOKEN, followed by the tokenization of their contents.

Example of a script re-writer that transforms float literals into Decimal objects:

```
def decistmt(s):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print +21.3e-5*-.1234/81.7'
    >>> decistmt(s)
    "print +Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7')"

    >>> exec(s)
    -3.21716034272e-007
    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7

    """
    result = []
    g = generate_tokens(StringIO(s).readline)   # tokenize the string
    for toknum, tokval, _, _, _  in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
            result.extend([
```

(下页继续)

(续上页)

```
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
            result.append((toknum, tokval))
    return untokenize(result)
```

32.8 tabnanny —模糊缩进检测

源代码: Lib/tabnanny.py

目前，该模块旨在作为脚本调用。但是可以使用下面描述的*check()* 函数将其导入 IDE。

注解: 此模块提供的 API 可能会在将来的版本中更改；此类更改可能无法向后兼容。

---

tabnanny.**check**(*file_or_dir*)
If *file_or_dir* is a directory and not a symbolic link, then recursively descend the directory tree named by *file_or_dir*, checking all `.py` files along the way. If *file_or_dir* is an ordinary Python source file, it is checked for whitespace related problems. The diagnostic messages are written to standard output using the print statement.

tabnanny.**verbose**
此标志指明是否打印详细消息。如果作为脚本调用则是通过 `-v` 选项来增加。

tabnanny.**filename_only**
此旗标指明是否只打印包含空格相关问题文件的文件名。如果作为脚本调用则是通过 `-q` 选项来设为真值。

exception tabnanny.**NannyNag**
如果检测到模糊缩进则由*process_tokens()* 引发。在*check()* 中捕获并处理。

tabnanny.**process_tokens**(*tokens*)
此函数由*check()* 用来处理由*tokenize* 模块所生成的标记。

参见:

模块*tokenize* 用于 Python 源代码的词法扫描程序。

32.9 pyclbr —Python class browser support

Source code: Lib/pyclbr.py

The *pyclbr* module can be used to determine some limited information about the classes, methods and top-level functions defined in a module. The information provided is sufficient to implement a traditional three-pane class browser. The information is extracted from the source code rather than by importing the module, so this module is safe to use with untrusted code. This restriction makes it impossible to use this module with modules not implemented in Python, including all standard and optional extension modules.

pyclbr.**readmodule**(*module*, *path=None*)
Read a module and return a dictionary mapping class names to class descriptor objects. The parameter *module* should be the name of a module as a string; it may be the name of a module within a package. The *path* parameter should be a sequence, and is used to augment the value of `sys.path`, which is used to locate module source code.

pyclbr.**readmodule_ex**(*module*, *path=None*)
Like *readmodule()*, but the returned dictionary, in addition to mapping class names to class descriptor objects, also maps top-level function names to function descriptor objects. Moreover, if the module being read is a package, the key `'__path__'` in the returned dictionary has as its value a list which contains the package search path.

32.9.1 类对象

The `Class` objects used as values in the dictionary returned by *readmodule()* and *readmodule_ex()* provide the following data attributes:

Class.**module**
The name of the module defining the class described by the class descriptor.

Class.**name**
The name of the class.

Class.**super**
A list of `Class` objects which describe the immediate base classes of the class being described. Classes which are named as superclasses but which are not discoverable by *readmodule()* are listed as a string with the class name instead of as `Class` objects.

Class.**methods**
A dictionary mapping method names to line numbers.

Class.**file**
Name of the file containing the `class` statement defining the class.

Class.**lineno**
The line number of the `class` statement within the file named by *file*.

32.9.2 函数对象

The `Function` objects used as values in the dictionary returned by *readmodule_ex()* provide the following attributes:

Function.**module**
The name of the module defining the function described by the function descriptor.

Function.**name**
The name of the function.

Function.**file**
Name of the file containing the `def` statement defining the function.

Function.**lineno**
The line number of the `def` statement within the file named by *file*.

32.10 py_compile —Compile Python source files

Source code: Lib/py_compile.py

The *py_compile* module provides a function to generate a byte-code file from a source file, and another function used when the module source file is invoked as a script.

Though not often needed, this function can be useful when installing modules for shared use, especially if some of the users may not have permission to write the byte-code cache files in the directory containing the source code.

exception py_compile.**PyCompileError**
Exception raised when an error occurs while attempting to compile the file.

py_compile.**compile**(*file*[, *cfile*[, *dfile*[, *doraise*]]])
Compile a source file to byte-code and write out the byte-code cache file. The source code is loaded from the file named *file*. The byte-code is written to *cfile*, which defaults to *file* + `'c'` (`'o'` if optimization is enabled in the current interpreter). If *dfile* is specified, it is used as the name of the source file in error messages instead of *file*. If *doraise* is true, a *PyCompileError* is raised when an error is encountered while compiling *file*. If *doraise* is false (the default), an error string is written to `sys.stderr`, but no exception is raised.

py_compile.**main**([*args*])
Compile several source files. The files named in *args* (or on the command line, if *args* is not specified) are compiled and the resulting bytecode is cached in the normal manner. This function does not search a directory structure to locate source files; it only compiles files named explicitly. If `'-'` is the only parameter in args, the list of files is taken from standard input.

在 2.7 版更改: Added support for `'-'`.

When this module is run as a script, the *main()* is used to compile all the files named on the command line. The exit status is nonzero if one of the files could not be compiled.

在 2.6 版更改: Added the nonzero exit status when module is run as a script.

参见:

Module ***compileall*** Utilities to compile all Python source files in a directory tree.

32.11 compileall —Byte-compile Python libraries

Source code: Lib/compileall.py

This module provides some utility functions to support installing Python libraries. These functions compile Python source files in a directory tree. This module can be used to create the cached byte-code files at library installation time, which makes them available for use even by users who don't have write permission to the library directories.

32.11.1 Command-line use

This module can work as a script (using **python -m compileall**) to compile Python sources.

directory ...
file ...
Positional arguments are files to compile or directories that contain source files, traversed recursively. If no argument is given, behave as if the command line was `-l <directories from sys.path>`.

-l
Do not recurse into subdirectories, only compile source code files directly contained in the named or implied directories.

-f
Force rebuild even if timestamps are up-to-date.

-q
Do not print the list of files compiled, print only error messages.

-d `destdir`
Directory prepended to the path to each file being compiled. This will appear in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

-x `regex`
regex is used to search the full path to each file considered for compilation, and if the regex produces a match, the file is skipped.

-i `list`
Read the file `list` and add each line that it contains to the list of files and directories to compile. If `list` is `-`, read lines from `stdin`.

在 2.7 版更改: Added the `-i` option.

32.11.2 Public functions

`compileall.`**compile_dir**(*dir*[, *maxlevels*[, *ddir*[, *force*[, *rx*[, *quiet*]]]]])
Recursively descend the directory tree named by *dir*, compiling all `.py` files along the way.

The *maxlevels* parameter is used to limit the depth of the recursion; it defaults to `10`.

If *ddir* is given, it is prepended to the path to each file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If *force* is true, modules are re-compiled even if the timestamps are up to date.

If *rx* is given, its search method is called on the complete path to each file considered for compilation, and if it returns a true value, the file is skipped.

If *quiet* is true, nothing is printed to the standard output unless errors occur.

`compileall.`**compile_file**(*fullname*[, *ddir*[, *force*[, *rx*[, *quiet*]]]])
Compile the file with path *fullname*.

If *ddir* is given, it is prepended to the path to the file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If *rx* is given, its search method is passed the full path name to the file being compiled, and if it returns a true value, the file is not compiled and `True` is returned.

If *quiet* is true, nothing is printed to the standard output unless errors occur.

2.7 新版功能.

compileall.**compile_path**([*skip_curdir*[, *maxlevels*[, *force*]]])
Byte-compile all the `.py` files found along `sys.path`. If *skip_curdir* is true (the default), the current directory is not included in the search. All other parameters are passed to the *compile_dir()* function. Note that unlike the other compile functions, `maxlevels` defaults to `0`.

To force a recompile of all the `.py` files in the `Lib/` subdirectory and all its subdirectories:

```
import compileall

compileall.compile_dir('Lib/', force=True)

# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'), force=True)
```

参见:

模块*py_compile* Byte-compile a single source file.

32.12 dis —Python 字节码反汇编器

Source code: Lib/dis.py

---

dis 模块通过反汇编支持 CPython 的*bytecode* 分析。该模块作为输入的 CPython 字节码在文件 `Include/opcode.h` 中定义，并由编译器和解释器使用。

CPython implementation detail: Bytecode is an implementation detail of the CPython interpreter! No guarantees are made that bytecode will not be added, removed, or changed between versions of Python. Use of this module should not be considered to work across Python VMs or Python releases.

示例：给出函数 `myfunc()`:

```
def myfunc(alist):
    return len(alist)
```

the following command can be used to get the disassembly of `myfunc()`:

```
>>> dis.dis(myfunc)
  2           0 LOAD_GLOBAL              0 (len)
              3 LOAD_FAST                0 (alist)
              6 CALL_FUNCTION            1
              9 RETURN_VALUE
```

(“2” 是行号)。

The *dis* module defines the following functions and constants:

dis.**dis**([*bytesource*])
Disassemble the *bytesource* object. *bytesource* can denote either a module, a class, a method, a function, or a code object. For a module, it disassembles all functions. For a class, it disassembles all methods. For a single code sequence, it prints one line per bytecode instruction. If no object is provided, it disassembles the last traceback.

dis.**distb**([*tb*])
Disassembles the top-of-stack function of a traceback, using the last traceback if none was passed. The instruction causing the exception is indicated.

dis.**disassemble**(*code*[, *lasti*])
Disassembles a code object, indicating the last instruction if *lasti* was provided. The output is divided in the following columns:

1. 行号，用于每行的第一条指令
2. 当前指令，表示为 `-->`，
3. 一个标记的指令，用 >> 表示，
4. 指令的地址，
5. 操作码名称，
6. 操作参数，和
7. 括号中的参数解释。

参数解释识别本地和全局变量名称、常量值、分支目标和比较运算符。

dis.**disco**(*code*[, *lasti*])
A synonym for `disassemble()`. It is more convenient to type, and kept for compatibility with earlier Python releases.

dis.**findlinestarts**(*code*)
This generator function uses the `co_firstlineno` and `co_lnotab` attributes of the code object *code* to find the offsets which are starts of lines in the source code. They are generated as `(offset, lineno)` pairs.

dis.**findlabels**(*code*)
Detect all offsets in the code object *code* which are jump targets, and return a list of these offsets.

dis.**opname**
操作名称序列，可使用字节码来索引。

dis.**opmap**
映射操作名称到字节码的字典

dis.**cmp_op**
所有比较操作名称的序列。

dis.**hasconst**
访问常量的字节码序列。

dis.**hasfree**
Sequence of bytecodes that access a free variable.

dis.**hasname**
按名称访问属性的字节码序列

dis.**hasjrel**
具有相对跳转目标的字节码序列。

dis.**hasjabs**
具有绝对跳转目标的字节码序列。

dis.**haslocal**
访问局部变量的字节码序列。

dis.**hascompare**
布尔运算的字节码序列

32.12.1 Python 字节码说明

Python 编译器当前生成以下字节码指令。

STOP_CODE()
Indicates end-of-code to the compiler, not used by the interpreter.

NOP()
什么都不做。用作字节码优化器的占位符。

POP_TOP()
删除堆栈顶部（TOS）项。

ROT_TWO()
交换两个最顶层的堆栈项。

ROT_THREE()
将第二个和第三个堆叠项目向上提升一个位置，向上移动到位置三。

ROT_FOUR()
Lifts second, third and forth stack item one position up, moves top down to position four.

DUP_TOP()
复制堆栈顶部的引用。

Unary Operations take the top of the stack, apply the operation, and push the result back on the stack.

UNARY_POSITIVE()
实现 `TOS = +TOS` 。

UNARY_NEGATIVE()
实现 `TOS = -TOS` 。

UNARY_NOT()
实现 `TOS = not TOS` 。

UNARY_CONVERT()
Implements `TOS = `TOS``.

UNARY_INVERT()
实现 `TOS = ~TOS` 。

GET_ITER()
实现 `TOS = iter(TOS)` 。

二元操作从堆栈中删除堆栈顶部（TOS）和第二个最顶层堆栈项（TOS1）。它们执行操作，并将结果放回堆栈。

BINARY_POWER()
实现 `TOS = TOS1 ** TOS` 。

BINARY_MULTIPLY()
实现 `TOS = TOS1 * TOS` 。

BINARY_DIVIDE()
Implements `TOS = TOS1 / TOS` when `from __future__ import division` is not in effect.

BINARY_FLOOR_DIVIDE()
实现 `TOS = TOS1 // TOS`。

BINARY_TRUE_DIVIDE()
Implements `TOS = TOS1 / TOS` when `from __future__ import division` is in effect.

BINARY_MODULO()
实现 TOS = TOS1 % TOS 。

BINARY_ADD()
实现 TOS = TOS1 + TOS 。

BINARY_SUBTRACT()
实现 TOS = TOS1 - TOS 。

BINARY_SUBSCR()
实现 TOS = TOS1[TOS] 。

BINARY_LSHIFT()
实现 TOS = TOS1 << TOS 。

BINARY_RSHIFT()
实现 TOS = TOS1 >> TOS 。

BINARY_AND()
实现 TOS = TOS1 & TOS 。

BINARY_XOR()
实现 TOS = TOS1 ^ TOS 。

BINARY_OR()
实现 TOS = TOS1 | TOS 。

就地操作就像二元操作，因为它们删除了 TOS 和 TOS1，并将结果推回到堆栈上，但是当 TOS1 支持它时，操作就地完成，并且产生的 TOS 可能是（但不一定）原来的 TOS1。

INPLACE_POWER()
就地实现 TOS = TOS1 ** TOS 。

INPLACE_MULTIPLY()
就地实现 TOS = TOS1 * TOS 。

INPLACE_DIVIDE()
Implements in-place TOS = TOS1 / TOS when from __future__ import division is not in effect.

INPLACE_FLOOR_DIVIDE()
就地实现 TOS = TOS1 // TOS 。

INPLACE_TRUE_DIVIDE()
Implements in-place TOS = TOS1 / TOS when from __future__ import division is in effect.

INPLACE_MODULO()
就地实现 TOS = TOS1 % TOS 。

INPLACE_ADD()
就地实现 TOS = TOS1 + TOS 。

INPLACE_SUBTRACT()
就地实现 TOS = TOS1 - TOS 。

INPLACE_LSHIFT()
就地实现 TOS = TOS1 << TOS 。

INPLACE_RSHIFT()
就地实现 TOS = TOS1 >> TOS 。

INPLACE_AND()
就地实现 TOS = TOS1 & TOS 。

INPLACE_XOR()
就地实现 `TOS = TOS1 ^ TOS` 。

INPLACE_OR()
就地实现 `TOS = TOS1 | TOS` 。

The slice opcodes take up to three parameters.

SLICE+0()
Implements `TOS = TOS[:]`.

SLICE+1()
Implements `TOS = TOS1[TOS:]`.

SLICE+2()
Implements `TOS = TOS1[:TOS]`.

SLICE+3()
Implements `TOS = TOS2[TOS1:TOS]`.

Slice assignment needs even an additional parameter. As any statement, they put nothing on the stack.

STORE_SLICE+0()
Implements `TOS[:] = TOS1`.

STORE_SLICE+1()
Implements `TOS1[TOS:] = TOS2`.

STORE_SLICE+2()
Implements `TOS1[:TOS] = TOS2`.

STORE_SLICE+3()
Implements `TOS2[TOS1:TOS] = TOS3`.

DELETE_SLICE+0()
Implements `del TOS[:]`.

DELETE_SLICE+1()
Implements `del TOS1[TOS:]`.

DELETE_SLICE+2()
Implements `del TOS1[:TOS]`.

DELETE_SLICE+3()
Implements `del TOS2[TOS1:TOS]`.

STORE_SUBSCR()
实现 `TOS1[TOS] = TOS2` 。

DELETE_SUBSCR()
实现 `del TOS1[TOS]` 。

Miscellaneous opcodes.

PRINT_EXPR()
实现交互模式的表达式语句。TOS 从堆栈中被移除并打印。在非交互模式下，表达式语句以*POP_TOP* 终止。

PRINT_ITEM()
Prints TOS to the file-like object bound to `sys.stdout`. There is one such instruction for each item in the `print` statement.

PRINT_ITEM_TO()
Like PRINT_ITEM, but prints the item second from TOS to the file-like object at TOS. This is used by the extended print statement.

PRINT_NEWLINE()
Prints a new line on sys.stdout. This is generated as the last operation of a print statement, unless the statement ends with a comma.

PRINT_NEWLINE_TO()
Like PRINT_NEWLINE, but prints the new line on the file-like object on the TOS. This is used by the extended print statement.

BREAK_LOOP()
Terminates a loop due to a break statement.

CONTINUE_LOOP(*target*)
Continues a loop due to a continue statement. *target* is the address to jump to (which should be a *FOR_ITER* instruction).

LIST_APPEND(*i*)
Calls list.append(TOS[-i], TOS). Used to implement list comprehensions. While the appended value is popped off, the list object remains on the stack so that it is available for further iterations of the loop.

LOAD_LOCALS()
Pushes a reference to the locals of the current scope on the stack. This is used in the code for a class definition: After the class body is evaluated, the locals are passed to the class definition.

RETURN_VALUE()
返回 TOS 到函数的调用者。

YIELD_VALUE()
Pops TOS and yields it from a *generator*.

IMPORT_STAR()
将所有不以 '_' 开头的符号直接从模块 TOS 加载到本地名称空间。加载所有名称后弹出该模块。这个操作码实现了 from module import * 。

EXEC_STMT()
Implements exec TOS2,TOS1,TOS. The compiler fills missing optional parameters with None.

POP_BLOCK()
Removes one block from the block stack. Per frame, there is a stack of blocks, denoting nested loops, try statements, and such.

END_FINALLY()
Terminates a finally clause. The interpreter recalls whether the exception has to be re-raised, or whether the function returns, and continues with the outer-next block.

BUILD_CLASS()
Creates a new class object. TOS is the methods dictionary, TOS1 the tuple of the names of the base classes, and TOS2 the class name.

SETUP_WITH(*delta*)
This opcode performs several operations before a with block starts. First, it loads __exit__() from the context manager and pushes it onto the stack for later use by *WITH_CLEANUP*. Then, __enter__() is called, and a finally block pointing to *delta* is pushed. Finally, the result of calling the enter method is pushed onto the stack. The next opcode will either ignore it (*POP_TOP*), or store it in (a) variable(s) (*STORE_FAST*, *STORE_NAME*, or *UNPACK_SEQUENCE*).

WITH_CLEANUP()
Cleans up the stack when a with statement block exits. On top of the stack are 1–3 values indicating how/why

the finally clause was entered:

- TOP = `None`
- (TOP, SECOND) = (`WHY_{RETURN, CONTINUE}`), retval
- TOP = `WHY_*`; no retval below it
- (TOP, SECOND, THIRD) = exc_info()

Under them is EXIT, the context manager's `__exit__()` bound method.

In the last case, `EXIT(TOP, SECOND, THIRD)` is called, otherwise `EXIT(None, None, None)`.

EXIT is removed from the stack, leaving the values above it in the same order. In addition, if the stack represents an exception, *and* the function call returns a 'true' value, this information is "zapped", to prevent `END_FINALLY` from re-raising the exception. (But non-local gotos should still be resumed.)

All of the following opcodes expect arguments. An argument is two bytes, with the more significant byte last.

STORE_NAME(*namei*)
Implements `name = TOS`. *namei* is the index of *name* in the attribute `co_names` of the code object. The compiler tries to use `STORE_FAST` or `STORE_GLOBAL` if possible.

DELETE_NAME(*namei*)
实现 `del name`，其中 *namei* 是代码对象的 `co_names` 属性的索引。

UNPACK_SEQUENCE(*count*)
将 TOS 解包为 *count* 个单独的值，它们将按从右至左的顺序被放入堆栈。

DUP_TOPX(*count*)
Duplicate *count* items, keeping them in the same order. Due to implementation limits, *count* should be between 1 and 5 inclusive.

STORE_ATTR(*namei*)
实现 `TOS.name = TOS1`，其中 *namei* 是 name 在 `co_names` 中的索引号。

DELETE_ATTR(*namei*)
实现 `del TOS.name`，使用 *namei* 作为 `co_names` 中的索引号。

STORE_GLOBAL(*namei*)
Works as `STORE_NAME`, but stores the name as a global.

DELETE_GLOBAL(*namei*)
Works as `DELETE_NAME`, but deletes a global name.

LOAD_CONST(*consti*)
将 `co_consts[consti]` 推入栈顶。

LOAD_NAME(*namei*)
将与 `co_names[namei]` 相关联的值推入栈顶。

BUILD_TUPLE(*count*)
创建一个使用了来自栈的 *count* 个项的元组，并将结果元组推入栈顶。

BUILD_LIST(*count*)
Works as `BUILD_TUPLE`, but creates a list.

BUILD_SET(*count*)
Works as `BUILD_TUPLE`, but creates a set.

2.7 新版功能.

BUILD_MAP(*count*)
Pushes a new dictionary object onto the stack. The dictionary is pre-sized to hold *count* entries.

LOAD_ATTR(*namei*)
将 TOS 替换为 `getattr(TOS, co_names[namei])`。

COMPARE_OP(*opname*)
执行布尔运算操作。操作名称可在 `cmp_op[opname]` 中找到。

IMPORT_NAME(*namei*)
Imports the module `co_names[namei]`. TOS and TOS1 are popped and provide the *fromlist* and *level* arguments of *__import__()*. The module object is pushed onto the stack. The current namespace is not affected: for a proper import statement, a subsequent `STORE_FAST` instruction modifies the namespace.

IMPORT_FROM(*namei*)
Loads the attribute `co_names[namei]` from the module found in TOS. The resulting object is pushed onto the stack, to be subsequently stored by a `STORE_FAST` instruction.

JUMP_FORWARD(*delta*)
将字节码计数器的值增加 *delta*。

POP_JUMP_IF_TRUE(*target*)
如果 TOS 为真值，则将字节码计数器的值设为 *target*。TOS 会被弹出。

POP_JUMP_IF_FALSE(*target*)
如果 TOS 为假值，则将字节码计数器的值设为 *target*。TOS 会被弹出。

JUMP_IF_TRUE_OR_POP(*target*)
如果 TOS 为真值，则将字节码计数器的值设为 *target* 并将 TOS 留在栈顶。否则（如 TOS 为假值），TOS 会被弹出。

JUMP_IF_FALSE_OR_POP(*target*)
如果 TOS 为假值，则将字节码计数器的值设为 *target* 并将 TOS 留在栈顶。否则（如 TOS 为假值），TOS 会被弹出。

JUMP_ABSOLUTE(*target*)
将字节码计数器的值设为 *target*。

FOR_ITER(*delta*)
`TOS` is an *iterator*. Call its `next()` method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted `TOS` is popped, and the bytecode counter is incremented by *delta*.

LOAD_GLOBAL(*namei*)
加载名称为 `co_names[namei]` 的全局对象推入栈顶。

SETUP_LOOP(*delta*)
Pushes a block for a loop onto the block stack. The block spans from the current instruction with a size of *delta* bytes.

SETUP_EXCEPT(*delta*)
Pushes a try block from a try-except clause onto the block stack. *delta* points to the first except block.

SETUP_FINALLY(*delta*)
Pushes a try block from a try-except clause onto the block stack. *delta* points to the finally block.

STORE_MAP()
Store a key and value pair in a dictionary. Pops the key and value while leaving the dictionary on the stack.

LOAD_FAST(*var_num*)
将指向局部对象 `co_varnames[var_num]` 的引用推入栈顶。

STORE_FAST(*var_num*)
将 TOS 存放到局部变量 `co_varnames[var_num]`。

DELETE_FAST(*var_num*)
移除局部对象 `co_varnames[var_num]`。

LOAD_CLOSURE(*i*)
将一个包含在单元的第 *i* 个空位中的对单元的引用推入栈顶并释放可用的存储空间。如果 *i* 小于 *co_cellvars* 的长度则变量的名称为 `co_cellvars[i]`。否则为 `co_freevars[i - len(co_cellvars)]`。

LOAD_DEREF(*i*)
加载包含在单元的第 *i* 个空位中的单元并释放可用的存储空间。将一个对单元所包含对象的引用推入栈顶。

STORE_DEREF(*i*)
将 TOS 存放到包含在单元的第 *i* 个空位中的单元内并释放可用存储空间。

SET_LINENO(*lineno*)
This opcode is obsolete.

RAISE_VARARGS(*argc*)
Raises an exception. *argc* indicates the number of arguments to the raise statement, ranging from 0 to 3. The handler will find the traceback as TOS2, the parameter as TOS1, and the exception as TOS.

CALL_FUNCTION(*argc*)
Calls a callable object. The low byte of *argc* indicates the number of positional arguments, the high byte the number of keyword arguments. The stack contains keyword arguments on top (if any), then the positional arguments below that (if any), then the callable object to call below that. Each keyword argument is represented with two values on the stack: the argument's name, and its value, with the argument's value above the name on the stack. The positional arguments are pushed in the order that they are passed in to the callable object, with the right-most positional argument on top. `CALL_FUNCTION` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

MAKE_FUNCTION(*argc*)
Pushes a new function object on the stack. TOS is the code associated with the function. The function object is defined to have *argc* default parameters, which are found below TOS.

MAKE_CLOSURE(*argc*)
Creates a new function object, sets its *func_closure* slot, and pushes it on the stack. TOS is the code associated with the function, TOS1 the tuple containing cells for the closure's free variables. The function also has *argc* default parameters, which are found below the cells.

BUILD_SLICE(*argc*)
将一个切片对象推入栈顶。*argc* 必须为 2 或 3。如果为 2，则推入 `slice(TOS1, TOS)`；如果为 3，则推入 `slice(TOS2, TOS1, TOS)`。请参阅*slice()* 内置函数了解详细信息。

EXTENDED_ARG(*ext*)
Prefixes any opcode which has an argument too big to fit into the default two bytes. *ext* holds two additional bytes which, taken together with the subsequent opcode's argument, comprise a four-byte argument, *ext* being the two most-significant bytes.

CALL_FUNCTION_VAR(*argc*)
Calls a callable object, similarly to *CALL_FUNCTION*. *argc* represents the number of keyword and positional arguments, identically to *CALL_FUNCTION*. The top of the stack contains an iterable object containing additional positional arguments. Below that are keyword arguments (if any), positional arguments (if any) and a callable object, identically to *CALL_FUNCTION*. Before the callable object is called, the iterable object is "unpacked" and its contents are appended to the positional arguments passed in. The iterable object is ignored when computing the value of `argc`.

CALL_FUNCTION_KW(*argc*)
Calls a callable object, similarly to *CALL_FUNCTION*. *argc* represents the number of keyword and positional arguments, identically to *CALL_FUNCTION*. The top of the stack contains a mapping object containing additional keyword arguments. Below that are keyword arguments (if any), positional arguments (if any) and a callable object, identically to *CALL_FUNCTION*. Before the callable is called, the mapping object at the top of the stack

is “unpacked” and its contents are appended to the keyword arguments passed in. The mapping object at the top of the stack is ignored when computing the value of `argc`.

CALL_FUNCTION_VAR_KW(*argc*)
Calls a callable object, similarly to *CALL_FUNCTION_VAR* and *CALL_FUNCTION_KW*. *argc* represents the number of keyword and positional arguments, identically to *CALL_FUNCTION*. The top of the stack contains a mapping object, as per *CALL_FUNCTION_KW*. Below that is an iterable object, as per *CALL_FUNCTION_VAR*. Below that are keyword arguments (if any), positional arguments (if any) and a callable object, identically to *CALL_FUNCTION*. Before the callable is called, the mapping object and iterable object are each “unpacked” and their contents passed in as keyword and positional arguments respectively, identically to *CALL_FUNCTION_VAR* and *CALL_FUNCTION_KW*. The mapping object and iterable object are both ignored when computing the value of `argc`.

HAVE_ARGUMENT()
This is not really an opcode. It identifies the dividing line between opcodes which don’t take arguments < `HAVE_ARGUMENT` and those which do >= `HAVE_ARGUMENT`.

32.13 pickletools —pickle 开发者工具集

2.3 新版功能.

源代码: Lib/pickletools.py

This module contains various constants relating to the intimate details of the *pickle* module, some lengthy comments about the implementation, and a few useful functions for analyzing pickled data. The contents of this module are useful for Python core developers who are working on the *pickle* and *cPickle* implementations; ordinary users of the *pickle* module probably won’t find the *pickletools* module relevant.

pickletools.**dis**(*pickle*, *out=None*, *memo=None*, *indentlevel=4*)
Outputs a symbolic disassembly of the pickle to the file-like object *out*, defaulting to `sys.stdout`. *pickle* can be a string or a file-like object. *memo* can be a Python dictionary that will be used as the pickle’s memo; it can be used to perform disassemblies across multiple pickles created by the same pickler. Successive levels, indicated by `MARK` opcodes in the stream, are indented by *indentlevel* spaces.

pickletools.**genops**(*pickle*)
提供包含 pickle 中所有操作码的*iterator*，返回一个 `(opcode, arg, pos)` 三元组的序列。*opcode* 是 `OpcodeInfo` 类的一个实例；*arg* 是 Python 对象形式的 opcode 参数的已解码值；*pos* 是 opcode 所在的位置。*pickle* 可以是一个字符串或一个文件类对象。

pickletools.**optimize**(*picklestring*)
在消除未使用的 `PUT` 操作码之后返回一个新的等效 pickle 字符串。优化后的 pickle 将更为简短，耗费更为的传输时间，要求更少的存储空间并能更高效地解封。

2.6 新版功能.

CHAPTER 33

Python compiler package

2.6 版后已移除: The *compiler* package has been removed in Python 3.

The Python compiler package is a tool for analyzing Python source code and generating Python bytecode. The compiler contains libraries to generate an abstract syntax tree from Python source code and to generate Python *bytecode* from the tree.

The *compiler* package is a Python source to bytecode translator written in Python. It uses the built-in parser and standard *parser* module to generate a concrete syntax tree. This tree is used to generate an abstract syntax tree (AST) and then Python bytecode.

The full functionality of the package duplicates the built-in compiler provided with the Python interpreter. It is intended to match its behavior almost exactly. Why implement another compiler that does the same thing? The package is useful for a variety of purposes. It can be modified more easily than the built-in compiler. The AST it generates is useful for analyzing Python source code.

This chapter explains how the various components of the *compiler* package work. It blends reference material with a tutorial.

33.1 The basic interface

The top-level of the package defines four functions. If you import *compiler*, you will get these functions and a collection of modules contained in the package.

`compiler.`**`parse`**(*buf*)
: Returns an abstract syntax tree for the Python source code in *buf*. The function raises *SyntaxError* if there is an error in the source code. The return value is a `compiler.ast.Module` instance that contains the tree.

`compiler.`**`parseFile`**(*path*)
: Return an abstract syntax tree for the Python source code in the file specified by *path*. It is equivalent to `parse(open(path).read())`.

`compiler.`**`walk`**(*ast*, *visitor*[, *verbose*])
: Do a pre-order walk over the abstract syntax tree *ast*. Call the appropriate method on the *visitor* instance for each node encountered.

compiler.**compile**(*source*, *filename*, *mode*, *flags=None*, *dont_inherit=None*)
Compile the string *source*, a Python module, statement or expression, into a code object that can be executed by the exec statement or *eval()*. This function is a replacement for the built-in *compile()* function.

The *filename* will be used for run-time error messages.

The *mode* must be 'exec' to compile a module, 'single' to compile a single (interactive) statement, or 'eval' to compile an expression.

The *flags* and *dont_inherit* arguments affect future-related statements, but are not supported yet.

compiler.**compileFile**(*source*)
Compiles the file *source* and generates a .pyc file.

The *compiler* package contains the following modules: *ast*, consts, future, misc, pyassem, pycodegen, symbols, transformer, and visitor.

33.2 Limitations

There are some problems with the error checking of the compiler package. The interpreter detects syntax errors in two distinct phases. One set of errors is detected by the interpreter's parser, the other set by the compiler. The compiler package relies on the interpreter's parser, so it get the first phases of error checking for free. It implements the second phase itself, and that implementation is incomplete. For example, the compiler package does not raise an error if a name appears more than once in an argument list: `def f(x, x): ...`

A future version of the compiler should fix these problems.

33.3 Python Abstract Syntax

The *compiler.ast* module defines an abstract syntax for Python. In the abstract syntax tree, each node represents a syntactic construct. The root of the tree is Module object.

The abstract syntax offers a higher level interface to parsed Python source code. The *parser* module and the compiler written in C for the Python interpreter use a concrete syntax tree. The concrete syntax is tied closely to the grammar description used for the Python parser. Instead of a single node for a construct, there are often several levels of nested nodes that are introduced by Python's precedence rules.

The abstract syntax tree is created by the compiler.transformer module. The transformer relies on the built-in Python parser to generate a concrete syntax tree. It generates an abstract syntax tree from the concrete tree.

The transformer module was created by Greg Stein and Bill Tutt for an experimental Python-to-C compiler. The current version contains a number of modifications and improvements, but the basic form of the abstract syntax and of the transformer are due to Stein and Tutt.

33.3.1 AST Nodes

The *compiler.ast* module is generated from a text file that describes each node type and its elements. Each node type is represented as a class that inherits from the abstract base class *compiler.ast.Node* and defines a set of named attributes for child nodes.

class compiler.ast.**Node**
The *Node* instances are created automatically by the parser generator. The recommended interface for specific *Node* instances is to use the public attributes to access child nodes. A public attribute may be bound to a single

node or to a sequence of nodes, depending on the *Node* type. For example, the `bases` attribute of the `Class` node, is bound to a list of base class nodes, and the `doc` attribute is bound to a single node.

Each *Node* instance has a `lineno` attribute which may be `None`. XXX Not sure what the rules are for which nodes will have a useful lineno.

All *Node* objects offer the following methods:

getChildren()
Returns a flattened list of the child nodes and objects in the order they occur. Specifically, the order of the nodes is the order in which they appear in the Python grammar. Not all of the children are *Node* instances. The names of functions and classes, for example, are plain strings.

getChildNodes()
Returns a flattened list of the child nodes in the order they occur. This method is like *getChildren()*, except that it only returns those children that are *Node* instances.

Two examples illustrate the general structure of *Node* classes. The `while` statement is defined by the following grammar production:

```
while_stmt:     "while" expression ":" suite
               ["else" ":" suite]
```

The `While` node has three attributes: *test*, `body`, and `else_`. (If the natural name for an attribute is also a Python reserved word, it can't be used as an attribute name. An underscore is appended to the word to make it a legal identifier, hence `else_` instead of `else`.)

The `if` statement is more complicated because it can include several tests.

```
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
```

The `If` node only defines two attributes: `tests` and `else_`. The `tests` attribute is a sequence of test expression, consequent body pairs. There is one pair for each `if`/`elif` clause. The first element of the pair is the test expression. The second elements is a `Stmt` node that contains the code to execute if the test is true.

The `getChildren()` method of `If` returns a flat list of child nodes. If there are three `if`/`elif` clauses and no `else` clause, then `getChildren()` will return a list of six elements: the first test expression, the first `Stmt`, the second text expression, etc.

The following table lists each of the *Node* subclasses defined in *compiler.ast* and each of the public attributes available on their instances. The values of most of the attributes are themselves *Node* instances or sequences of instances. When the value is something other than an instance, the type is noted in the comment. The attributes are listed in the order in which they are returned by `getChildren()` and `getChildNodes()`.

Node type	Attribute	Value
`Add`	`left`	left operand
	`right`	right operand
`And`	`nodes`	list of operands
`AssAttr`		*attribute as target of assignment*
	`expr`	expression on the left-hand side of the dot
	`attrname`	the attribute name, a string
	`flags`	XXX
`AssList`	`nodes`	list of list elements being assigned to
`AssName`	`name`	name being assigned to
	`flags`	XXX
`AssTuple`	`nodes`	list of tuple elements being assigned to
`Assert`	*test*	the expression to be tested

下页继续

表 1 – 续上页

Node type	Attribute	Value
	`fail`	the value of the *`AssertionError`*
`Assign`	`nodes`	a list of assignment targets, one per equal sign
	`expr`	the value being assigned
`AugAssign`	`node`	
	`op`	
	`expr`	
`Backquote`	`expr`	
`Bitand`	`nodes`	
`Bitor`	`nodes`	
`Bitxor`	`nodes`	
`Break`		
`CallFunc`	`node`	expression for the callee
	`args`	a list of arguments
	`star_args`	the extended *-arg value
	`dstar_args`	the extended **-arg value
`Class`	`name`	the name of the class, a string
	`bases`	a list of base classes
	`doc`	doc string, a string or `None`
	`code`	the body of the class statement
`Compare`	`expr`	
	`ops`	
`Const`	`value`	
`Continue`		
`Decorators`	`nodes`	List of function decorator expressions
`Dict`	`items`	
`Discard`	`expr`	
`Div`	`left`	
	`right`	
`Ellipsis`		
`Expression`	`node`	
`Exec`	`expr`	
	`locals`	
	`globals`	
`FloorDiv`	`left`	
	`right`	
`For`	`assign`	
	`list`	
	`body`	
	`else_`	
`From`	`modname`	
	`names`	
`Function`	`decorators`	`Decorators` or `None`
	`name`	name used in def, a string
	`argnames`	list of argument names, as strings
	`defaults`	list of default values
	`flags`	xxx
	`doc`	doc string, a string or `None`
	`code`	the body of the function
`GenExpr`	*`code`*	

下页继续

表 1 – 续上页

Node type	Attribute	Value
GenExprFor	assign	
	iter	
	ifs	
GenExprIf	*test*	
GenExprInner	expr	
	quals	
Getattr	expr	
	attrname	
Global	names	
If	tests	
	else_	
Import	names	
Invert	expr	
Keyword	name	
	expr	
Lambda	argnames	
	defaults	
	flags	
	code	
LeftShift	left	
	right	
List	nodes	
ListComp	expr	
	quals	
ListCompFor	assign	
	list	
	ifs	
ListCompIf	*test*	
Mod	left	
	right	
Module	doc	doc string, a string or None
	node	body of the module, a Stmt
Mul	left	
	right	
Name	name	
Not	expr	
Or	nodes	
Pass		
Power	left	
	right	
Print	nodes	
	dest	
Printnl	nodes	
	dest	
Raise	expr1	
	expr2	
	expr3	
Return	value	
RightShift	left	

下页继续

表 1 – 续上页

Node type	Attribute	Value
	right	
Slice	expr	
	flags	
	lower	
	upper	
Sliceobj	nodes	list of statements
Stmt	nodes	
Sub	left	
	right	
Subscript	expr	
	flags	
	subs	
TryExcept	body	
	handlers	
	else_	
TryFinally	body	
	final	
Tuple	nodes	
UnaryAdd	expr	
UnarySub	expr	
While	*test*	
	body	
	else_	
With	expr	
	vars	
	body	
Yield	value	

33.3.2 Assignment nodes

There is a collection of nodes used to represent assignments. Each assignment statement in the source code becomes a single `Assign` node in the AST. The `nodes` attribute is a list that contains a node for each assignment target. This is necessary because assignment can be chained, e.g. `a = b = 2`. Each *Node* in the list will be one of the following classes: `AssAttr`, `AssList`, `AssName`, or `AssTuple`.

Each target assignment node will describe the kind of object being assigned to: `AssName` for a simple name, e.g. `a = 1`. `AssAttr` for an attribute assigned, e.g. `a.x = 1`. `AssList` and `AssTuple` for list and tuple expansion respectively, e.g. `a, b, c = a_tuple`.

The target assignment nodes also have a `flags` attribute that indicates whether the node is being used for assignment or in a delete statement. The `AssName` is also used to represent a delete statement, e.g. `del x`.

When an expression contains several attribute references, an assignment or delete statement will contain only one `AssAttr` node –for the final attribute reference. The other attribute references will be represented as `Getattr` nodes in the `expr` attribute of the `AssAttr` instance.

33.3.3 Examples

This section shows several simple examples of ASTs for Python source code. The examples demonstrate how to use the `parse()` function, what the repr of an AST looks like, and how to access attributes of an AST node.

The first module defines a single function. Assume it is stored in `doublelib.py`.

```
"""This is an example module.

This is the docstring.
"""

def double(x):
    "Return twice the argument"
    return x * 2
```

In the interactive interpreter session below, I have reformatted the long AST reprs for readability. The AST reprs use unqualified class names. If you want to create an instance from a repr, you must import the class names from the *`compiler.ast`* module.

```
>>> import compiler
>>> mod = compiler.parseFile("doublelib.py")
>>> mod
Module('This is an example module.\n\nThis is the docstring.\n',
       Stmt([Function(None, 'double', ['x'], [], 0,
                      'Return twice the argument',
                      Stmt([Return(Mul((Name('x'), Const(2))))]))]))
>>> from compiler.ast import *
>>> Module('This is an example module.\n\nThis is the docstring.\n',
...     Stmt([Function(None, 'double', ['x'], [], 0,
...                    'Return twice the argument',
...                    Stmt([Return(Mul((Name('x'), Const(2))))]))]))
Module('This is an example module.\n\nThis is the docstring.\n',
       Stmt([Function(None, 'double', ['x'], [], 0,
                      'Return twice the argument',
                      Stmt([Return(Mul((Name('x'), Const(2))))]))]))
>>> mod.doc
'This is an example module.\n\nThis is the docstring.\n'
>>> for node in mod.node.nodes:
...     print node
...
Function(None, 'double', ['x'], [], 0, 'Return twice the argument',
         Stmt([Return(Mul((Name('x'), Const(2))))]))
>>> func = mod.node.nodes[0]
>>> func.code
Stmt([Return(Mul((Name('x'), Const(2))))])
```

33.4 Using Visitors to Walk ASTs

The visitor pattern is …The *compiler* package uses a variant on the visitor pattern that takes advantage of Python's introspection features to eliminate the need for much of the visitor's infrastructure.

The classes being visited do not need to be programmed to accept visitors. The visitor need only define visit methods for classes it is specifically interested in; a default visit method can handle the rest.

XXX The magic `visit()` method for visitors.

`compiler.visitor.`**`walk`**(*tree*, *visitor*[, *verbose*])

class `compiler.visitor.`**ASTVisitor**

The *ASTVisitor* is responsible for walking over the tree in the correct order. A walk begins with a call to *preorder()*. For each node, it checks the *visitor* argument to *preorder()* for a method named 'visitNodeType,' where NodeType is the name of the node's class, e.g. for a `While` node a `visitWhile()` would be called. If the method exists, it is called with the node as its first argument.

The visitor method for a particular node type can control how child nodes are visited during the walk. The *ASTVisitor* modifies the visitor argument by adding a visit method to the visitor; this method can be used to visit a particular child node. If no visitor is found for a particular node type, the *default()* method is called.

ASTVisitor objects have the following methods:

XXX describe extra arguments

default(*node*[, ...])

dispatch(*node*[, ...])

preorder(*tree*, *visitor*)

33.5 Bytecode Generation

The code generator is a visitor that emits bytecodes. Each visit method can call the `emit()` method to emit a new bytecode. The basic code generator is specialized for modules, classes, and functions. An assembler converts that emitted instructions to the low-level bytecode format. It handles things like generation of constant lists of code objects and calculation of jump offsets.

CHAPTER 34

杂项服务

本章中介绍的模块提供了所有 Python 版本中提供的各种杂项服务。这是一个概述：

34.1 formatter —通用格式化输出

This module supports two interface definitions, each with multiple implementations. The *formatter* interface is used by the `HTMLParser` class of the `htmllib` module, and the *writer* interface is required by the formatter interface.

Formatter objects transform an abstract flow of formatting events into specific output events on writer objects. Formatters manage several stack structures to allow various properties of a writer object to be changed and restored; writers need not be able to handle relative changes nor any sort of "change back" operation. Specific writer properties which may be controlled via formatter objects are horizontal alignment, font, and left margin indentations. A mechanism is provided which supports providing arbitrary, non-exclusive style settings to a writer as well. Additional interfaces facilitate formatting events which are not reversible, such as paragraph separation.

Writer objects encapsulate device interfaces. Abstract devices, such as file formats, are supported as well as physical devices. The provided implementations all work with abstract devices. The interface makes available mechanisms for setting the properties which formatter objects manage and inserting data into the output.

34.1.1 The Formatter Interface

Interfaces to create formatters are dependent on the specific formatter class being instantiated. The interfaces described below are the required interfaces which all formatters must support once initialized.

One data element is defined at the module level:

`formatter.`**`AS_IS`**
Value which can be used in the font specification passed to the `push_font()` method described below, or as the new value to any other `push_property()` method. Pushing the `AS_IS` value allows the corresponding `pop_property()` method to be called without having to track whether the property was changed.

The following attributes are defined for formatter instance objects:

formatter.**writer**
The writer instance with which the formatter interacts.

formatter.**end_paragraph**(*blanklines*)
Close any open paragraphs and insert at least *blanklines* before the next paragraph.

formatter.**add_line_break**()
Add a hard line break if one does not already exist. This does not break the logical paragraph.

formatter.**add_hor_rule**(**args*, ***kw*)
Insert a horizontal rule in the output. A hard break is inserted if there is data in the current paragraph, but the logical paragraph is not broken. The arguments and keywords are passed on to the writer's `send_line_break()` method.

formatter.**add_flowing_data**(*data*)
Provide data which should be formatted with collapsed whitespace. Whitespace from preceding and successive calls to *add_flowing_data()* is considered as well when the whitespace collapse is performed. The data which is passed to this method is expected to be word-wrapped by the output device. Note that any word-wrapping still must be performed by the writer object due to the need to rely on device and font information.

formatter.**add_literal_data**(*data*)
Provide data which should be passed to the writer unchanged. Whitespace, including newline and tab characters, are considered legal in the value of *data*.

formatter.**add_label_data**(*format*, *counter*)
Insert a label which should be placed to the left of the current left margin. This should be used for constructing bulleted or numbered lists. If the *format* value is a string, it is interpreted as a format specification for *counter*, which should be an integer. The result of this formatting becomes the value of the label; if *format* is not a string it is used as the label value directly. The label value is passed as the only argument to the writer's `send_label_data()` method. Interpretation of non-string label values is dependent on the associated writer.

Format specifications are strings which, in combination with a counter value, are used to compute label values. Each character in the format string is copied to the label value, with some characters recognized to indicate a transform on the counter value. Specifically, the character `'1'` represents the counter value formatter as an Arabic number, the characters `'A'` and `'a'` represent alphabetic representations of the counter value in upper and lower case, respectively, and `'I'` and `'i'` represent the counter value in Roman numerals, in upper and lower case. Note that the alphabetic and roman transforms require that the counter value be greater than zero.

formatter.**flush_softspace**()
Send any pending whitespace buffered from a previous call to *add_flowing_data()* to the associated writer object. This should be called before any direct manipulation of the writer object.

formatter.**push_alignment**(*align*)
Push a new alignment setting onto the alignment stack. This may be *AS_IS* if no change is desired. If the alignment value is changed from the previous setting, the writer's `new_alignment()` method is called with the *align* value.

formatter.**pop_alignment**()
Restore the previous alignment.

formatter.**push_font**(*(size*, *italic*, *bold*, *teletype)*)
Change some or all font properties of the writer object. Properties which are not set to *AS_IS* are set to the values passed in while others are maintained at their current settings. The writer's `new_font()` method is called with the fully resolved font specification.

formatter.**pop_font**()
Restore the previous font.

formatter.**push_margin**(*margin*)
Increase the number of left margin indentations by one, associating the logical tag *margin* with the new indentation.

The initial margin level is 0. Changed values of the logical tag must be true values; false values other than *AS_IS* are not sufficient to change the margin.

formatter.**pop_margin**()
Restore the previous margin.

formatter.**push_style**(**styles*)
Push any number of arbitrary style specifications. All styles are pushed onto the styles stack in order. A tuple representing the entire stack, including *AS_IS* values, is passed to the writer's new_styles() method.

formatter.**pop_style**([*n=1*])
Pop the last *n* style specifications passed to *push_style()*. A tuple representing the revised stack, including *AS_IS* values, is passed to the writer's new_styles() method.

formatter.**set_spacing**(*spacing*)
Set the spacing style for the writer.

formatter.**assert_line_data**([*flag=1*])
Inform the formatter that data has been added to the current paragraph out-of-band. This should be used when the writer has been manipulated directly. The optional *flag* argument can be set to false if the writer manipulations produced a hard line break at the end of the output.

34.1.2 Formatter Implementations

Two implementations of formatter objects are provided by this module. Most applications may use one of these classes without modification or subclassing.

class formatter.**NullFormatter**([*writer*])
A formatter which does nothing. If *writer* is omitted, a *NullWriter* instance is created. No methods of the writer are called by *NullFormatter* instances. Implementations should inherit from this class if implementing a writer interface but don't need to inherit any implementation.

class formatter.**AbstractFormatter**(*writer*)
The standard formatter. This implementation has demonstrated wide applicability to many writers, and may be used directly in most circumstances. It has been used to implement a full-featured World Wide Web browser.

34.1.3 The Writer Interface

Interfaces to create writers are dependent on the specific writer class being instantiated. The interfaces described below are the required interfaces which all writers must support once initialized. Note that while most applications can use the *AbstractFormatter* class as a formatter, the writer must typically be provided by the application.

writer.**flush**()
Flush any buffered output or device control events.

writer.**new_alignment**(*align*)
Set the alignment style. The *align* value can be any object, but by convention is a string or None, where None indicates that the writer's "preferred" alignment should be used. Conventional *align* values are 'left', 'center', 'right', and 'justify'.

writer.**new_font**(*font*)
Set the font style. The value of *font* will be None, indicating that the device's default font should be used, or a tuple of the form (size, italic, bold, teletype). Size will be a string indicating the size of font that should be used; specific strings and their interpretation must be defined by the application. The *italic*, *bold*, and *teletype* values are Boolean values specifying which of those font attributes should be used.

writer.**new_margin**(*margin*, *level*)
Set the margin level to the integer *level* and the logical tag to *margin*. Interpretation of the logical tag is at the writer's discretion; the only restriction on the value of the logical tag is that it not be a false value for non-zero values of *level*.

writer.**new_spacing**(*spacing*)
Set the spacing style to *spacing*.

writer.**new_styles**(*styles*)
Set additional styles. The *styles* value is a tuple of arbitrary values; the value *AS_IS* should be ignored. The *styles* tuple may be interpreted either as a set or as a stack depending on the requirements of the application and writer implementation.

writer.**send_line_break**()
Break the current line.

writer.**send_paragraph**(*blankline*)
Produce a paragraph separation of at least *blankline* blank lines, or the equivalent. The *blankline* value will be an integer. Note that the implementation will receive a call to *send_line_break()* before this call if a line break is needed; this method should not include ending the last line of the paragraph. It is only responsible for vertical spacing between paragraphs.

writer.**send_hor_rule**(**args*, ***kw*)
Display a horizontal rule on the output device. The arguments to this method are entirely application- and writer-specific, and should be interpreted with care. The method implementation may assume that a line break has already been issued via *send_line_break()*.

writer.**send_flowing_data**(*data*)
Output character data which may be word-wrapped and re-flowed as needed. Within any sequence of calls to this method, the writer may assume that spans of multiple whitespace characters have been collapsed to single space characters.

writer.**send_literal_data**(*data*)
Output character data which has already been formatted for display. Generally, this should be interpreted to mean that line breaks indicated by newline characters should be preserved and no new line breaks should be introduced. The data may contain embedded newline and tab characters, unlike data provided to the send_formatted_data() interface.

writer.**send_label_data**(*data*)
Set *data* to the left of the current left margin, if possible. The value of *data* is not restricted; treatment of non-string values is entirely application- and writer-dependent. This method will only be called at the beginning of a line.

34.1.4 Writer Implementations

Three implementations of the writer object interface are provided as examples by this module. Most applications will need to derive new writer classes from the *NullWriter* class.

class formatter.**NullWriter**
A writer which only provides the interface definition; no actions are taken on any methods. This should be the base class for all writers which do not need to inherit any implementation methods.

class formatter.**AbstractWriter**
A writer which can be used in debugging formatters, but not much else. Each method simply announces itself by printing its name and arguments on standard output.

class formatter.**DumbWriter**(*file=None*, *maxcol=72*)
Simple writer class which writes output on the file object passed in as *file* or, if *file* is None, on standard output. The

output is simply word-wrapped to the number of columns specified by *maxcol*. This class is suitable for reflowing a sequence of paragraphs.

CHAPTER 35

Windows 系统相关模块

本章节叙述的模块只在 Windows 平台上可用。

35.1 msilib —Read and write Microsoft Installer files

2.5 新版功能.

The *msilib* supports the creation of Microsoft Installer (`.msi`) files. Because these files often contain an embedded "cabinet" file (`.cab`), it also exposes an API to create CAB files. Support for reading `.cab` files is currently not implemented; read support for the `.msi` database is possible.

This package aims to provide complete access to all tables in an `.msi` file, therefore, it is a fairly low-level API. Two primary applications of this package are the *distutils* command `bdist_msi`, and the creation of Python installer package itself (although that currently uses a different version of `msilib`).

The package contents can be roughly split into four parts: low-level CAB routines, low-level MSI routines, higher-level MSI routines, and standard table structures.

`msilib.`**`FCICreate`**(*cabname*, *files*)
Create a new CAB file named *cabname*. *files* must be a list of tuples, each containing the name of the file on disk, and the name of the file inside the CAB file.

The files are added to the CAB file in the order they appear in the list. All files are added into a single CAB file, using the MSZIP compression algorithm.

Callbacks to Python for the various steps of MSI creation are currently not exposed.

`msilib.`**`UuidCreate`**()
Return the string representation of a new unique identifier. This wraps the Windows API functions `UuidCreate()` and `UuidToString()`.

`msilib.`**`OpenDatabase`**(*path*, *persist*)
Return a new database object by calling MsiOpenDatabase. *path* is the file name of the MSI file; *persist* can be one of the constants `MSIDBOPEN_CREATEDIRECT`, `MSIDBOPEN_CREATE`, `MSIDBOPEN_DIRECT`, `MSIDBOPEN_READONLY`, or `MSIDBOPEN_TRANSACT`, and may include the flag `MSIDBOPEN_PATCHFILE`.

See the Microsoft documentation for the meaning of these flags; depending on the flags, an existing database is opened, or a new one created.

msilib.**CreateRecord**(*count*)
Return a new record object by calling MSICreateRecord(). *count* is the number of fields of the record.

msilib.**init_database**(*name*, *schema*, *ProductName*, *ProductCode*, *ProductVersion*, *Manufacturer*)
Create and return a new database *name*, initialize it with *schema*, and set the properties *ProductName*, *ProductCode*, *ProductVersion*, and *Manufacturer*.

schema must be a module object containing tables and _Validation_records attributes; typically, *msilib.schema* should be used.

The database will contain just the schema and the validation records when this function returns.

msilib.**add_data**(*database*, *table*, *records*)
Add all *records* to the table named *table* in *database*.

The *table* argument must be one of the predefined tables in the MSI schema, e.g. 'Feature', 'File', 'Component', 'Dialog', 'Control', etc.

records should be a list of tuples, each one containing all fields of a record according to the schema of the table. For optional fields, None can be passed.

Field values can be int or long numbers, strings, or instances of the Binary class.

class msilib.**Binary**(*filename*)
Represents entries in the Binary table; inserting such an object using *add_data()* reads the file named *filename* into the table.

msilib.**add_tables**(*database*, *module*)
Add all table content from *module* to *database*. *module* must contain an attribute *tables* listing all tables for which content should be added, and one attribute per table that has the actual content.

This is typically used to install the sequence tables.

msilib.**add_stream**(*database*, *name*, *path*)
Add the file *path* into the _Stream table of *database*, with the stream name *name*.

msilib.**gen_uuid**()
Return a new UUID, in the format that MSI typically requires (i.e. in curly braces, and with all hexdigits in uppercase).

参见:

FCICreateFile UuidCreate UuidToString

35.1.1 Database Objects

Database.**OpenView**(*sql*)
Return a view object, by calling MSIDatabaseOpenView(). *sql* is the SQL statement to execute.

Database.**Commit**()
Commit the changes pending in the current transaction, by calling MSIDatabaseCommit().

Database.**GetSummaryInformation**(*count*)
Return a new summary information object, by calling MsiGetSummaryInformation(). *count* is the maximum number of updated values.

参见:

MSIDatabaseOpenView MSIDatabaseCommit MSIGetSummaryInformation

35.1.2 View Objects

View.**Execute**(*params*)
Execute the SQL query of the view, through MSIViewExecute(). If *params* is not None, it is a record describing actual values of the parameter tokens in the query.

View.**GetColumnInfo**(*kind*)
Return a record describing the columns of the view, through calling MsiViewGetColumnInfo(). *kind* can be either MSICOLINFO_NAMES or MSICOLINFO_TYPES.

View.**Fetch**()
Return a result record of the query, through calling MsiViewFetch().

View.**Modify**(*kind*, *data*)
Modify the view, by calling MsiViewModify(). *kind* can be one of MSIMODIFY_SEEK, MSIMODIFY_REFRESH, MSIMODIFY_INSERT, MSIMODIFY_UPDATE, MSIMODIFY_ASSIGN, MSIMODIFY_REPLACE, MSIMODIFY_MERGE, MSIMODIFY_DELETE, MSIMODIFY_INSERT_TEMPORARY, MSIMODIFY_VALIDATE, MSIMODIFY_VALIDATE_NEW, MSIMODIFY_VALIDATE_FIELD, or MSIMODIFY_VALIDATE_DELETE.

data must be a record describing the new data.

View.**Close**()
Close the view, through MsiViewClose().

参见:

MsiViewExecute MSIViewGetColumnInfo MsiViewFetch MsiViewModify MsiViewClose

35.1.3 Summary Information Objects

SummaryInformation.**GetProperty**(*field*)
Return a property of the summary, through MsiSummaryInfoGetProperty(). *field* is the name of the property, and can be one of the constants PID_CODEPAGE, PID_TITLE, PID_SUBJECT, PID_AUTHOR, PID_KEYWORDS, PID_COMMENTS, PID_TEMPLATE, PID_LASTAUTHOR, PID_REVNUMBER, PID_LASTPRINTED, PID_CREATE_DTM, PID_LASTSAVE_DTM, PID_PAGECOUNT, PID_WORDCOUNT, PID_CHARCOUNT, PID_APPNAME, or PID_SECURITY.

SummaryInformation.**GetPropertyCount**()
Return the number of summary properties, through MsiSummaryInfoGetPropertyCount().

SummaryInformation.**SetProperty**(*field*, *value*)
Set a property through MsiSummaryInfoSetProperty(). *field* can have the same values as in *GetProperty()*, *value* is the new value of the property. Possible value types are integer and string.

SummaryInformation.**Persist**()
Write the modified properties to the summary information stream, using MsiSummaryInfoPersist().

参见:

MsiSummaryInfoGetProperty MsiSummaryInfoGetPropertyCount MsiSummaryInfoSetProperty MsiSummaryInfoPersist

35.1.4 Record Objects

Record.**GetFieldCount**()
Return the number of fields of the record, through MsiRecordGetFieldCount().

Record.**GetInteger**(*field*)
Return the value of *field* as an integer where possible. *field* must be an integer.

Record.**GetString**(*field*)
Return the value of *field* as a string where possible. *field* must be an integer.

Record.**SetString**(*field*, *value*)
Set *field* to *value* through MsiRecordSetString(). *field* must be an integer; *value* a string.

Record.**SetStream**(*field*, *value*)
Set *field* to the contents of the file named *value*, through MsiRecordSetStream(). *field* must be an integer; *value* a string.

Record.**SetInteger**(*field*, *value*)
Set *field* to *value* through MsiRecordSetInteger(). Both *field* and *value* must be an integer.

Record.**ClearData**()
Set all fields of the record to 0, through MsiRecordClearData().

参见:

MsiRecordGetFieldCount MsiRecordSetString MsiRecordSetStream MsiRecordSetInteger MsiRecordClear

35.1.5 Errors

All wrappers around MSI functions raise MSIError; the string inside the exception will contain more detail.

35.1.6 CAB Objects

class msilib.**CAB**(*name*)
The class *CAB* represents a CAB file. During MSI construction, files will be added simultaneously to the Files table, and to a CAB file. Then, when all files have been added, the CAB file can be written, then added to the MSI file.

name is the name of the CAB file in the MSI file.

append(*full*, *file*, *logical*)
Add the file with the pathname *full* to the CAB file, under the name *logical*. If there is already a file named *logical*, a new file name is created.

Return the index of the file in the CAB file, and the new name of the file inside the CAB file.

commit(*database*)
Generate a CAB file, add it as a stream to the MSI file, put it into the Media table, and remove the generated file from the disk.

35.1.7 Directory Objects

class msilib.**Directory**(*database*, *cab*, *basedir*, *physical*, *logical*, *default*[, *componentflags*])

Create a new directory in the Directory table. There is a current component at each point in time for the directory, which is either explicitly created through *start_component()*, or implicitly when files are added for the first time. Files are added into the current component, and into the cab file. To create a directory, a base directory object needs to be specified (can be None), the path to the physical directory, and a logical directory name. *default* specifies the DefaultDir slot in the directory table. *componentflags* specifies the default flags that new components get.

start_component([*component*[, *feature*[, *flags*[, *keyfile*[, *uuid*]]]]])

Add an entry to the Component table, and make this component the current component for this directory. If no component name is given, the directory name is used. If no *feature* is given, the current feature is used. If no *flags* are given, the directory's default flags are used. If no *keyfile* is given, the KeyPath is left null in the Component table.

add_file(*file*[, *src*[, *version*[, *language*]]])

Add a file to the current component of the directory, starting a new one if there is no current component. By default, the file name in the source and the file table will be identical. If the *src* file is specified, it is interpreted relative to the current directory. Optionally, a *version* and a *language* can be specified for the entry in the File table.

glob(*pattern*[, *exclude*])

Add a list of files to the current component as specified in the glob pattern. Individual files can be excluded in the *exclude* list.

remove_pyc()

Remove .pyc/.pyo files on uninstall.

参见:

Directory Table File Table Component Table FeatureComponents Table

35.1.8 相关特性

class msilib.**Feature**(*database*, *id*, *title*, *desc*, *display*[, *level=1*[, *parent*[, *directory*[, *attributes=0*]]]])

Add a new record to the Feature table, using the values *id*, *parent.id*, *title*, *desc*, *display*, *level*, *directory*, and *attributes*. The resulting feature object can be passed to the start_component() method of *Directory*.

set_current()

Make this feature the current feature of *msilib*. New components are automatically added to the default feature, unless a feature is explicitly specified.

参见:

Feature Table

35.1.9 GUI classes

msilib provides several classes that wrap the GUI tables in an MSI database. However, no standard user interface is provided; use `bdist_msi` to create MSI files with a user-interface for installing Python packages.

class `msilib.`**Control**(*dlg*, *name*)
Base class of the dialog controls. *dlg* is the dialog object the control belongs to, and *name* is the control's name.

event(*event*, *argument*[, *condition=1*[, *ordering*]])
Make an entry into the `ControlEvent` table for this control.

mapping(*event*, *attribute*)
Make an entry into the `EventMapping` table for this control.

condition(*action*, *condition*)
Make an entry into the `ControlCondition` table for this control.

class `msilib.`**RadioButtonGroup**(*dlg*, *name*, *property*)
Create a radio button control named *name*. *property* is the installer property that gets set when a radio button is selected.

add(*name*, *x*, *y*, *width*, *height*, *text*[, *value*])
Add a radio button named *name* to the group, at the coordinates *x*, *y*, *width*, *height*, and with the label *text*. If *value* is omitted, it defaults to *name*.

class `msilib.`**Dialog**(*db*, *name*, *x*, *y*, *w*, *h*, *attr*, *title*, *first*, *default*, *cancel*)
Return a new *Dialog* object. An entry in the `Dialog` table is made, with the specified coordinates, dialog attributes, title, name of the first, default, and cancel controls.

control(*name*, *type*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *control_next*, *help*)
Return a new *Control* object. An entry in the `Control` table is made with the specified parameters.

This is a generic method; for specific types, specialized methods are provided.

text(*name*, *x*, *y*, *width*, *height*, *attributes*, *text*)
Add and return a `Text` control.

bitmap(*name*, *x*, *y*, *width*, *height*, *text*)
Add and return a `Bitmap` control.

line(*name*, *x*, *y*, *width*, *height*)
Add and return a `Line` control.

pushbutton(*name*, *x*, *y*, *width*, *height*, *attributes*, *text*, *next_control*)
Add and return a `PushButton` control.

radiogroup(*name*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *next_control*)
Add and return a `RadioButtonGroup` control.

checkbox(*name*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *next_control*)
Add and return a `CheckBox` control.

参见:

Dialog Table Control Table Control Types ControlCondition Table ControlEvent Table EventMapping Table RadioButton Table

35.1.10 Precomputed tables

msilib provides a few subpackages that contain only schema and table definitions. Currently, these definitions are based on MSI version 2.0.

msilib.**schema**
: This is the standard MSI schema for MSI 2.0, with the *tables* variable providing a list of table definitions, and *_Validation_records* providing the data for MSI validation.

msilib.**sequence**
: This module contains table contents for the standard sequence tables: *AdminExecuteSequence*, *AdminUISequence*, *AdvtExecuteSequence*, *InstallExecuteSequence*, and *InstallUISequence*.

msilib.**text**
: This module contains definitions for the UIText and ActionText tables, for the standard installer actions.

35.2 msvcrt —Useful routines from the MS VC++ runtime

These functions provide access to some useful capabilities on Windows platforms. Some higher-level modules use these functions to build the Windows implementations of their services. For example, the *getpass* module uses this in the implementation of the *getpass()* function.

Further documentation on these functions can be found in the Platform API documentation.

The module implements both the normal and wide char variants of the console I/O api. The normal API deals only with ASCII characters and is of limited use for internationalized applications. The wide char API should be used where ever possible.

35.2.1 File Operations

msvcrt.**locking**(*fd*, *mode*, *nbytes*)
: Lock part of a file based on file descriptor *fd* from the C runtime. Raises *IOError* on failure. The locked region of the file extends from the current file position for *nbytes* bytes, and may continue beyond the end of the file. *mode* must be one of the `LK_*` constants listed below. Multiple regions in a file may be locked at the same time, but may not overlap. Adjacent regions are not merged; they must be unlocked individually.

msvcrt.**LK_LOCK**
msvcrt.**LK_RLCK**
: Locks the specified bytes. If the bytes cannot be locked, the program immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, *IOError* is raised.

msvcrt.**LK_NBLCK**
msvcrt.**LK_NBRLCK**
: Locks the specified bytes. If the bytes cannot be locked, *IOError* is raised.

msvcrt.**LK_UNLCK**
: Unlocks the specified bytes, which must have been previously locked.

msvcrt.**setmode**(*fd*, *flags*)
: Set the line-end translation mode for the file descriptor *fd*. To set it to text mode, *flags* should be *os.O_TEXT*; for binary, it should be *os.O_BINARY*.

msvcrt.**open_osfhandle**(*handle*, *flags*)
: Create a C runtime file descriptor from the file handle *handle*. The *flags* parameter should be a bitwise OR of *os.O_APPEND*, *os.O_RDONLY*, and *os.O_TEXT*. The returned file descriptor may be used as a parameter to *os.fdopen()* to create a file object.

msvcrt.**get_osfhandle**(*fd*)
Return the file handle for the file descriptor *fd*. Raises *IOError* if *fd* is not recognized.

35.2.2 Console I/O

msvcrt.**kbhit**()
Return true if a keypress is waiting to be read.

msvcrt.**getch**()
Read a keypress and return the resulting character. Nothing is echoed to the console. This call will block if a keypress is not already available, but will not wait for `Enter` to be pressed. If the pressed key was a special function key, this will return `'\000'` or `'\xe0'`; the next call will return the keycode. The `Control-C` keypress cannot be read with this function.

msvcrt.**getwch**()
Wide char variant of *getch()*, returning a Unicode value.

2.6 新版功能.

msvcrt.**getche**()
Similar to *getch()*, but the keypress will be echoed if it represents a printable character.

msvcrt.**getwche**()
Wide char variant of *getche()*, returning a Unicode value.

2.6 新版功能.

msvcrt.**putch**(*char*)
Print the character *char* to the console without buffering.

msvcrt.**putwch**(*unicode_char*)
Wide char variant of *putch()*, accepting a Unicode value.

2.6 新版功能.

msvcrt.**ungetch**(*char*)
Cause the character *char* to be "pushed back" into the console buffer; it will be the next character read by *getch()* or *getche()*.

msvcrt.**ungetwch**(*unicode_char*)
Wide char variant of *ungetch()*, accepting a Unicode value.

2.6 新版功能.

35.2.3 Other Functions

msvcrt.**heapmin**()
Force the `malloc()` heap to clean itself up and return unused blocks to the operating system. On failure, this raises *IOError*.

35.3 _winreg —Windows registry access

注解： The *_winreg* module has been renamed to `winreg` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

2.0 新版功能.

These functions expose the Windows registry API to Python. Instead of using an integer as the registry handle, a *handle object* is used to ensure that the handles are closed correctly, even if the programmer neglects to explicitly close them.

This module offers the following functions:

`_winreg.`**`CloseKey`**(*hkey*)
Closes a previously opened registry key. The *hkey* argument specifies a previously opened key.

注解： If *hkey* is not closed using this method (or via *hkey.Close()*), it is closed when the *hkey* object is destroyed by Python.

`_winreg.`**`ConnectRegistry`**(*computer_name*, *key*)
Establishes a connection to a predefined registry handle on another computer, and returns a *handle object*.

computer_name is the name of the remote computer, of the form `r"\\computername"`. If `None`, the local computer is used.

key is the predefined handle to connect to.

The return value is the handle of the opened key. If the function fails, a *WindowsError* exception is raised.

`_winreg.`**`CreateKey`**(*key*, *sub_key*)
Creates or opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the key this method opens or creates.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, a *WindowsError* exception is raised.

`_winreg.`**`CreateKeyEx`**(*key*, *sub_key*[, *res*[, *sam*]])
Creates or opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the key this method opens or creates.

res is a reserved integer, and must be zero. The default is zero.

sam is an integer that specifies an access mask that describes the desired security access for the key. Default is *KEY_ALL_ACCESS*. See *Access Rights* for other allowed values.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, a *WindowsError* exception is raised.

2.7 新版功能.

_winreg.**DeleteKey**(*key*, *sub_key*)
Deletes the specified key.

key is an already open key, or any one of the predefined *HKEY_* constants*.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, a `WindowsError` exception is raised.

_winreg.**DeleteKeyEx**(*key*, *sub_key*[, *sam*[, *res*]])
Deletes the specified key.

注解: The `DeleteKeyEx()` function is implemented with the RegDeleteKeyEx Windows API function, which is specific to 64-bit versions of Windows. See the RegDeleteKeyEx documentation.

---

key is an already open key, or any one of the predefined *HKEY_* constants*.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

res is a reserved integer, and must be zero. The default is zero.

sam is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_WOW64_64KEY`. See *Access Rights* for other allowed values.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, a `WindowsError` exception is raised.

On unsupported Windows versions, `NotImplementedError` is raised.

2.7 新版功能.

_winreg.**DeleteValue**(*key*, *value*)
Removes a named value from a registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value is a string that identifies the value to remove.

_winreg.**EnumKey**(*key*, *index*)
Enumerates subkeys of an open registry key, returning a string.

key is an already open key, or any one of the predefined *HKEY_* constants*.

index is an integer that identifies the index of the key to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly until a `WindowsError` exception is raised, indicating, no more values are available.

_winreg.**EnumValue**(*key*, *index*)
Enumerates values of an open registry key, returning a tuple.

key is an already open key, or any one of the predefined *HKEY_* constants*.

index is an integer that identifies the index of the value to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly, until a *WindowsError* exception is raised, indicating no more values.

The result is a tuple of 3 items:

Index	Meaning
0	A string that identifies the value name
1	An object that holds the value data, and whose type depends on the underlying registry type
2	An integer that identifies the type of the value data (see table in docs for *SetValueEx()*)

_winreg.**ExpandEnvironmentStrings**(*unicode*)

Expands environment variable placeholders %NAME% in unicode strings like *REG_EXPAND_SZ*:

```
>>> ExpandEnvironmentStrings(u"%windir%")
u"C:\\Windows"
```

2.6 新版功能.

_winreg.**FlushKey**(*key*)

Writes all the attributes of a key to the registry.

key is an already open key, or one of the predefined *HKEY_* constants*.

It is not necessary to call *FlushKey()* to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike *CloseKey()*, the *FlushKey()* method returns only when all the data has been written to the registry. An application should only call *FlushKey()* if it requires absolute certainty that registry changes are on disk.

注解： If you don't know whether a *FlushKey()* call is required, it probably isn't.

_winreg.**LoadKey**(*key*, *sub_key*, *file_name*)

Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

key is a handle returned by *ConnectRegistry()* or one of the constants *HKEY_USERS* or *HKEY_LOCAL_MACHINE*.

sub_key is a string that identifies the subkey to load.

file_name is the name of the file to load registry data from. This file must have been created with the *SaveKey()* function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to *LoadKey()* fails if the calling process does not have the SE_RESTORE_PRIVILEGE privilege. Note that privileges are different from permissions –see the RegLoadKey documentation for more details.

If *key* is a handle returned by *ConnectRegistry()*, then the path specified in *file_name* is relative to the remote computer.

_winreg.**OpenKey**(*key*, *sub_key*[, *res*[, *sam*]])

Opens the specified key, returning a *handle object*.

key is an already open key, or any one of the predefined *HKEY_* constants*.

sub_key is a string that identifies the sub_key to open.

res is a reserved integer, and must be zero. The default is zero.

sam is an integer that specifies an access mask that describes the desired security access for the key. Default is *KEY_READ*. See *Access Rights* for other allowed values.

The result is a new handle to the specified key.

If the function fails, *WindowsError* is raised.

_winreg.**OpenKeyEx**()
The functionality of *OpenKeyEx()* is provided via *OpenKey()*, by the use of default arguments.

_winreg.**QueryInfoKey**(*key*)
Returns information about a key, as a tuple.

key is an already open key, or one of the predefined *HKEY_* constants*.

The result is a tuple of 3 items:

Index	Meaning
0	An integer giving the number of sub keys this key has.
1	An integer giving the number of values this key has.
2	A long integer giving when the key was last modified (if available) as 100's of nanoseconds since Jan 1, 1601.

_winreg.**QueryValue**(*key*, *sub_key*)
Retrieves the unnamed value for a key, as a string.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that holds the name of the subkey with which the value is associated. If this parameter is None or empty, the function retrieves the value set by the *SetValue()* method for the key identified by *key*.

Values in the registry have name, type, and data components. This method retrieves the data for a key's first value that has a NULL name. But the underlying API call doesn't return the type, so always use *QueryValueEx()* if possible.

_winreg.**QueryValueEx**(*key*, *value_name*)
Retrieves the type and data for a specified value name associated with an open registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value_name is a string indicating the value to query.

The result is a tuple of 2 items:

Index	Meaning
0	The value of the registry item.
1	An integer giving the registry type for this value (see table in docs for *SetValueEx()*)

_winreg.**SaveKey**(*key*, *file_name*)
Saves the specified key, and all its subkeys to the specified file.

key is an already open key, or one of the predefined *HKEY_* constants*.

file_name is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the *LoadKey()* method.

If *key* represents a key on a remote computer, the path described by *file_name* is relative to the remote computer. The caller of this method must possess the SeBackupPrivilege security privilege. Note that privileges are different than permissions –see the Conflicts Between User Rights and Permissions documentation for more details.

This function passes NULL for *security_attributes* to the API.

_winreg.**SetValue**(*key*, *sub_key*, *type*, *value*)
Associates a value with a specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the subkey with which the value is associated.

type is an integer that specifies the type of the data. Currently this must be `REG_SZ`, meaning only strings are supported. Use the `SetValueEx()` function for support for other data types.

value is a string that specifies the new value.

If the key specified by the *sub_key* parameter does not exist, the SetValue function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the *key* parameter must have been opened with `KEY_SET_VALUE` access.

`_winreg.`**`SetValueEx`**(*key*, *value_name*, *reserved*, *type*, *value*)
Stores data in the value field of an open registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value_name is a string that names the subkey with which the value is associated.

type is an integer that specifies the type of the data. See *Value Types* for the available types.

reserved can be anything –zero is always passed to the API.

value is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the key parameter must have been opened with `KEY_SET_VALUE` access.

To open the key, use the `CreateKey()` or `OpenKey()` methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

`_winreg.`**`DisableReflectionKey`**(*key*)
Disables registry reflection for 32-bit processes running on a 64-bit operating system.

key is an already open key, or one of the predefined *HKEY_* constants*.

Will generally raise `NotImplementedError` if executed on a 32-bit operating system.

If the key is not on the reflection list, the function succeeds but has no effect. Disabling reflection for a key does not affect reflection of any subkeys.

`_winreg.`**`EnableReflectionKey`**(*key*)
Restores registry reflection for the specified disabled key.

key is an already open key, or one of the predefined *HKEY_* constants*.

Will generally raise `NotImplementedError` if executed on a 32-bit operating system.

Restoring reflection for a key does not affect reflection of any subkeys.

`_winreg.`**`QueryReflectionKey`**(*key*)
Determines the reflection state for the specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

Returns `True` if reflection is disabled.

Will generally raise `NotImplementedError` if executed on a 32-bit operating system.

35.3.1 Constants

The following constants are defined for use in many *_winreg* functions.

HKEY_* Constants

_winreg.**HKEY_CLASSES_ROOT**
Registry entries subordinate to this key define types (or classes) of documents and the properties associated with those types. Shell and COM applications use the information stored under this key.

_winreg.**HKEY_CURRENT_USER**
Registry entries subordinate to this key define the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences.

_winreg.**HKEY_LOCAL_MACHINE**
Registry entries subordinate to this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software.

_winreg.**HKEY_USERS**
Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

_winreg.**HKEY_PERFORMANCE_DATA**
Registry entries subordinate to this key allow you to access performance data. The data is not actually stored in the registry; the registry functions cause the system to collect the data from its source.

_winreg.**HKEY_CURRENT_CONFIG**
Contains information about the current hardware profile of the local computer system.

_winreg.**HKEY_DYN_DATA**
This key is not used in versions of Windows after 98.

Access Rights

For more information, see Registry Key Security and Access.

_winreg.**KEY_ALL_ACCESS**
Combines the STANDARD_RIGHTS_REQUIRED, *KEY_QUERY_VALUE*, *KEY_SET_VALUE*, *KEY_CREATE_SUB_KEY*, *KEY_ENUMERATE_SUB_KEYS*, *KEY_NOTIFY*, and *KEY_CREATE_LINK* access rights.

_winreg.**KEY_WRITE**
Combines the STANDARD_RIGHTS_WRITE, *KEY_SET_VALUE*, and *KEY_CREATE_SUB_KEY* access rights.

_winreg.**KEY_READ**
Combines the STANDARD_RIGHTS_READ, *KEY_QUERY_VALUE*, *KEY_ENUMERATE_SUB_KEYS*, and *KEY_NOTIFY* values.

_winreg.**KEY_EXECUTE**
Equivalent to *KEY_READ*.

_winreg.**KEY_QUERY_VALUE**
Required to query the values of a registry key.

_winreg.**KEY_SET_VALUE**
Required to create, delete, or set a registry value.

_winreg.**KEY_CREATE_SUB_KEY**
Required to create a subkey of a registry key.

_winreg.**KEY_ENUMERATE_SUB_KEYS**
Required to enumerate the subkeys of a registry key.

_winreg.**KEY_NOTIFY**
Required to request change notifications for a registry key or for subkeys of a registry key.

_winreg.**KEY_CREATE_LINK**
Reserved for system use.

64-bit Specific

For more information, see Accessing an Alternate Registry View.

_winreg.**KEY_WOW64_64KEY**
Indicates that an application on 64-bit Windows should operate on the 64-bit registry view.

_winreg.**KEY_WOW64_32KEY**
Indicates that an application on 64-bit Windows should operate on the 32-bit registry view.

Value Types

For more information, see Registry Value Types.

_winreg.**REG_BINARY**
Binary data in any form.

_winreg.**REG_DWORD**
32-bit number.

_winreg.**REG_DWORD_LITTLE_ENDIAN**
A 32-bit number in little-endian format.

_winreg.**REG_DWORD_BIG_ENDIAN**
A 32-bit number in big-endian format.

_winreg.**REG_EXPAND_SZ**
Null-terminated string containing references to environment variables (`%PATH%`).

_winreg.**REG_LINK**
A Unicode symbolic link.

_winreg.**REG_MULTI_SZ**
A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)

_winreg.**REG_NONE**
No defined value type.

_winreg.**REG_RESOURCE_LIST**
A device-driver resource list.

_winreg.**REG_FULL_RESOURCE_DESCRIPTOR**
A hardware setting.

_winreg.**REG_RESOURCE_REQUIREMENTS_LIST**
A hardware resource list.

_winreg.**REG_SZ**
A null-terminated string.

35.3.2 Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee cleanup, you can call either the *Close()* method on the object, or the *CloseKey()* function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the handle object is encouraged.

Handle objects provide semantics for __nonzero__() –thus:

```
if handle:
    print "Yes"
```

will print Yes if the handle is currently valid (has not been closed or detached).

The object also support comparison semantics, so handle objects will compare true if they both reference the same underlying Windows handle value.

Handle objects can be converted to an integer (e.g., using the built-in *int()* function), in which case the underlying Windows handle value is returned. You can also use the *Detach()* method to return the integer handle, and also disconnect the Windows handle from the handle object.

PyHKEY.**Close**()
Closes the underlying Windows handle.

If the handle is already closed, no error is raised.

PyHKEY.**Detach**()
Detaches the Windows handle from the handle object.

The result is an integer (or long on 64 bit Windows) that holds the value of the handle before it is detached. If the handle is already detached or closed, this will return zero.

After calling this function, the handle is effectively invalidated, but the handle is not closed. You would call this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.

PyHKEY.**__enter__**()
PyHKEY.**__exit__**(**exc_info*)
The HKEY object implements __enter__() and __exit__() and thus supports the context protocol for the with statement:

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...  # work with key
```

will automatically close *key* when control leaves the with block.

2.6 新版功能.

35.4 winsound —Sound-playing interface for Windows

1.5.2 新版功能.

The *winsound* module provides access to the basic sound-playing machinery provided by Windows platforms. It includes functions and several constants.

winsound.**Beep**(*frequency*, *duration*)
Beep the PC's speaker. The *frequency* parameter specifies frequency, in hertz, of the sound, and must be in the range 37 through 32,767. The *duration* parameter specifies the number of milliseconds the sound should last. If the system is not able to beep the speaker, *RuntimeError* is raised.

1.6 新版功能.

winsound.**PlaySound**(*sound*, *flags*)
Call the underlying `PlaySound()` function from the Platform API. The *sound* parameter may be a filename, audio data as a string, or `None`. Its interpretation depends on the value of *flags*, which can be a bitwise ORed combination of the constants described below. If the *sound* parameter is `None`, any currently playing waveform sound is stopped. If the system indicates an error, *RuntimeError* is raised.

winsound.**MessageBeep**([*type=MB_OK*])
Call the underlying `MessageBeep()` function from the Platform API. This plays a sound as specified in the registry. The *type* argument specifies which sound to play; possible values are `-1`, `MB_ICONASTERISK`, `MB_ICONEXCLAMATION`, `MB_ICONHAND`, `MB_ICONQUESTION`, and `MB_OK`, all described below. The value `-1` produces a "simple beep"; this is the final fallback if a sound cannot be played otherwise.

2.3 新版功能.

winsound.**SND_FILENAME**
The *sound* parameter is the name of a WAV file. Do not use with *SND_ALIAS*.

winsound.**SND_ALIAS**
The *sound* parameter is a sound association name from the registry. If the registry contains no such name, play the system default sound unless *SND_NODEFAULT* is also specified. If no default sound is registered, raise *RuntimeError*. Do not use with *SND_FILENAME*.

All Win32 systems support at least the following; most systems support many more:

PlaySound() name	Corresponding Control Panel Sound name
`'SystemAsterisk'`	Asterisk
`'SystemExclamation'`	Exclamation
`'SystemExit'`	Exit Windows
`'SystemHand'`	Critical Stop
`'SystemQuestion'`	Question

例如

```
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

winsound.**SND_LOOP**
Play the sound repeatedly. The *SND_ASYNC* flag must also be used to avoid blocking. Cannot be used with *SND_MEMORY*.

winsound.**SND_MEMORY**
The *sound* parameter to *PlaySound()* is a memory image of a WAV file, as a string.

> **注解：** This module does not support playing from a memory image asynchronously, so a combination of this flag and *SND_ASYNC* will raise *RuntimeError*.

winsound.**SND_PURGE**
Stop playing all instances of the specified sound.

> **注解：** This flag is not supported on modern Windows platforms.

winsound.**SND_ASYNC**
Return immediately, allowing sounds to play asynchronously.

winsound.**SND_NODEFAULT**
If the specified sound cannot be found, do not play the system default sound.

winsound.**SND_NOSTOP**
Do not interrupt sounds currently playing.

winsound.**SND_NOWAIT**
Return immediately if the sound driver is busy.

> **注解：** This flag is not supported on modern Windows platforms.

winsound.**MB_ICONASTERISK**
Play the `SystemDefault` sound.

winsound.**MB_ICONEXCLAMATION**
Play the `SystemExclamation` sound.

winsound.**MB_ICONHAND**
Play the `SystemHand` sound.

winsound.**MB_ICONQUESTION**
Play the `SystemQuestion` sound.

winsound.**MB_OK**
Play the `SystemDefault` sound.

CHAPTER 36

Unix 专有服务

本章描述的模块提供了 Unix 操作系统独有特性的接口，在某些情况下也适用于它的某些或许多衍生版。以下为模块概览：

36.1 posix —最常见的 POSIX 系统调用

此模块提供了对基于 C 标准和 POSIX 标准（一种稍加修改的 Unix 接口）进行标准化的系统功能的访问。

请勿直接导入此模块。而应导入*os*模块，它提供了此接口的 可移植版本。在 Unix 上，*os*模块提供了*posix*接口的一个超集。在非 Unix 操作系统上*posix*模块将不可用，但会通过*os*接口提供它的一个可用子集。一旦导入了*os*，用它替代*posix*时就 没有性能惩罚。此外，*os*还提供了一些附加功能，例如在 os.environ 中的某个条目被修改时会自动调用*putenv()*。

错误将作为异常被报告；对于类型错误会给出普通异常，而系统调用所报告的异常则会引发*OSError*。

36.1.1 大文件支持

Several operating systems (including AIX, HP-UX, Irix and Solaris) provide support for files that are larger than 2 GB from a C programming model where `int` and `long` are 32-bit values. This is typically accomplished by defining the relevant size and offset types as 64-bit values. Such files are sometimes referred to as *large files*.

Large file support is enabled in Python when the size of an `off_t` is larger than a `long` and the `long long` type is available and is at least as large as an `off_t`. Python longs are then used to represent file sizes, offsets and other values that can exceed the range of a Python int. It may be necessary to configure and compile Python with certain compiler flags to enable this mode. For example, it is enabled by default with recent versions of Irix, but with Solaris 2.6 and 2.7 you need to do something like:

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \
        ./configure
```

在支持大文件的 Linux 系统中，可以这样做：

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \
        ./configure
```

36.1.2 重要的模块内容

除了*os*模块文档已说明的许多函数，*posix*还定义了下列数据项:

posix.**environ**
A dictionary representing the string environment at the time the interpreter was started. For example, environ['HOME'] is the pathname of your home directory, equivalent to getenv("HOME") in C.

修改此字典不会影响由*execv()*, *popen()*或*system()*所传入的字符串环境；如果你需要修改环境，请将 environ 传给*execve()*或者为*system()*或*popen()*的命令字符串添加变量赋值和 export 语句。

注解: The *os* module provides an alternate implementation of environ which updates the environment on modification. Note also that updating os.environ will render this dictionary obsolete. Use of the *os* module version of this is recommended over direct access to the *posix* module.

36.2 pwd —用户密码数据库

此模块可以访问 Unix 用户账户名及密码数据库，在所有 Unix 版本上均可使用。

密码数据库中的条目以元组对象返回，属性对应 passwd 中的结构（属性如下所示，可参考 <pwd.h>):

索引	属性	含义
0	pw_name	登录名
1	pw_passwd	密码，可能已经加密
2	pw_uid	用户 ID 数值
3	pw_gid	组 ID 数值
4	pw_gecos	用户名或备注
5	pw_dir	用户主目录
6	pw_shell	用户的命令解释器

其中 uid 和 gid 是整数，其他是字符串，如果找不到对应的项目，抛出*KeyError*异常。

注解: 传统的 Unix 系统中，pw_passwd 的值通常使用 DES 导出的算法加密（参阅*crypt*模块）。不过现在的 unix 系统使用 *影子密码*系统。在这些 unix 上，*pw_passwd* 只包含星号（'*'）或字母（'x'），而加密的密码存储在文件 /etc/shadow 中，此文件不是全局可读的。在 *pw_passwd* 中是否包含有用信息是系统相关的。如果可以访问到加密的密码，就需要使用*spwd*模块了。

本模块定义如下内容:

pwd.**getpwuid**(*uid*)
给定用户的数值 ID，返回密码数据库的对应项目。

pwd.**getpwnam**(*name*)
给定用户名，返回密码数据库的对应项目。

pwd.**getpwall**()
返回密码数据库中所有项目的列表，顺序不是固定的。

参见:

模块*grp* 针对用户组数据库的接口，与本模块类似。

模块*spwd* 针对影子密码数据库的接口，与本模块类似。

36.3 spwd —The shadow password database

2.5 新版功能.

This module provides access to the Unix shadow password database. It is available on various Unix versions.

You must have enough privileges to access the shadow password database (this usually means you have to be root).

Shadow password database entries are reported as a tuple-like object, whose attributes correspond to the members of the spwd structure (Attribute field below, see <shadow.h>):

索引	属性	含义
0	sp_nam	登录名
1	sp_pwd	Encrypted password
2	sp_lstchg	Date of last change
3	sp_min	Minimal number of days between changes
4	sp_max	Maximum number of days between changes
5	sp_warn	Number of days before password expires to warn user about it
6	sp_inact	Number of days after password expires until account is blocked
7	sp_expire	Number of days since 1970-01-01 until account is disabled
8	sp_flag	Reserved

The sp_nam and sp_pwd items are strings, all others are integers. *KeyError* is raised if the entry asked for cannot be found.

It defines the following items:

spwd.**getspnam**(*name*)
Return the shadow password database entry for the given user name.

spwd.**getspall**()
Return a list of all available shadow password database entries, in arbitrary order.

参见:

模块*grp* 针对用户组数据库的接口，与本模块类似。

Module *pwd* An interface to the normal password database, similar to this.

36.4 grp —The group database

This module provides access to the Unix group database. It is available on all Unix versions.

Group database entries are reported as a tuple-like object, whose attributes correspond to the members of the group structure (Attribute field below, see <pwd.h>):

索引	属性	含义
0	gr_name	the name of the group
1	gr_passwd	the (encrypted) group password; often empty
2	gr_gid	the numerical group ID
3	gr_mem	all the group member’ s user names

The gid is an integer, name and password are strings, and the member list is a list of strings. (Note that most users are not explicitly listed as members of the group they are in according to the password database. Check both databases to get complete membership information. Also note that a gr_name that starts with a + or − is likely to be a YP/NIS reference and may not be accessible via *getgrnam()* or *getgrgid()*.)

本模块定义如下内容：

grp.**getgrgid**(*gid*)
Return the group database entry for the given numeric group ID. *KeyError* is raised if the entry asked for cannot be found.

grp.**getgrnam**(*name*)
Return the group database entry for the given group name. *KeyError* is raised if the entry asked for cannot be found.

grp.**getgrall**()
Return a list of all available group entries, in arbitrary order.

参见：

Module *pwd* An interface to the user database, similar to this.

模块*spwd* 针对影子密码数据库的接口，与本模块类似。

36.5 crypt —Function to check Unix passwords

This module implements an interface to the *crypt(3)* routine, which is a one-way hash function based upon a modified DES algorithm; see the Unix man page for further details. Possible uses include allowing Python scripts to accept typed passwords from the user, or attempting to crack Unix passwords with a dictionary.

Notice that the behavior of this module depends on the actual implementation of the *crypt(3)* routine in the running system. Therefore, any extensions available on the current implementation will also be available on this module.

crypt.**crypt**(*word*, *salt*)
word will usually be a user’ s password as typed at a prompt or in a graphical interface. *salt* is usually a random two-character string which will be used to perturb the DES algorithm in one of 4096 ways. The characters in *salt* must be in the set [./a-zA-Z0-9]. Returns the hashed password as a string, which will be composed of characters from the same alphabet as the salt (the first two characters represent the salt itself).

Since a few *crypt(3)* extensions allow different values, with different sizes in the *salt*, it is recommended to use the full crypted password as salt when checking for a password.

A simple example illustrating typical use:

```
import crypt, getpass, pwd

def login():
    username = raw_input('Python login:')
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd == '*':
            raise NotImplementedError(
                "Sorry, currently no support for shadow passwords")
        cleartext = getpass.getpass()
        return crypt.crypt(cleartext, cryptedpasswd) == cryptedpasswd
    else:
        return 1
```

36.6 dl —Call C functions in shared objects

2.6 版后已移除: The *dl* module has been removed in Python 3. Use the *ctypes* module instead.

The *dl* module defines an interface to the `dlopen()` function, which is the most common interface on Unix platforms for handling dynamically linked libraries. It allows the program to call arbitrary functions in such a library.

警告: The *dl* module bypasses the Python type system and error handling. If used incorrectly it may cause segmentation faults, crashes or other incorrect behaviour.

注解: This module will not work unless `sizeof(int) == sizeof(long) == sizeof(char *)` If this is not the case, *SystemError* will be raised on import.

The *dl* module defines the following function:

`dl.`**open**(*name*[, *mode=RTLD_LAZY*])

Open a shared object file, and return a handle. Mode signifies late binding (*RTLD_LAZY*) or immediate binding (*RTLD_NOW*). Default is *RTLD_LAZY*. Note that some systems do not support *RTLD_NOW*.

Return value is a `dlobject`.

The *dl* module defines the following constants:

`dl.`**RTLD_LAZY**

Useful as an argument to *open()*.

`dl.`**RTLD_NOW**

Useful as an argument to *open()*. Note that on systems which do not support immediate binding, this constant will not appear in the module. For maximum portability, use *hasattr()* to determine if the system supports immediate binding.

The *dl* module defines the following exception:

exception `dl.`**error**

Exception raised when an error has occurred inside the dynamic loading and linking routines.

Example:

```
>>> import dl, time
>>> a=dl.open('/lib/libc.so.6')
>>> a.call('time'), time.time()
(929723914, 929723914.498)
```

This example was tried on a Debian GNU/Linux system, and is a good example of the fact that using this module is usually a bad alternative.

36.6.1 Dl Objects

Dl objects, as returned by *open()* above, have the following methods:

`dl.close()`
Free all resources, except the memory.

`dl.sym(`*name*`)`
Return the pointer for the function named *name*, as a number, if it exists in the referenced shared object, otherwise `None`. This is useful in code like:

```
>>> if a.sym('time'):
...     a.call('time')
... else:
...     time.time()
```

(Note that this function will return a non-zero number, as zero is the *NULL* pointer)

`dl.call(`*name*`[, `*arg1*`[, `*arg2...*`]])`
Call the function named *name* in the referenced shared object. The arguments must be either Python integers, which will be passed as is, Python strings, to which a pointer will be passed, or `None`, which will be passed as *NULL*. Note that strings should only be passed to functions as `const char*`, as Python will not like its string mutated.

There must be at most 10 arguments, and arguments not given will be treated as `None`. The function's return value must be a C `long`, which is a Python integer.

36.7 termios —POSIX 风格的 tty 控制

This module provides an interface to the POSIX calls for tty I/O control. For a complete description of these calls, see *termios(2)* Unix manual page. It is only available for those Unix versions that support POSIX *termios* style tty I/O control configured during installation.

All functions in this module take a file descriptor *fd* as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or a file object, such as `sys.stdin` itself.

这个模块还定义了与此处所提供的函数一起使用的所有必要的常量；这些常量与它们在 C 中的对应常量同名。请参考你的系统文档了解有关如何使用这些终端控制接口的更多信息。

这个模块定义了以下函数：

`termios.tcgetattr(`*fd*`)`
对于文件描述符 *fd* 返回一个包含 tty 属性的列表，形式如下：`[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]`，其中 *cc* 为一个包含 tty 特殊字符的列表（每一项都是长度为 1 的字符串，索引号为 `VMIN` 和 `VTIME` 的项除外，这些字段如有定义则应为整数）。对旗标和速度以及 *cc* 数组中索引的解读必须使用在*termios* 模块中定义的符号常量来完成。

termios.**tcsetattr**(*fd*, *when*, *attributes*)
根据 *attributes* 列表设置文件描述符 *fd* 的 tty 属性，该列表即*tcgetattr()* 所返回的对象。*when* 参数确定何时改变属性: TCSANOW 表示立即改变，TCSADRAIN 表示在传输所有队列输出后再改变，或 TCSAFLUSH 表示在传输所有队列输出并丢失所有队列输入后再改变。

termios.**tcsendbreak**(*fd*, *duration*)
Send a break on file descriptor *fd*. A zero *duration* sends a break for 0.25 –0.5 seconds; a nonzero *duration* has a system dependent meaning.

termios.**tcdrain**(*fd*)
进入等待状态直到写入文件描述符 *fd* 的所有输出都传送完毕。

termios.**tcflush**(*fd*, *queue*)
在文件描述符 *fd* 上丢弃队列数据。*queue* 选择器指定哪个队列: TCIFLUSH 表示输入队列，TCOFLUSH 表示输出队列，或 TCIOFLUSH 表示两个队列同时。

termios.**tcflow**(*fd*, *action*)
在文件描述符 *fd* 上挂起一战恢复输入或输出。*action* 参数可以为 TCOOFF 表示挂起输出，TCOON 表示重启输出，TCIOFF 表示挂起输入，或 TCION 表示重启输入。

参见:

模块*tty* 针对常用终端控制操作的便捷函数。

36.7.1 示例

这个函数可提示输入密码并且关闭回显。请注意其采取的技巧是使用一个单独的*tcgetattr()* 调用和一个 try …finally 语句来确保旧的 tty 属性无论在何种情况下都会被原样保存:

```
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = raw_input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

36.8 tty —终端控制功能

tty 模块定义了将 tty 放入 cbreak 和 raw 模式的函数。

因为它需要*termios* 模块，所以只能在 Unix 上运行。

tty 模块定义了以下函数:

tty.**setraw**(*fd*[, *when*])
将文件描述符 *fd* 的模式更改为 raw 。如果 *when* 被省略，则默认为 termios.TCSAFLUSH ，并传递给*termios.tcsetattr()* 。

tty.**setcbreak**(*fd*[, *when*])
将文件描述符 *fd* 的模式更改为 cbreak 。如果 *when* 被省略，则默认为 termios.TCSAFLUSH ，并传递给*termios.tcsetattr()* 。

参见:

模块*termios* 低级终端控制接口。

36.9 pty —伪终端工具

pty 模块定义了一些处理“伪终端”概念的操作：启动另一个进程并能以程序方式在其控制终端中进行读写。

由于伪终端处理高度依赖于具体平台，因此此功能只有针对 Linux 的代码。(Linux 代码也可在其他平台上工作，但是未经测试。)

pty 模块定义了下列函数:

pty.**fork**()
分叉。将子进程的控制终端连接到一个伪终端。返回值为 (pid, fd)。请注意子进程获得 *pid* 0 而 *fd* 为 *invalid*。父进程返回值为子进程的 *pid* 而 *fd* 为一个连接到子进程的控制终端（并同时连接到子进程的标准输入和输出）的文件描述符。

pty.**openpty**()
打开一个新的伪终端对，如果可能将使用*os.openpty()*，或是针对通用 Unix 系统的模拟代码。返回一个文件描述符对 (master, slave)，分别表示主从两端。

pty.**spawn**(*argv*[, *master_read*[, *stdin_read*]])
Spawn a process, and connect its controlling terminal with the current process’s standard io. This is often used to baffle programs which insist on reading from the controlling terminal.

The functions *master_read* and *stdin_read* should be functions which read from a file descriptor. The defaults try to read 1024 bytes each time they are called.

36.10 fcntl —The fcntl and ioctl system calls

This module performs file control and I/O control on file descriptors. It is an interface to the `fcntl()` and `ioctl()` Unix routines. For a complete description of these calls, see *fcntl(2)* and *ioctl(2)* Unix manual pages.

All functions in this module take a file descriptor *fd* as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or a file object, such as `sys.stdin` itself, which provides a *fileno()* which returns a genuine file descriptor.

这个模块定义了以下函数:

fcntl.**fcntl**(*fd*, *op*[, *arg*])
Perform the operation *op* on file descriptor *fd* (file objects providing a *fileno()* method are accepted as well). The values used for for *op* are operating system dependent, and are available as constants in the *fcntl* module, using the same names as used in the relevant C header files. The argument *arg* is optional, and defaults to the integer value `0`. When present, it can either be an integer value, or a string. With the argument missing or an integer value, the return value of this function is the integer return value of the C `fcntl()` call. When the argument is a string it represents a binary structure, e.g. created by *struct.pack()*. The binary data is copied to a buffer whose address is passed to the C `fcntl()` call. The return value after a successful call is the contents of the buffer, converted to a string object. The length of the returned string will be the same as the length of the *arg* argument. This is limited to 1024 bytes. If the information returned in the buffer by the operating system is larger than 1024 bytes, this is most likely to result in a segmentation violation or a more subtle data corruption.

If the `fcntl()` fails, an *IOError* is raised.

fcntl.**ioctl**(*fd*, *op*[, *arg*[, *mutate_flag*]])

This function is identical to the *fcntl()* function, except that the operations are typically defined in the library module *termios* and the argument handling is even more complicated.

The op parameter is limited to values that can fit in 32-bits. Additional constants of interest for use as the *op* argument can be found in the *termios* module, under the same names as used in the relevant C header files.

The parameter *arg* can be one of an integer, absent (treated identically to the integer 0), an object supporting the read-only buffer interface (most likely a plain Python string) or an object supporting the read-write buffer interface.

In all but the last case, behaviour is as for the *fcntl()* function.

If a mutable buffer is passed, then the behaviour is determined by the value of the *mutate_flag* parameter.

If it is false, the buffer's mutability is ignored and behaviour is as for a read-only buffer, except that the 1024 byte limit mentioned above is avoided –so long as the buffer you pass is as least as long as what the operating system wants to put there, things should work.

If *mutate_flag* is true, then the buffer is (in effect) passed to the underlying *ioctl()* system call, the latter's return code is passed back to the calling Python, and the buffer's new contents reflect the action of the *ioctl()*. This is a slight simplification, because if the supplied buffer is less than 1024 bytes long it is first copied into a static buffer 1024 bytes long which is then passed to *ioctl()* and copied back into the supplied buffer.

If *mutate_flag* is not supplied, then from Python 2.5 it defaults to true, which is a change from versions 2.3 and 2.4. Supply the argument explicitly if version portability is a priority.

If the ioctl() fails, an *IOError* exception is raised.

举个例子:

```
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, "  "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
```

fcntl.**flock**(*fd*, *op*)

Perform the lock operation *op* on file descriptor *fd* (file objects providing a *fileno()* method are accepted as well). See the Unix manual *flock(2)* for details. (On some systems, this function is emulated using fcntl().)

If the flock() fails, an *IOError* exception is raised.

fcntl.**lockf**(*fd*, *operation*[, *length*[, *start*[, *whence*]]])

This is essentially a wrapper around the *fcntl()* locking calls. *fd* is the file descriptor of the file to lock or unlock, and *operation* is one of the following values:

- LOCK_UN –unlock
- LOCK_SH –acquire a shared lock
- LOCK_EX –acquire an exclusive lock

When *operation* is LOCK_SH or LOCK_EX, it can also be bitwise ORed with LOCK_NB to avoid blocking on lock acquisition. If LOCK_NB is used and the lock cannot be acquired, an *IOError* will be raised and the exception will have an *errno* attribute set to EACCES or EAGAIN (depending on the operating system; for portability, check for both values). On at least some systems, LOCK_EX can only be used if the file descriptor refers to a file opened for writing.

length is the number of bytes to lock, *start* is the byte offset at which the lock starts, relative to *whence*, and *whence* is as with `io.IOBase.seek()`, specifically:

- `0` –relative to the start of the file (`os.SEEK_SET`)
- `1` –relative to the current buffer position (`os.SEEK_CUR`)
- `2` –relative to the end of the file (`os.SEEK_END`)

The default for *start* is 0, which means to start at the beginning of the file. The default for *length* is 0 which means to lock to the end of the file. The default for *whence* is also 0.

Examples (all on a SVR4 compliant system):

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

Note that in the first example the return value variable *rv* will hold an integer value; in the second example it will hold a string value. The structure lay-out for the *lockdata* variable is system dependent —therefore using the `flock()` call may be better.

参见:

模块 os If the locking flags `O_SHLOCK` and `O_EXLOCK` are present in the `os` module (on BSD only), the `os.open()` function provides an alternative to the `lockf()` and `flock()` functions.

36.11 pipes —终端管道接口

源代码: Lib/pipes.py

`pipes` 定义了一个类用来抽象 *pipeline* 的概念—将数据从一个文件转到另一文件的转换器序列。

由于模块使用了 **/bin/sh** 命令行，因此要求有 POSIX 或兼容`os.system()` 和`os.popen()` 的终端程序。

class pipes.**Template**
对 pipeline 的抽象。

示例:

```
>>> import pipes
>>> t = pipes.Template()
>>> t.append('tr a-z A-Z', '--')
>>> f = t.open('pipefile', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('pipefile').read()
'HELLO WORLD'
```

pipes.**quote**(*s*)
2.7 版后已移除: Prior to Python 2.7, this function was not publicly documented. It is finally exposed publicly in Python 3.3 as the `quote` function in the `shlex` module.

Return a shell-escaped version of the string *s*. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

This idiom would be unsafe:

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print command  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

quote() lets you plug the security hole:

```
>>> command = 'ls -l {}'.format(quote(filename))
>>> print command
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print remote_command
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"''
```

The quoting is compatible with UNIX shells and with *shlex.split()*:

```
>>> remote_command = shlex.split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = shlex.split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

36.11.1 模板对象

模板对象有以下方法:

`Template.`**`reset`**`()`
将一个管道模板恢复为初始状态。

`Template.`**`clone`**`()`
返回一个新的等价的管道模板。

`Template.`**`debug`**`(`*flag*`)`
如果 *flag* 为真值，则启用调试。否则禁用调试。当启用调试时，要执行的命令会被打印出来，并且会给予终端 `set -x` 命令以输出更详细的信息。

`Template.`**`append`**`(`*cmd*, *kind*`)`
在末尾添加一个新的动作。*cmd* 变量必须为一个有效的 bourne 终端命令。*kind* 变量由两个字母组成。

第一个字母可以为 `'-'` (这表示命令将读取其标准输入), `'f'` (这表示命令将读取在命令行中给定的文件) 或 `'.'` (这表示命令将不读取输入，因而必须放在前面。)

类似地，第二个字母可以为 `'-'` (这表示命令将写入到标准输出), `'f'` (这表示命令将写入在命令行中给定的文件) 或 `'.'` (这表示命令将不执行写入，因而必须放在末尾。)

`Template.`**`prepend`**`(`*cmd*, *kind*`)`
在开头添加一个新的动作。请参阅*append()* 获取相应参数的说明。

`Template.`**`open`**`(`*file*, *mode*`)`
返回一个文件类对象，打开到 *file*，但是将从管道读取或写入。请注意只能给出 `'r'`, `'w'` 中的一个。

`Template.`**`copy`**`(`*infile*, *outfile*`)`
通过管道将 *infile* 拷贝到 *outfile*。

36.12 posixfile —File-like objects with locking support

1.5 版后已移除: The locking operation that this module provides is done better and more portably by the *fcntl.lockf()* call.

This module implements some additional functionality over the built-in file objects. In particular, it implements file locking, control over the file flags, and an easy interface to duplicate the file object. The module defines a new file object, the posixfile object. It has all the standard file object methods and adds the methods described below. This module only works for certain flavors of Unix, since it uses *fcntl.fcntl()* for file locking.

To instantiate a posixfile object, use the *posixfile.open()* function. The resulting object looks and feels roughly the same as a standard file object.

The *posixfile* module defines the following constants:

posixfile.**SEEK_SET**
Offset is calculated from the start of the file.

posixfile.**SEEK_CUR**
Offset is calculated from the current position in the file.

posixfile.**SEEK_END**
Offset is calculated from the end of the file.

The *posixfile* module defines the following functions:

posixfile.**open**(*filename*[, *mode*[, *bufsize*]])
Create a new posixfile object with the given filename and mode. The *filename*, *mode* and *bufsize* arguments are interpreted the same way as by the built-in *open()* function.

posixfile.**fileopen**(*fileobject*)
Create a new posixfile object with the given standard file object. The resulting object has the same filename and mode as the original file object.

The posixfile object defines the following additional methods:

posixfile.**lock**(*fmt*[, *len*[, *start*[, *whence*]]])
Lock the specified section of the file that the file object is referring to. The format is explained below in a table. The *len* argument specifies the length of the section that should be locked. The default is `0`. *start* specifies the starting offset of the section, where the default is `0`. The *whence* argument specifies where the offset is relative to. It accepts one of the constants *SEEK_SET*, *SEEK_CUR* or *SEEK_END*. The default is *SEEK_SET*. For more information about the arguments refer to the *fcntl(2)* manual page on your system.

posixfile.**flags**([*flags*])
Set the specified flags for the file that the file object is referring to. The new flags are ORed with the old flags, unless specified otherwise. The format is explained below in a table. Without the *flags* argument a string indicating the current flags is returned (this is the same as the `?` modifier). For more information about the flags refer to the *fcntl(2)* manual page on your system.

posixfile.**dup**()
Duplicate the file object and the underlying file pointer and file descriptor. The resulting object behaves as if it were newly opened.

posixfile.**dup2**(*fd*)
Duplicate the file object and the underlying file pointer and file descriptor. The new object will have the given file descriptor. Otherwise the resulting object behaves as if it were newly opened.

posixfile.**file**()
Return the standard file object that the posixfile object is based on. This is sometimes necessary for functions that insist on a standard file object.

All methods raise *IOError* when the request fails.

Format characters for the `lock()` method have the following meaning:

Format	Meaning
`u`	unlock the specified region
`r`	request a read lock for the specified section
`w`	request a write lock for the specified section

In addition the following modifiers can be added to the format:

Modifier	Meaning	Notes
`|`	wait until the lock has been granted	
`?`	return the first lock conflicting with the requested lock, or `None` if there is no conflict.	(1)

Note:

(1) The lock returned is in the format `(mode, len, start, whence, pid)` where *mode* is a character representing the type of lock ('r' or 'w'). This modifier prevents a request from being granted; it is for query purposes only.

Format characters for the `flags()` method have the following meanings:

Format	Meaning
`a`	append only flag
`c`	close on exec flag
`n`	no delay flag (also called non-blocking flag)
`s`	synchronization flag

In addition the following modifiers can be added to the format:

Modifier	Meaning	Notes
`!`	turn the specified flags 'off', instead of the default 'on'	(1)
`=`	replace the flags, instead of the default 'OR' operation	(1)
`?`	return a string in which the characters represent the flags that are set.	(2)

Notes:

(1) The `!` and `=` modifiers are mutually exclusive.

(2) This string represents the flags after they may have been altered by the same call.

Examples:

```
import posixfile

file = posixfile.open('testfile', 'w')
file.lock('w|')
...
file.lock('u')
file.close()
```

36.13 resource —Resource usage information

This module provides basic mechanisms for measuring and controlling system resources utilized by a program.

Symbolic constants are used to specify particular system resources and to request usage information about either the current process or its children.

A single exception is defined for errors:

exception resource.**error**
The functions described below may raise this error if the underlying system call failures unexpectedly.

36.13.1 Resource Limits

Resources usage can be limited using the *setrlimit()* function described below. Each resource is controlled by a pair of limits: a soft limit and a hard limit. The soft limit is the current limit, and may be lowered or raised by a process over time. The soft limit can never exceed the hard limit. The hard limit can be lowered to any value greater than the soft limit, but not raised. (Only processes with the effective UID of the super-user can raise a hard limit.)

The specific resources that can be limited are system dependent. They are described in the *getrlimit(2)* man page. The resources listed below are supported when the underlying operating system supports them; resources which cannot be checked or controlled by the operating system are not defined in this module for those platforms.

resource.**RLIM_INFINITY**
Constant used to represent the limit for an unlimited resource.

resource.**getrlimit**(*resource*)
Returns a tuple `(soft, hard)` with the current soft and hard limits of *resource*. Raises *ValueError* if an invalid resource is specified, or *error* if the underlying system call fails unexpectedly.

resource.**setrlimit**(*resource*, *limits*)
Sets new limits of consumption of *resource*. The *limits* argument must be a tuple `(soft, hard)` of two integers describing the new limits. A value of *RLIM_INFINITY* can be used to request a limit that is unlimited.

Raises *ValueError* if an invalid resource is specified, if the new soft limit exceeds the hard limit, or if a process tries to raise its hard limit. Specifying a limit of *RLIM_INFINITY* when the hard or system limit for that resource is not unlimited will result in a *ValueError*. A process with the effective UID of super-user can request any valid limit value, including unlimited, but *ValueError* will still be raised if the requested limit exceeds the system imposed limit.

`setrlimit` may also raise *error* if the underlying system call fails.

These symbols define resources whose consumption can be controlled using the *setrlimit()* and *getrlimit()* functions described below. The values of these symbols are exactly the constants used by C programs.

The Unix man page for *getrlimit(2)* lists the available resources. Note that not all systems use the same symbol or same value to denote the same resource. This module does not attempt to mask platform differences —symbols not defined for a platform will not be available from this module on that platform.

resource.**RLIMIT_CORE**
The maximum size (in bytes) of a core file that the current process can create. This may result in the creation of a partial core file if a larger core would be required to contain the entire process image.

resource.**RLIMIT_CPU**
The maximum amount of processor time (in seconds) that a process can use. If this limit is exceeded, a `SIGXCPU` signal is sent to the process. (See the *signal* module documentation for information about how to catch this signal and do something useful, e.g. flush open files to disk.)

resource.**RLIMIT_FSIZE**
The maximum size of a file which the process may create.

resource.**RLIMIT_DATA**
The maximum size (in bytes) of the process's heap.

resource.**RLIMIT_STACK**
The maximum size (in bytes) of the call stack for the current process. This only affects the stack of the main thread in a multi-threaded process.

resource.**RLIMIT_RSS**
The maximum resident set size that should be made available to the process.

resource.**RLIMIT_NPROC**
The maximum number of processes the current process may create.

resource.**RLIMIT_NOFILE**
The maximum number of open file descriptors for the current process.

resource.**RLIMIT_OFILE**
The BSD name for *RLIMIT_NOFILE*.

resource.**RLIMIT_MEMLOCK**
The maximum address space which may be locked in memory.

resource.**RLIMIT_VMEM**
The largest area of mapped memory which the process may occupy.

resource.**RLIMIT_AS**
The maximum area (in bytes) of address space which may be taken by the process.

36.13.2 Resource Usage

These functions are used to retrieve resource usage information:

resource.**getrusage**(*who*)
This function returns an object that describes the resources consumed by either the current process or its children, as specified by the *who* parameter. The *who* parameter should be specified using one of the RUSAGE_* constants described below.

The fields of the return value each describe how a particular system resource has been used, e.g. amount of time spent running is user mode or number of times the process was swapped out of main memory. Some values are dependent on the clock tick internal, e.g. the amount of memory the process is using.

For backward compatibility, the return value is also accessible as a tuple of 16 elements.

The fields ru_utime and ru_stime of the return value are floating point values representing the amount of time spent executing in user mode and the amount of time spent executing in system mode, respectively. The remaining values are integers. Consult the *getrusage(2)* man page for detailed information about these values. A brief summary is presented here:

索引	域	Resource
0	ru_utime	time in user mode (float)
1	ru_stime	time in system mode (float)
2	ru_maxrss	maximum resident set size
3	ru_ixrss	shared memory size
4	ru_idrss	unshared memory size
5	ru_isrss	unshared stack size
6	ru_minflt	page faults not requiring I/O
7	ru_majflt	page faults requiring I/O
8	ru_nswap	number of swap outs
9	ru_inblock	block input operations
10	ru_oublock	block output operations
11	ru_msgsnd	messages sent
12	ru_msgrcv	messages received
13	ru_nsignals	signals received
14	ru_nvcsw	voluntary context switches
15	ru_nivcsw	involuntary context switches

This function will raise a *ValueError* if an invalid *who* parameter is specified. It may also raise *error* exception in unusual circumstances.

在 2.3 版更改: Added access to values as attributes of the returned object.

resource.**getpagesize**()
Returns the number of bytes in a system page. (This need not be the same as the hardware page size.)

The following RUSAGE_* symbols are passed to the *getrusage()* function to specify which processes information should be provided for.

resource.**RUSAGE_SELF**
RUSAGE_SELF should be used to request information pertaining only to the process itself.

resource.**RUSAGE_CHILDREN**
Pass to *getrusage()* to request resource information for child processes of the calling process.

resource.**RUSAGE_BOTH**
Pass to *getrusage()* to request resources consumed by both the current process and child processes. May not be available on all systems.

36.14 nis —Sun 的 NIS (黄页) 接口

nis 模块提供了对 NIS 库的轻量级包装，适用于多个主机的集中管理。

因为 NIS 仅存在于 Unix 系统，此模块仅在 Unix 上可用。

nis 模块定义了以下函数:

nis.**match**(*key*, *mapname*[, *domain=default_domain*])
返回 *key* 在映射 *mapname* 中的匹配结果，如无结果则会引发错误 (*nis.error*)。两个参数都应为字符串，*key* 定长 8 个比特。返回值为任意字节数组（可包含 NULL 和其他特殊值）。

请注意如果 *mapname* 是另一名称的别名则会先检查别名。

在 2.5 版更改: *domain* 参数可允许重载用于查找的 NIS 域。如果未指定，则会在默认 NIS 域中查找。

nis.**cat**(*mapname*[, *domain=default_domain*])
返回一个字典，其元素为 *key* 到 *value* 的映射使得 `match(key, mapname)==value`。请注意字典的键和值均为任意字节数组。

请注意如果 *mapname* 是另一名称的别名则会先检查别名。

在 2.5 版更改: *domain* 参数可允许重载用于查找的 NIS 域。如果未指定，则会在默认 NIS 域中查找。

nis.**maps**([*domain=default_domain*])
返回全部可用映射的列表。

在 2.5 版更改: *domain* 参数可允许重载用于查找的 NIS 域。如果未指定，则会在默认 NIS 域中查找。

nis.**get_default_domain**()
返回系统默认的 NIS 域。

2.5 新版功能.

nis 模块定义了以下异常:

exception nis.**error**
当 NIS 函数返回一个错误码时引发的异常。

36.15 Unix syslog 库例程

This module provides an interface to the Unix `syslog` library routines. Refer to the Unix manual pages for a detailed description of the `syslog` facility.

This module wraps the system `syslog` family of routines. A pure Python library that can speak to a syslog server is available in the *logging.handlers* module as `SysLogHandler`.

这个模块定义了以下函数:

syslog.**syslog**(*message*)
syslog.**syslog**(*priority*, *message*)
Send the string *message* to the system logger. A trailing newline is added if necessary. Each message is tagged with a priority composed of a *facility* and a *level*. The optional *priority* argument, which defaults to `LOG_INFO`, determines the message priority. If the facility is not encoded in *priority* using logical-or (`LOG_INFO | LOG_USER`), the value given in the *openlog()* call is used.

If *openlog()* has not been called prior to the call to *syslog()*, `openlog()` will be called with no arguments.

syslog.**openlog**([*ident*[, *logoption*[, *facility*]]])
Logging options of subsequent *syslog()* calls can be set by calling *openlog()*. *syslog()* will call *openlog()* with no arguments if the log is not currently open.

The optional *ident* keyword argument is a string which is prepended to every message, and defaults to `sys.argv[0]` with leading path components stripped. The optional *logoption* keyword argument (default is 0) is a bit field –see below for possible values to combine. The optional *facility* keyword argument (default is `LOG_USER`) sets the default facility for messages which do not have a facility explicitly encoded.

syslog.**closelog**()
Reset the syslog module values and call the system library `closelog()`.

This causes the module to behave as it does when initially imported. For example, *openlog()* will be called on the first *syslog()* call (if *openlog()* hasn't already been called), and *ident* and other *openlog()* parameters are reset to defaults.

syslog.**setlogmask**(*maskpri*)
Set the priority mask to *maskpri* and return the previous mask value. Calls to *syslog()* with a priority level not set in *maskpri* are ignored. The default is to log all priorities. The function `LOG_MASK(pri)` calculates the

mask for the individual priority *pri*. The function `LOG_UPTO(pri)` calculates the mask for all priorities up to and including *pri*.

The module defines the following constants:

Priority levels (high to low): `LOG_EMERG`, `LOG_ALERT`, `LOG_CRIT`, `LOG_ERR`, `LOG_WARNING`, `LOG_NOTICE`, `LOG_INFO`, `LOG_DEBUG`.

Facilities: `LOG_KERN`, `LOG_USER`, `LOG_MAIL`, `LOG_DAEMON`, `LOG_AUTH`, `LOG_LPR`, `LOG_NEWS`, `LOG_UUCP`, `LOG_CRON`, `LOG_SYSLOG` and `LOG_LOCAL0` to `LOG_LOCAL7`.

Log options: `LOG_PID`, `LOG_CONS`, `LOG_NDELAY`, `LOG_NOWAIT` and `LOG_PERROR` if defined in `<syslog.h>`.

36.15.1 例子

Simple example

A simple set of examples:

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

An example of setting some log options, these would include the process ID in logged messages, and write the messages to the destination facility used for mail logging:

```
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

36.16 `commands` —Utilities for running commands

2.6 版后已移除: The *commands* module has been removed in Python 3. Use the *subprocess* module instead.

The *commands* module contains wrapper functions for *os.popen()* which take a system command as a string and return any output generated by the command and, optionally, the exit status.

The *subprocess* module provides more powerful facilities for spawning new processes and retrieving their results. Using the *subprocess* module is preferable to using the *commands* module.

注解: In Python 3.x, *getstatus()* and two undocumented functions (`mk2arg()` and `mkarg()`) have been removed. Also, *getstatusoutput()* and *getoutput()* have been moved to the *subprocess* module.

The *commands* module defines the following functions:

`commands.`**`getstatusoutput`**`(`*cmd*`)`

Execute the string *cmd* in a shell with *os.popen()* and return a 2-tuple `(status, output)`. *cmd* is actually run as `{ cmd ; } 2>&1`, so that the returned output will contain output or error messages. A trailing newline is stripped from the output. The exit status for the command can be interpreted according to the rules for the C function `wait()`.

commands.**getoutput**(*cmd*)
Like *getstatusoutput()*, except the exit status is ignored and the return value is a string containing the command's output.

commands.**getstatus**(*file*)
Return the output of `ls -ld file` as a string. This function uses the *getoutput()* function, and properly escapes backslashes and dollar signs in the argument.

2.6 版后已移除: This function is nonobvious and useless. The name is also misleading in the presence of *getstatusoutput()*.

Example:

```
>>> import commands
>>> commands.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> commands.getstatusoutput('cat /bin/junk')
(256, 'cat: /bin/junk: No such file or directory')
>>> commands.getstatusoutput('/bin/junk')
(256, 'sh: /bin/junk: not found')
>>> commands.getoutput('ls /bin/ls')
'/bin/ls'
>>> commands.getstatus('/bin/ls')
'-rwxr-xr-x  1 root        13352 Oct 14  1994 /bin/ls'
```

参见:

Module ***subprocess*** Module for spawning and managing subprocesses.

CHAPTER 37

Mac OS X specific services

This chapter describes modules that are only available on the Mac OS X platform.

See the chapters *MacPython OSA Modules* and *Undocumented Mac OS modules* for more modules, and the HOWTO using-on-mac for a general introduction to Mac-specific Python programming.

注解： Most of the OS X APIs that these modules use are deprecated or removed in recent versions of OS X. Many are not available when Python is executing in 64-bit mode. These modules have been removed in Python 3. You should avoid using them in Python 2.

37.1 `ic` —Access to the Mac OS X Internet Config

This module provides access to various internet-related preferences set through **`System Preferences`** or the **`Finder`**.

注解： This module has been removed in Python 3.x.

There is a low-level companion module `icglue` which provides the basic Internet Config access functionality. This low-level module is not documented, but the docstrings of the routines document the parameters and the routine names are the same as for the Pascal or C API to Internet Config, so the standard IC programmers' documentation can be used if this module is needed.

The *`ic`* module defines the *`error`* exception and symbolic names for all error codes Internet Config can produce; see the source for details.

exception `ic.error`
Exception raised on errors in the *`ic`* module.

The *`ic`* module defines the following class and function:

class ic.**IC**([*signature*[, *ic*]])
Create an Internet Config object. The signature is a 4-character creator code of the current application (default `'Pyth'`) which may influence some of ICs settings. The optional *ic* argument is a low-level `icglue.icinstance` created beforehand, this may be useful if you want to get preferences from a different config file, etc.

ic.**launchurl**(*url*[, *hint*])
ic.**parseurl**(*data*[, *start*[, *end*[, *hint*]]])
ic.**mapfile**(*file*)
ic.**maptypecreator**(*type*, *creator*[, *filename*])
ic.**settypecreator**(*file*)
These functions are "shortcuts" to the methods of the same name, described below.

37.1.1 IC Objects

IC objects have a mapping interface, hence to obtain the mail address you simply get `ic['MailAddress']`. Assignment also works, and changes the option in the configuration file.

The module knows about various datatypes, and converts the internal IC representation to a "logical" Python data structure. Running the *ic* module standalone will run a test program that lists all keys and values in your IC database, this will have to serve as documentation.

If the module does not know how to represent the data it returns an instance of the `ICOpaqueData` type, with the raw data in its `data` attribute. Objects of this type are also acceptable values for assignment.

Besides the dictionary interface, *IC* objects have the following methods:

IC.**launchurl**(*url*[, *hint*])
Parse the given URL, launch the correct application and pass it the URL. The optional *hint* can be a scheme name such as `'mailto:'`, in which case incomplete URLs are completed with this scheme. If *hint* is not provided, incomplete URLs are invalid.

IC.**parseurl**(*data*[, *start*[, *end*[, *hint*]]])
Find a URL somewhere in *data* and return start position, end position and the URL. The optional *start* and *end* can be used to limit the search, so for instance if a user clicks in a long text field you can pass the whole text field and the click-position in *start* and this routine will return the whole URL in which the user clicked. As above, *hint* is an optional scheme used to complete incomplete URLs.

IC.**mapfile**(*file*)
Return the mapping entry for the given *file*, which can be passed as either a filename or an `FSSpec()` result, and which need not exist.

The mapping entry is returned as a tuple `(version, type, creator, postcreator, flags, extension, appname, postappname, mimetype, entryname)`, where *version* is the entry version number, *type* is the 4-character filetype, *creator* is the 4-character creator type, *postcreator* is the 4-character creator code of an optional application to post-process the file after downloading, *flags* are various bits specifying whether to transfer in binary or ascii and such, *extension* is the filename extension for this file type, *appname* is the printable name of the application to which this file belongs, *postappname* is the name of the postprocessing application, *mimetype* is the MIME type of this file and *entryname* is the name of this entry.

IC.**maptypecreator**(*type*, *creator*[, *filename*])
Return the mapping entry for files with given 4-character *type* and *creator* codes. The optional *filename* may be specified to further help finding the correct entry (if the creator code is `'????'`, for instance).

The mapping entry is returned in the same format as for *mapfile*.

IC.**settypecreator**(*file*)
Given an existing *file*, specified either as a filename or as an `FSSpec()` result, set its creator and type correctly

based on its extension. The finder is told about the change, so the finder icon will be updated quickly.

37.2 MacOS —Access to Mac OS interpreter features

This module provides access to MacOS specific functionality in the Python interpreter, such as how the interpreter event-loop functions and the like. Use with care.

注解: This module has been removed in Python 3.x.

Note the capitalization of the module name; this is a historical artifact.

MacOS.**runtimemodel**
Always `'macho'`, from Python 2.4 on. In earlier versions of Python the value could also be `'ppc'` for the classic Mac OS 8 runtime model or `'carbon'` for the Mac OS 9 runtime model.

MacOS.**linkmodel**
The way the interpreter has been linked. As extension modules may be incompatible between linking models, packages could use this information to give more decent error messages. The value is one of `'static'` for a statically linked Python, `'framework'` for Python in a Mac OS X framework, `'shared'` for Python in a standard Unix shared library. Older Pythons could also have the value `'cfm'` for Mac OS 9-compatible Python.

exception MacOS.**Error**
This exception is raised on MacOS generated errors, either from functions in this module or from other mac-specific modules like the toolbox interfaces. The arguments are the integer error code (the `OSErr` value) and a textual description of the error code. Symbolic names for all known error codes are defined in the standard module *macerrors*.

MacOS.**GetErrorString**(*errno*)
Return the textual description of MacOS error code *errno*.

MacOS.**DebugStr**(*message*[, *object*])
On Mac OS X the string is simply printed to stderr (on older Mac OS systems more elaborate functionality was available), but it provides a convenient location to attach a breakpoint in a low-level debugger like **gdb**.

注解: Not available in 64-bit mode.

MacOS.**SysBeep**()
Ring the bell.

注解: Not available in 64-bit mode.

MacOS.**GetTicks**()
Get the number of clock ticks (1/60th of a second) since system boot.

MacOS.**GetCreatorAndType**(*file*)
Return the file creator and file type as two four-character strings. The *file* parameter can be a pathname or an `FSSpec` or `FSRef` object.

注解: It is not possible to use an `FSSpec` in 64-bit mode.

MacOS.**SetCreatorAndType**(*file*, *creator*, *type*)
Set the file creator and file type. The *file* parameter can be a pathname or an FSSpec or FSRef object. *creator* and *type* must be four character strings.

注解: It is not possible to use an FSSpec in 64-bit mode.

MacOS.**openrf**(*name*[, *mode*])
Open the resource fork of a file. Arguments are the same as for the built-in function *open()*. The object returned has file-like semantics, but it is not a Python file object, so there may be subtle differences.

MacOS.**WMAvailable**()
Checks whether the current process has access to the window manager. The method will return False if the window manager is not available, for instance when running on Mac OS X Server or when logged in via ssh, or when the current interpreter is not running from a fullblown application bundle. A script runs from an application bundle either when it has been started with **pythonw** instead of **python** or when running as an applet.

MacOS.**splash**([*resourceid*])
Opens a splash screen by resource id. Use resourceid 0 to close the splash screen.

注解: Not available in 64-bit mode.

37.3 macostools —Convenience routines for file manipulation

This module contains some convenience routines for file-manipulation on the Macintosh. All file parameters can be specified as pathnames, FSRef or FSSpec objects. This module expects a filesystem which supports forked files, so it should not be used on UFS partitions.

注解: This module has been removed in Python 3.

The *macostools* module defines the following functions:

macostools.**copy**(*src*, *dst*[, *createpath*[, *copytimes*]])
Copy file *src* to *dst*. If *createpath* is non-zero the folders leading to *dst* are created if necessary. The method copies data and resource fork and some finder information (creator, type, flags) and optionally the creation, modification and backup times (default is to copy them). Custom icons, comments and icon position are not copied.

注解: This function does not work in 64-bit code because it uses APIs that are not available in 64-bit mode.

macostools.**copytree**(*src*, *dst*)
Recursively copy a file tree from *src* to *dst*, creating folders as needed. *src* and *dst* should be specified as pathnames.

注解: This function does not work in 64-bit code because it uses APIs that are not available in 64-bit mode.

macostools.**mkalias**(*src*, *dst*)
Create a finder alias *dst* pointing to *src*.

注解: This function does not work in 64-bit code because it uses APIs that are not available in 64-bit mode.

macostools.**touched**(*dst*)
Tell the finder that some bits of finder-information such as creator or type for file *dst* has changed. The file can be specified by pathname or fsspec. This call should tell the finder to redraw the files icon.

2.6 版后已移除: The function is a no-op on OS X.

macostools.**BUFSIZ**
The buffer size for `copy`, default 1 megabyte.

Note that the process of creating finder aliases is not specified in the Apple documentation. Hence, aliases created with *mkalias()* could conceivably have incompatible behaviour in some cases.

37.4 findertools —The finder's Apple Events interface

This module contains routines that give Python programs access to some functionality provided by the finder. They are implemented as wrappers around the AppleEvent interface to the finder.

All file and folder parameters can be specified either as full pathnames, or as `FSRef` or `FSSpec` objects.

The *findertools* module defines the following functions:

findertools.**launch**(*file*)
Tell the finder to launch *file*. What launching means depends on the file: applications are started, folders are opened and documents are opened in the correct application.

findertools.**Print**(*file*)
Tell the finder to print a file. The behaviour is identical to selecting the file and using the print command in the finder's file menu.

findertools.**copy**(*file*, *destdir*)
Tell the finder to copy a file or folder *file* to folder *destdir*. The function returns an `Alias` object pointing to the new file.

findertools.**move**(*file*, *destdir*)
Tell the finder to move a file or folder *file* to folder *destdir*. The function returns an `Alias` object pointing to the new file.

findertools.**sleep**()
Tell the finder to put the Macintosh to sleep, if your machine supports it.

findertools.**restart**()
Tell the finder to perform an orderly restart of the machine.

findertools.**shutdown**()
Tell the finder to perform an orderly shutdown of the machine.

37.5 EasyDialogs —Basic Macintosh dialogs

The *EasyDialogs* module contains some simple dialogs for the Macintosh. The dialogs get launched in a separate application which appears in the dock and must be clicked on for the dialogs be displayed. All routines take an optional resource ID parameter *id* with which one can override the `DLOG` resource used for the dialog, provided that the dialog items correspond (both type and item number) to those in the default `DLOG` resource. See source code for details.

注解: This module has been removed in Python 3.x.

The *EasyDialogs* module defines the following functions:

EasyDialogs.**Message**(*str*[, *id*[, *ok*]])
Displays a modal dialog with the message text *str*, which should be at most 255 characters long. The button text defaults to "OK", but is set to the string argument *ok* if the latter is supplied. Control is returned when the user clicks the "OK" button.

EasyDialogs.**AskString**(*prompt*[, *default*[, *id*[, *ok*[, *cancel*]]]])
Asks the user to input a string value via a modal dialog. *prompt* is the prompt message, and the optional *default* supplies the initial value for the string (otherwise `""` is used). The text of the "OK" and "Cancel" buttons can be changed with the *ok* and *cancel* arguments. All strings can be at most 255 bytes long. *AskString()* returns the string entered or *None* in case the user cancelled.

EasyDialogs.**AskPassword**(*prompt*[, *default*[, *id*[, *ok*[, *cancel*]]]])
Asks the user to input a string value via a modal dialog. Like *AskString()*, but with the text shown as bullets. The arguments have the same meaning as for *AskString()*.

EasyDialogs.**AskYesNoCancel**(*question*[, *default*[, *yes*[, *no*[, *cancel*[, *id*]]]]])
Presents a dialog with prompt *question* and three buttons labelled "Yes", "No", and "Cancel". Returns `1` for "Yes", `0` for "No" and `-1` for "Cancel". The value of *default* (or `0` if *default* is not supplied) is returned when the `RETURN` key is pressed. The text of the buttons can be changed with the *yes*, *no*, and *cancel* arguments; to prevent a button from appearing, supply `""` for the corresponding argument.

EasyDialogs.**ProgressBar**([*title*[, *maxval*[, *label*[, *id*]]]])
Displays a modeless progress-bar dialog. This is the constructor for the *ProgressBar* class described below. *title* is the text string displayed (default "Working…"), *maxval* is the value at which progress is complete (default `0`, indicating that an indeterminate amount of work remains to be done), and *label* is the text that is displayed above the progress bar itself.

EasyDialogs.**GetArgv**(*[optionlist[commandlist[, addoldfile[, addnewfile[, addfolder[, id]]]]]]*)
Displays a dialog which aids the user in constructing a command-line argument list. Returns the list in `sys.argv` format, suitable for passing as an argument to *getopt.getopt()*. *addoldfile*, *addnewfile*, and *addfolder* are boolean arguments. When nonzero, they enable the user to insert into the command line paths to an existing file, a (possibly) not-yet-existent file, and a folder, respectively. (Note: Option arguments must appear in the command line before file and folder arguments in order to be recognized by *getopt.getopt()*.) Arguments containing spaces can be specified by enclosing them within single or double quotes. A *SystemExit* exception is raised if the user presses the "Cancel" button.

optionlist is a list that determines a popup menu from which the allowed options are selected. Its items can take one of two forms: *optstr* or `(optstr, descr)`. When present, *descr* is a short descriptive string that is displayed in the dialog while this option is selected in the popup menu. The correspondence between *optstr*s and command-line arguments is:

optstr format	Command-line format
`x`	`-x` (short option)
`x:` or `x=`	`-x` (short option with value)
`xyz`	`--xyz` (long option)
`xyz:` or `xyz=`	`--xyz` (long option with value)

commandlist is a list of items of the form *cmdstr* or `(cmdstr, descr)`, where *descr* is as above. The *cmdstr*s will appear in a popup menu. When chosen, the text of *cmdstr* will be appended to the command line as is, except that a trailing `':'` or `'='` (if present) will be trimmed off.

2.0 新版功能.

EasyDialogs.**AskFileForOpen**(*[message] [, typeList] [, defaultLocation] [, defaultOptionFlags] [, location] [, clientName] [, windowTitle] [, actionButtonLabel] [, cancelButtonLabel] [, preferenceKey] [, popupExtension] [, eventProc] [, previewProc] [, filterProc] [, wanted]*)

Post a dialog asking the user for a file to open, and return the file selected or `None` if the user cancelled. *message* is a text message to display, *typeList* is a list of 4-char filetypes allowable, *defaultLocation* is the pathname, `FSSpec` or `FSRef` of the folder to show initially, *location* is the `(x, y)` position on the screen where the dialog is shown, *actionButtonLabel* is a string to show instead of "Open" in the OK button, *cancelButtonLabel* is a string to show instead of "Cancel" in the cancel button, *wanted* is the type of value wanted as a return: `str`, `unicode`, `FSSpec`, `FSRef` and subtypes thereof are acceptable.

For a description of the other arguments please see the Apple Navigation Services documentation and the `EasyDialogs` source code.

EasyDialogs.**AskFileForSave**(*[message] [, savedFileName] [, defaultLocation] [, defaultOptionFlags] [, location] [, clientName] [, windowTitle] [, actionButtonLabel] [, cancelButtonLabel] [, preferenceKey] [, popupExtension] [, fileType] [, fileCreator] [, eventProc] [, wanted]*)

Post a dialog asking the user for a file to save to, and return the file selected or `None` if the user cancelled. *savedFileName* is the default for the file name to save to (the return value). See `AskFileForOpen()` for a description of the other arguments.

EasyDialogs.**AskFolder**(*[message] [, defaultLocation] [, defaultOptionFlags] [, location] [, clientName] [, windowTitle] [, actionButtonLabel] [, cancelButtonLabel] [, preferenceKey] [, popupExtension] [, eventProc] [, filterProc] [, wanted]*)

Post a dialog asking the user to select a folder, and return the folder selected or `None` if the user cancelled. See `AskFileForOpen()` for a description of the arguments.

参见:

Navigation Services Reference Programmer's reference documentation for the Navigation Services, a part of the Carbon framework.

37.5.1 ProgressBar Objects

`ProgressBar` objects provide support for modeless progress-bar dialogs. Both determinate (thermometer style) and indeterminate (barber-pole style) progress bars are supported. The bar will be determinate if its maximum value is greater than zero; otherwise it will be indeterminate.

在 2.2 版更改: Support for indeterminate-style progress bars was added.

The dialog is displayed immediately after creation. If the dialog's "Cancel" button is pressed, or if `Cmd-.` or `ESC` is typed, the dialog window is hidden and `KeyboardInterrupt` is raised (but note that this response does not occur until the progress bar is next updated, typically via a call to `inc()` or `set()`). Otherwise, the bar remains visible until the `ProgressBar` object is discarded.

`ProgressBar` objects possess the following attributes and methods:

ProgressBar.**curval**

The current value (of type integer or long integer) of the progress bar. The normal access methods coerce `curval` between `0` and `maxval`. This attribute should not be altered directly.

ProgressBar.**maxval**

The maximum value (of type integer or long integer) of the progress bar; the progress bar (thermometer style) is full when `curval` equals `maxval`. If `maxval` is `0`, the bar will be indeterminate (barber-pole). This attribute should not be altered directly.

ProgressBar.**title**(*[newstr]*)

Sets the text in the title bar of the progress dialog to *newstr*.

ProgressBar.**label**([*newstr*])
Sets the text in the progress box of the progress dialog to *newstr*.

ProgressBar.**set**(*value*[, *max*])
Sets the progress bar's *curval* to *value*, and also *maxval* to *max* if the latter is provided. *value* is first coerced between 0 and *maxval*. The thermometer bar is updated to reflect the changes, including a change from indeterminate to determinate or vice versa.

ProgressBar.**inc**([*n*])
Increments the progress bar's *curval* by *n*, or by 1 if *n* is not provided. (Note that *n* may be negative, in which case the effect is a decrement.) The progress bar is updated to reflect the change. If the bar is indeterminate, this causes one "spin" of the barber pole. The resulting *curval* is coerced between 0 and *maxval* if incrementing causes it to fall outside this range.

37.6 FrameWork —Interactive application framework

The *FrameWork* module contains classes that together provide a framework for an interactive Macintosh application. The programmer builds an application by creating subclasses that override various methods of the bases classes, thereby implementing the functionality wanted. Overriding functionality can often be done on various different levels, i.e. to handle clicks in a single dialog window in a non-standard way it is not necessary to override the complete event handling.

注解: This module has been removed in Python 3.x.

Work on the *FrameWork* has pretty much stopped, now that PyObjC is available for full Cocoa access from Python, and the documentation describes only the most important functionality, and not in the most logical manner at that. Examine the source or the examples for more details. The following are some comments posted on the MacPython newsgroup about the strengths and limitations of *FrameWork*:

> The strong point of *FrameWork* is that it allows you to break into the control-flow at many different places. *W*, for instance, uses a different way to enable/disable menus and that plugs right in leaving the rest intact. The weak points of *FrameWork* are that it has no abstract command interface (but that shouldn't be difficult), that its dialog support is minimal and that its control/toolbar support is non-existent.

The *FrameWork* module defines the following functions:

FrameWork.**Application**()
An object representing the complete application. See below for a description of the methods. The default __init__() routine creates an empty window dictionary and a menu bar with an apple menu.

FrameWork.**MenuBar**()
An object representing the menubar. This object is usually not created by the user.

FrameWork.**Menu**(*bar*, *title*[, *after*])
An object representing a menu. Upon creation you pass the MenuBar the menu appears in, the *title* string and a position (1-based) *after* where the menu should appear (default: at the end).

FrameWork.**MenuItem**(*menu*, *title*[, *shortcut*, *callback*])
Create a menu item object. The arguments are the menu to create, the item title string and optionally the keyboard shortcut and a callback routine. The callback is called with the arguments menu-id, item number within menu (1-based), current front window and the event record.

Instead of a callable object the callback can also be a string. In this case menu selection causes the lookup of a method in the topmost window and the application. The method name is the callback string with 'domenu_' prepended.

Calling the MenuBar fixmenudimstate() method sets the correct dimming for all menu items based on the current front window.

FrameWork.**Separator**(*menu*)
Add a separator to the end of a menu.

FrameWork.**SubMenu**(*menu*, *label*)
Create a submenu named *label* under menu *menu*. The menu object is returned.

FrameWork.**Window**(*parent*)
Creates a (modeless) window. *Parent* is the application object to which the window belongs. The window is not displayed until later.

FrameWork.**DialogWindow**(*parent*)
Creates a modeless dialog window.

FrameWork.**windowbounds**(*width*, *height*)
Return a (left, top, right, bottom) tuple suitable for creation of a window of given width and height. The window will be staggered with respect to previous windows, and an attempt is made to keep the whole window on-screen. However, the window will however always be the exact size given, so parts may be offscreen.

FrameWork.**setwatchcursor**()
Set the mouse cursor to a watch.

FrameWork.**setarrowcursor**()
Set the mouse cursor to an arrow.

37.6.1 Application Objects

Application objects have the following methods, among others:

Application.**makeusermenus**()
Override this method if you need menus in your application. Append the menus to the attribute menubar.

Application.**getabouttext**()
Override this method to return a text string describing your application. Alternatively, override the do_about() method for more elaborate "about" messages.

Application.**mainloop**([*mask*[, *wait*]])
This routine is the main event loop, call it to set your application rolling. *Mask* is the mask of events you want to handle, *wait* is the number of ticks you want to leave to other concurrent application (default 0, which is probably not a good idea). While raising *self* to exit the mainloop is still supported it is not recommended: call self._quit() instead.

The event loop is split into many small parts, each of which can be overridden. The default methods take care of dispatching events to windows and dialogs, handling drags and resizes, Apple Events, events for non-FrameWork windows, etc.

In general, all event handlers should return 1 if the event is fully handled and 0 otherwise (because the front window was not a FrameWork window, for instance). This is needed so that update events and such can be passed on to other windows like the Sioux console window. Calling MacOS.HandleEvent() is not allowed within *our_dispatch* or its callees, since this may result in an infinite loop if the code is called through the Python inner-loop event handler.

Application.**asyncevents**(*onoff*)
Call this method with a nonzero parameter to enable asynchronous event handling. This will tell the inner interpreter loop to call the application event handler *async_dispatch* whenever events are available. This will cause FrameWork window updates and the user interface to remain working during long computations, but will slow the interpreter down and may cause surprising results in non-reentrant code (such as FrameWork itself). By default *async_dispatch*

will immediately call *our_dispatch* but you may override this to handle only certain events asynchronously. Events you do not handle will be passed to Sioux and such.

The old on/off value is returned.

`Application.`**`_quit`**`()`
Terminate the running *`mainloop()`* call at the next convenient moment.

`Application.`**`do_char`**`(`*c*, *event*`)`
The user typed character *c*. The complete details of the event can be found in the *event* structure. This method can also be provided in a `Window` object, which overrides the application-wide handler if the window is frontmost.

`Application.`**`do_dialogevent`**`(`*event*`)`
Called early in the event loop to handle modeless dialog events. The default method simply dispatches the event to the relevant dialog (not through the `DialogWindow` object involved). Override if you need special handling of dialog events (keyboard shortcuts, etc).

`Application.`**`idle`**`(`*event*`)`
Called by the main event loop when no events are available. The null-event is passed (so you can look at mouse position, etc).

37.6.2 Window Objects

Window objects have the following methods, among others:

`Window.`**`open`**`()`
Override this method to open a window. Store the Mac OS window-id in `self.wid` and call the `do_postopen()` method to register the window with the parent application.

`Window.`**`close`**`()`
Override this method to do any special processing on window close. Call the `do_postclose()` method to cleanup the parent state.

`Window.`**`do_postresize`**`(`*width*, *height*, *macoswindowid*`)`
Called after the window is resized. Override if more needs to be done than calling `InvalRect`.

`Window.`**`do_contentclick`**`(`*local*, *modifiers*, *event*`)`
The user clicked in the content part of a window. The arguments are the coordinates (window-relative), the key modifiers and the raw event.

`Window.`**`do_update`**`(`*macoswindowid*, *event*`)`
An update event for the window was received. Redraw the window.

`Window.`**`do_activate`**`(`*activate*, *event*`)`
The window was activated (`activate == 1`) or deactivated (`activate == 0`). Handle things like focus highlighting, etc.

37.6.3 ControlsWindow Object

ControlsWindow objects have the following methods besides those of `Window` objects:

`ControlsWindow.`**`do_controlhit`**`(`*window*, *control*, *pcode*, *event*`)`
Part *pcode* of control *control* was hit by the user. Tracking and such has already been taken care of.

37.6.4 ScrolledWindow Object

ScrolledWindow objects are ControlsWindow objects with the following extra methods:

`ScrolledWindow.`**`scrollbars`**`(`[*wantx*[, *wanty*]]`)`
Create (or destroy) horizontal and vertical scrollbars. The arguments specify which you want (default: both). The scrollbars always have minimum `0` and maximum `32767`.

`ScrolledWindow.`**`getscrollbarvalues`**`()`
You must supply this method. It should return a tuple `(x, y)` giving the current position of the scrollbars (between `0` and `32767`). You can return `None` for either to indicate the whole document is visible in that direction.

`ScrolledWindow.`**`updatescrollbars`**`()`
Call this method when the document has changed. It will call *`getscrollbarvalues()`* and update the scrollbars.

`ScrolledWindow.`**`scrollbar_callback`**`(`*which*, *what*, *value*`)`
Supplied by you and called after user interaction. *which* will be `'x'` or `'y'`, *what* will be `'-'`, `'--'`, `'set'`, `'++'` or `'+'`. For `'set'`, *value* will contain the new scrollbar position.

`ScrolledWindow.`**`scalebarvalues`**`(`*absmin*, *absmax*, *curmin*, *curmax*`)`
Auxiliary method to help you calculate values to return from *`getscrollbarvalues()`*. You pass document minimum and maximum value and topmost (leftmost) and bottommost (rightmost) visible values and it returns the correct number or `None`.

`ScrolledWindow.`**`do_activate`**`(`*onoff*, *event*`)`
Takes care of dimming/highlighting scrollbars when a window becomes frontmost. If you override this method, call this one at the end of your method.

`ScrolledWindow.`**`do_postresize`**`(`*width*, *height*, *window*`)`
Moves scrollbars to the correct position. Call this method initially if you override it.

`ScrolledWindow.`**`do_controlhit`**`(`*window*, *control*, *pcode*, *event*`)`
Handles scrollbar interaction. If you override it call this method first, a nonzero return value indicates the hit was in the scrollbars and has been handled.

37.6.5 DialogWindow Objects

DialogWindow objects have the following methods besides those of `Window` objects:

`DialogWindow.`**`open`**`(`*resid*`)`
Create the dialog window, from the DLOG resource with id *resid*. The dialog object is stored in `self.wid`.

`DialogWindow.`**`do_itemhit`**`(`*item*, *event*`)`
Item number *item* was hit. You are responsible for redrawing toggle buttons, etc.

37.7 `autoGIL` —Global Interpreter Lock handling in event loops

The *`autoGIL`* module provides a function *`installAutoGIL()`* that automatically locks and unlocks Python's *Global Interpreter Lock* when running an event loop.

注解: This module has been removed in Python 3.x.

exception autoGIL.**AutoGILError**
Raised if the observer callback cannot be installed, for example because the current thread does not have a run loop.

autoGIL.**installAutoGIL**()
Install an observer callback in the event loop (CFRunLoop) for the current thread, that will lock and unlock the Global Interpreter Lock (GIL) at appropriate times, allowing other Python threads to run while the event loop is idle.

Availability: OSX 10.1 or later.

37.8 Mac OS Toolbox Modules

These are a set of modules that provide interfaces to various legacy Mac OS toolboxes. If applicable the module will define a number of Python objects for the various structures declared by the toolbox, and operations will be implemented as methods of the object. Other operations will be implemented as functions in the module. Not all operations possible in C will also be possible in Python (callbacks are often a problem), and parameters will occasionally be different in Python (input and output buffers, especially). All methods and functions have a __doc__ string describing their arguments and return values, and for additional description you are referred to Inside Macintosh or similar works.

These modules all live in a package called Carbon. Despite that name they are not all part of the Carbon framework: CF is really in the CoreFoundation framework and Qt is in the QuickTime framework. The normal use pattern is

```
from Carbon import AE
```

注解: Most of the OS X APIs that these modules use are deprecated or removed in recent versions of OS X. Many are not available when Python is executing in 64-bit mode. The Carbon modules have been removed in Python 3. You should avoid using them in Python 2.

37.8.1 Carbon.AE —Apple Events

37.8.2 Carbon.AH —Apple Help

37.8.3 Carbon.App —Appearance Manager

37.8.4 Carbon.Appearance —Appearance Manager constants

37.8.5 Carbon.CF —Core Foundation

The CFBase, CFArray, CFData, CFDictionary, CFString and CFURL objects are supported, some only partially.

The Scrap Manager supports the simplest form of cut & paste operations on the Macintosh. It can be use for both inter- and intra-application clipboard operations.

The `Scrap` module provides low-level access to the functions of the Scrap Manager. It contains the following functions:

`Carbon.Scrap.`**`InfoScrap`**`()`
Return current information about the scrap. The information is encoded as a tuple containing the fields `(size, handle, count, state, path)`.

Field	Meaning
size	Size of the scrap in bytes.
handle	Resource object representing the scrap.
count	Serial number of the scrap contents.
state	Integer; positive if in memory, `0` if on disk, negative if uninitialized.
path	Filename of the scrap when stored on disk.

参见:

Scrap Manager Apple's documentation for the Scrap Manager gives a lot of useful information about using the Scrap Manager in applications.

37.8.53 `Carbon.Snd` —Sound Manager

37.8.54 `Carbon.Sound` —Sound Manager constants

37.8.55 `Carbon.TE` —TextEdit

37.8.56 `Carbon.TextEdit` —TextEdit constants

37.8.57 `Carbon.Win` —Window Manager

37.8.58 `Carbon.Windows` —Window Manager constants

37.9 `ColorPicker` —Color selection dialog

The *`ColorPicker`* module provides access to the standard color picker dialog.

注解: This module has been removed in Python 3.x.

`ColorPicker.`**`GetColor`**`(`*prompt*, *rgb*`)`
Show a standard color selection dialog and allow the user to select a color. The user is given instruction by the *prompt* string, and the default color is set to *rgb*. *rgb* must be a tuple giving the red, green, and blue components of the color. *`GetColor()`* returns a tuple giving the user's selected color and a flag indicating whether they accepted the selection of cancelled.

CHAPTER 38

MacPython OSA Modules

This chapter describes the current implementation of the Open Scripting Architecture (OSA, also commonly referred to as AppleScript) for Python, allowing you to control scriptable applications from your Python program, and with a fairly pythonic interface. Development on this set of modules has stopped.

For a description of the various components of AppleScript and OSA, and to get an understanding of the architecture and terminology, you should read Apple's documentation. The "Applescript Language Guide" explains the conceptual model and the terminology, and documents the standard suite. The "Open Scripting Architecture" document explains how to use OSA from an application programmers point of view. In the Apple Help Viewer these books are located in the Developer Documentation, Core Technologies section.

As an example of scripting an application, the following piece of AppleScript will get the name of the frontmost **Finder** window and print it:

```
tell application "Finder"
    get name of window 1
end tell
```

In Python, the following code fragment will do the same:

```
import Finder

f = Finder.Finder()
print f.get(f.window(1).name)
```

As distributed the Python library includes packages that implement the standard suites, plus packages that interface to a small number of common applications.

To send AppleEvents to an application you must first create the Python package interfacing to the terminology of the application (what **Script Editor** calls the "Dictionary"). This can be done from within the **PythonIDE** or by running the `gensuitemodule.py` module as a standalone program from the command line.

The generated output is a package with a number of modules, one for every suite used in the program plus an `__init__` module to glue it all together. The Python inheritance graph follows the AppleScript inheritance graph, so if a program's dictionary specifies that it includes support for the Standard Suite, but extends one or two verbs with extra arguments then

the output suite will contain a module `Standard_Suite` that imports and re-exports everything from `StdSuites.Standard_Suite` but overrides the methods that have extra functionality. The output of *gensuitemodule* is pretty readable, and contains the documentation that was in the original AppleScript dictionary in Python docstrings, so reading it is a good source of documentation.

The output package implements a main class with the same name as the package which contains all the AppleScript verbs as methods, with the direct object as the first argument and all optional parameters as keyword arguments. AppleScript classes are also implemented as Python classes, as are comparisons and all the other thingies.

The main Python class implementing the verbs also allows access to the properties and elements declared in the AppleScript class "application". In the current release that is as far as the object orientation goes, so in the example above we need to use `f.get(f.window(1).name)` instead of the more Pythonic `f.window(1).name.get()`.

If an AppleScript identifier is not a Python identifier the name is mangled according to a small number of rules:

- spaces are replaced with underscores
- other non-alphanumeric characters are replaced with `_xx_` where `xx` is the hexadecimal character value
- any Python reserved word gets an underscore appended

Python also has support for creating scriptable applications in Python, but The following modules are relevant to MacPython AppleScript support:

38.1 `gensuitemodule` —Generate OSA stub packages

The *gensuitemodule* module creates a Python package implementing stub code for the AppleScript suites that are implemented by a specific application, according to its AppleScript dictionary.

It is usually invoked by the user through the **PythonIDE**, but it can also be run as a script from the command line (pass `--help` for help on the options) or imported from Python code. For an example of its use see `Mac/scripts/genallsuites.py` in a source distribution, which generates the stub packages that are included in the standard library.

It defines the following public functions:

`gensuitemodule.`**`is_scriptable`**(*application*)

Returns true if `application`, which should be passed as a pathname, appears to be scriptable. Take the return value with a grain of salt: **Internet Explorer** appears not to be scriptable but definitely is.

`gensuitemodule.`**`processfile`**(*application*[, *output*, *basepkgname*, *edit_modnames*, *creatorsignature*, *dump*, *verbose*])

Create a stub package for `application`, which should be passed as a full pathname. For a `.app` bundle this is the pathname to the bundle, not to the executable inside the bundle; for an unbundled CFM application you pass the filename of the application binary.

This function asks the application for its OSA terminology resources, decodes these resources and uses the resultant data to create the Python code for the package implementing the client stubs.

`output` is the pathname where the resulting package is stored, if not specified a standard "save file as" dialog is presented to the user. `basepkgname` is the base package on which this package will build, and defaults to `StdSuites`. Only when generating `StdSuites` itself do you need to specify this. `edit_modnames` is a dictionary that can be used to change modulenames that are too ugly after name mangling. `creator_signature` can be used to override the 4-char creator code, which is normally obtained from the `PkgInfo` file in the package or from the CFM file creator signature. When `dump` is given it should refer to a file object, and `processfile` will stop after decoding the resources and dump the Python representation of the terminology resources to this file. `verbose` should also be a file object, and specifying it will cause `processfile` to tell you what it is doing.

gensuitemodule.**processfile_fromresource**(*application*[, *output*, *basepkgname*, *edit_modnames*, *creatorsignature*, *dump*, *verbose*])

This function does the same as `processfile`, except that it uses a different method to get the terminology resources. It opens `application` as a resource file and reads all `"aete"` and `"aeut"` resources from this file.

38.2 aetools —OSA client support

The *aetools* module contains the basic functionality on which Python AppleScript client support is built. It also imports and re-exports the core functionality of the *aetypes* and *aepack* modules. The stub packages generated by *gensuitemodule* import the relevant portions of *aetools*, so usually you do not need to import it yourself. The exception to this is when you cannot use a generated suite package and need lower-level access to scripting.

The *aetools* module itself uses the AppleEvent support provided by the *Carbon.AE* module. This has one drawback: you need access to the window manager, see section osx-gui-scripts for details. This restriction may be lifted in future releases.

注解: This module has been removed in Python 3.x.

The *aetools* module defines the following functions:

aetools.**packevent**(*ae*, *parameters*, *attributes*)

Stores parameters and attributes in a pre-created `Carbon.AE.AEDesc` object. `parameters` and `attributes` are dictionaries mapping 4-character OSA parameter keys to Python objects. The objects are packed using `aepack.pack()`.

aetools.**unpackevent**(*ae*[, *formodulename*])

Recursively unpacks a `Carbon.AE.AEDesc` event to Python objects. The function returns the parameter dictionary and the attribute dictionary. The `formodulename` argument is used by generated stub packages to control where AppleScript classes are looked up.

aetools.**keysubst**(*arguments*, *keydict*)

Converts a Python keyword argument dictionary `arguments` to the format required by `packevent` by replacing the keys, which are Python identifiers, by the four-character OSA keys according to the mapping specified in `keydict`. Used by the generated suite packages.

aetools.**enumsubst**(*arguments*, *key*, *edict*)

If the `arguments` dictionary contains an entry for `key` convert the value for that entry according to dictionary `edict`. This converts human-readable Python enumeration names to the OSA 4-character codes. Used by the generated suite packages.

The *aetools* module defines the following class:

class aetools.**TalkTo**([*signature=None*, *start=0*, *timeout=0*])

Base class for the proxy used to talk to an application. `signature` overrides the class attribute `_signature` (which is usually set by subclasses) and is the 4-char creator code defining the application to talk to. `start` can be set to true to enable running the application on class instantiation. `timeout` can be specified to change the default timeout used while waiting for an AppleEvent reply.

TalkTo.**_start**()

Test whether the application is running, and attempt to start it if not.

TalkTo.**send**(*code*, *subcode*[, *parameters*, *attributes*])

Create the AppleEvent `Carbon.AE.AEDesc` for the verb with the OSA designation `code, subcode` (which are the usual 4-character strings), pack the `parameters` and `attributes` into it, send it to the target application, wait for the reply, unpack the reply with `unpackevent` and return the reply appleevent, the unpacked return values as a dictionary and the return attributes.

38.3 aepack —Conversion between Python variables and AppleEvent data containers

The *aepack* module defines functions for converting (packing) Python variables to AppleEvent descriptors and back (unpacking). Within Python the AppleEvent descriptor is handled by Python objects of built-in type AEDesc, defined in module *Carbon.AE*.

注解: This module has been removed in Python 3.x.

The *aepack* module defines the following functions:

aepack.**pack**(*x*[, *forcetype*])

Returns an AEDesc object containing a conversion of Python value x. If *forcetype* is provided it specifies the descriptor type of the result. Otherwise, a default mapping of Python types to Apple Event descriptor types is used, as follows:

Python type	descriptor type
FSSpec	typeFSS
FSRef	typeFSRef
Alias	typeAlias
integer	typeLong (32 bit integer)
float	typeFloat (64 bit floating point)
string	typeText
unicode	typeUnicodeText
list	typeAEList
dictionary	typeAERecord
instance	*see below*

If *x* is a Python instance then this function attempts to call an __aepack__() method. This method should return an AEDesc object.

If the conversion *x* is not defined above, this function returns the Python string representation of a value (the repr() function) encoded as a text descriptor.

aepack.**unpack**(*x*[, *formodulename*])

x must be an object of type AEDesc. This function returns a Python object representation of the data in the Apple Event descriptor *x*. Simple AppleEvent data types (integer, text, float) are returned as their obvious Python counterparts. Apple Event lists are returned as Python lists, and the list elements are recursively unpacked. Object references (ex. line 3 of document 1) are returned as instances of *aetypes.ObjectSpecifier*, unless formodulename is specified. AppleEvent descriptors with descriptor type typeFSS are returned as FSSpec objects. AppleEvent record descriptors are returned as Python dictionaries, with 4-character string keys and elements recursively unpacked.

The optional formodulename argument is used by the stub packages generated by *gensuitemodule*, and ensures that the OSA classes for object specifiers are looked up in the correct module. This ensures that if, say, the Finder returns an object specifier for a window you get an instance of Finder.Window and not a generic aetypes.Window. The former knows about all the properties and elements a window has in the Finder, while the latter knows no such things.

参见:

Module *Carbon.AE* Built-in access to Apple Event Manager routines.

Module *aetypes* Python definitions of codes for Apple Event descriptor types.

38.4 aetypes —AppleEvent objects

The *aetypes* defines classes used to represent Apple Event data descriptors and Apple Event object specifiers.

Apple Event data is contained in descriptors, and these descriptors are typed. For many descriptors the Python representation is simply the corresponding Python type: `typeText` in OSA is a Python string, `typeFloat` is a float, etc. For OSA types that have no direct Python counterpart this module declares classes. Packing and unpacking instances of these classes is handled automatically by *aepack*.

An object specifier is essentially an address of an object implemented in an Apple Event server. An Apple Event specifier is used as the direct object for an Apple Event or as the argument of an optional parameter. The *aetypes* module contains the base classes for OSA classes and properties, which are used by the packages generated by *gensuitemodule* to populate the classes and properties in a given suite.

For reasons of backward compatibility, and for cases where you need to script an application for which you have not generated the stub package this module also contains object specifiers for a number of common OSA classes such as `Document`, `Window`, `Character`, etc.

注解: This module has been removed in Python 3.x.

The `AEObjects` module defines the following classes to represent Apple Event descriptor data:

class aetypes.**Unknown**(*type*, *data*)
The representation of OSA descriptor data for which the *aepack* and *aetypes* modules have no support, i.e. anything that is not represented by the other classes here and that is not equivalent to a simple Python value.

class aetypes.**Enum**(*enum*)
An enumeration value with the given 4-character string value.

class aetypes.**InsertionLoc**(*of*, *pos*)
Position `pos` in object `of`.

class aetypes.**Boolean**(*bool*)
A boolean.

class aetypes.**StyledText**(*style*, *text*)
Text with style information (font, face, etc) included.

class aetypes.**AEText**(*script*, *style*, *text*)
Text with script system and style information included.

class aetypes.**IntlText**(*script*, *language*, *text*)
Text with script system and language information included.

class aetypes.**IntlWritingCode**(*script*, *language*)
Script system and language information.

class aetypes.**QDPoint**(*v*, *h*)
A quickdraw point.

class aetypes.**QDRectangle**(*v0*, *h0*, *v1*, *h1*)
A quickdraw rectangle.

class aetypes.**RGBColor**(*r*, *g*, *b*)
A color.

class aetypes.**Type**(*type*)
An OSA type value with the given 4-character name.

class aetypes.**Keyword**(*name*)
An OSA keyword with the given 4-character name.

class aetypes.**Range**(*start*, *stop*)
A range.

class aetypes.**Ordinal**(*abso*)
Non-numeric absolute positions, such as `"firs"`, first, or `"midd"`, middle.

class aetypes.**Logical**(*logc*, *term*)
The logical expression of applying operator `logc` to `term`.

class aetypes.**Comparison**(*obj1*, *relo*, *obj2*)
The comparison `relo` of `obj1` to `obj2`.

The following classes are used as base classes by the generated stub packages to represent AppleScript classes and properties in Python:

class aetypes.**ComponentItem**(*which*[, *fr*])
Abstract baseclass for an OSA class. The subclass should set the class attribute `want` to the 4-character OSA class code. Instances of subclasses of this class are equivalent to AppleScript Object Specifiers. Upon instantiation you should pass a selector in `which`, and optionally a parent object in `fr`.

class aetypes.**NProperty**(*fr*)
Abstract baseclass for an OSA property. The subclass should set the class attributes `want` and `which` to designate which property we are talking about. Instances of subclasses of this class are Object Specifiers.

class aetypes.**ObjectSpecifier**(*want*, *form*, *seld*[, *fr*])
Base class of `ComponentItem` and `NProperty`, a general OSA Object Specifier. See the Apple Open Scripting Architecture documentation for the parameters. Note that this class is not abstract.

38.5 MiniAEFrame —Open Scripting Architecture server support

The module *MiniAEFrame* provides a framework for an application that can function as an Open Scripting Architecture (OSA) server, i.e. receive and process AppleEvents. It can be used in conjunction with *FrameWork* or standalone. As an example, it is used in **PythonCGISlave**.

The *MiniAEFrame* module defines the following classes:

class MiniAEFrame.**AEServer**
A class that handles AppleEvent dispatch. Your application should subclass this class together with either *MiniApplication* or *FrameWork.Application*. Your `__init__()` method should call the `__init__()` method for both classes.

class MiniAEFrame.**MiniApplication**
A class that is more or less compatible with *FrameWork.Application* but with less functionality. Its event loop supports the apple menu, command-dot and AppleEvents; other events are passed on to the Python interpreter and/or Sioux. Useful if your application wants to use *AEServer* but does not provide its own windows, etc.

38.5.1 AEServer Objects

AEServer.**installaehandler**(*classe*, *type*, *callback*)
Installs an AppleEvent handler. *classe* and *type* are the four-character OSA Class and Type designators, `'****'` wildcards are allowed. When a matching AppleEvent is received the parameters are decoded and your callback is invoked.

AEServer.**callback**(*_object*, ***kwargs*)
Your callback is called with the OSA Direct Object as first positional parameter. The other parameters are passed as keyword arguments, with the 4-character designator as name. Three extra keyword parameters are passed: `_class` and `_type` are the Class and Type designators and `_attributes` is a dictionary with the AppleEvent attributes.

The return value of your method is packed with *aetools.packevent()* and sent as reply.

Note that there are some serious problems with the current design. AppleEvents which have non-identifier 4-character designators for arguments are not implementable, and it is not possible to return an error to the originator. This will be addressed in a future release.

In addition, support modules have been pre-generated for `Finder`, `Terminal`, `Explorer`, `Netscape`, `CodeWarrior`, `SystemEvents` and `StdSuites`.

CHAPTER 39

SGI IRIX Specific Services

The modules described in this chapter provide interfaces to features that are unique to SGI's IRIX operating system (versions 4 and 5).

39.1 `al` —Audio functions on the SGI

2.6 版后已移除: The *al* module has been removed in Python 3.

This module provides access to the audio facilities of the SGI Indy and Indigo workstations. See section 3A of the IRIX man pages for details. You'll need to read those man pages to understand what these functions do! Some of the functions are not available in IRIX releases before 4.0.5. Again, see the manual to check whether a specific function is available on your platform.

All functions and methods defined in this module are equivalent to the C functions with `AL` prefixed to their name.

Symbolic constants from the C header file `<audio.h>` are defined in the standard module *AL*, see below.

> **警告:** The current version of the audio library may dump core when bad argument values are passed rather than returning an error status. Unfortunately, since the precise circumstances under which this may happen are undocumented and hard to check, the Python interface can provide no protection against this kind of problems. (One example is specifying an excessive queue size —there is no documented upper limit.)

The module defines the following functions:

`al.`**`openport`**`(`*name*, *direction*[, *config*]`)`
The name and direction arguments are strings. The optional *config* argument is a configuration object as returned by *newconfig()*. The return value is an *audio port object*; methods of audio port objects are described below.

`al.`**`newconfig`**`()`
The return value is a new *audio configuration object*; methods of audio configuration objects are described below.

al.**queryparams**(*device*)
The device argument is an integer. The return value is a list of integers containing the data returned by ALqueryparams().

al.**getparams**(*device*, *list*)
The *device* argument is an integer. The list argument is a list such as returned by *queryparams()*; it is modified in place (!).

al.**setparams**(*device*, *list*)
The *device* argument is an integer. The *list* argument is a list such as returned by *queryparams()*.

39.1.1 Configuration Objects

Configuration objects returned by *newconfig()* have the following methods:

audio configuration.getqueuesize()
Return the queue size.

audio configuration.setqueuesize(size)
Set the queue size.

audio configuration.getwidth()
Get the sample width.

audio configuration.setwidth(width)
Set the sample width.

audio configuration.getchannels()
Get the channel count.

audio configuration.setchannels(nchannels)
Set the channel count.

audio configuration.getsampfmt()
Get the sample format.

audio configuration.setsampfmt(sampfmt)
Set the sample format.

audio configuration.getfloatmax()
Get the maximum value for floating sample formats.

audio configuration.setfloatmax(floatmax)
Set the maximum value for floating sample formats.

39.1.2 Port Objects

Port objects, as returned by *openport()*, have the following methods:

audio port.closeport()
Close the port.

audio port.getfd()
Return the file descriptor as an int.

audio port.getfilled()
Return the number of filled samples.

audio port.getfillable()
Return the number of fillable samples.

audio port.readsamps(nsamples)
Read a number of samples from the queue, blocking if necessary. Return the data as a string containing the raw data, (e.g., 2 bytes per sample in big-endian byte order (high byte, low byte) if you have set the sample width to 2 bytes).

audio port.writesamps(samples)
Write samples into the queue, blocking if necessary. The samples are encoded as described for the `readsamps()` return value.

audio port.getfillpoint()
Return the 'fill point'.

audio port.setfillpoint(fillpoint)
Set the 'fill point'.

audio port.getconfig()
Return a configuration object containing the current configuration of the port.

audio port.setconfig(config)
Set the configuration from the argument, a configuration object.

audio port.getstatus(list)
Get status information on last error.

39.2 AL —Constants used with the al module

2.6 版后已移除: The *AL* module has been removed in Python 3.

This module defines symbolic constants needed to use the built-in module *al* (see above); they are equivalent to those defined in the C header file `<audio.h>` except that the name prefix `AL_` is omitted. Read the module source for a complete list of the defined names. Suggested use:

```
import al
from AL import *
```

39.3 cd —CD-ROM access on SGI systems

2.6 版后已移除: The *cd* module has been removed in Python 3.

This module provides an interface to the Silicon Graphics CD library. It is available only on Silicon Graphics systems.

The way the library works is as follows. A program opens the CD-ROM device with *open()* and creates a parser to parse the data from the CD with *createparser()*. The object returned by *open()* can be used to read data from the CD, but also to get status information for the CD-ROM device, and to get information about the CD, such as the table of contents. Data from the CD is passed to the parser, which parses the frames, and calls any callback functions that have previously been added.

An audio CD is divided into *tracks* or *programs* (the terms are used interchangeably). Tracks can be subdivided into *indices*. An audio CD contains a *table of contents* which gives the starts of the tracks on the CD. Index 0 is usually the pause before the start of a track. The start of the track as given by the table of contents is normally the start of index 1.

Positions on a CD can be represented in two ways. Either a frame number or a tuple of three values, minutes, seconds and frames. Most functions use the latter representation. Positions can be both relative to the beginning of the CD, and to the beginning of the track.

Module *cd* defines the following functions and constants:

cd.**createparser**()
Create and return an opaque parser object. The methods of the parser object are described below.

cd.**msftoframe**(*minutes*, *seconds*, *frames*)
Converts a `(minutes, seconds, frames)` triple representing time in absolute time code into the corresponding CD frame number.

cd.**open**([*device*[, *mode*]])
Open the CD-ROM device. The return value is an opaque player object; methods of the player object are described below. The device is the name of the SCSI device file, e.g. `'/dev/scsi/sc0d4l0'`, or `None`. If omitted or `None`, the hardware inventory is consulted to locate a CD-ROM drive. The *mode*, if not omitted, should be the string `'r'`.

The module defines the following variables:

exception cd.**error**
Exception raised on various errors.

cd.**DATASIZE**
The size of one frame's worth of audio data. This is the size of the audio data as passed to the callback of type `audio`.

cd.**BLOCKSIZE**
The size of one uninterpreted frame of audio data.

The following variables are states as returned by `getstatus()`:

cd.**READY**
The drive is ready for operation loaded with an audio CD.

cd.**NODISC**
The drive does not have a CD loaded.

cd.**CDROM**
The drive is loaded with a CD-ROM. Subsequent play or read operations will return I/O errors.

cd.**ERROR**
An error occurred while trying to read the disc or its table of contents.

cd.**PLAYING**
The drive is in CD player mode playing an audio CD through its audio jacks.

cd.**PAUSED**
The drive is in CD layer mode with play paused.

cd.**STILL**
The equivalent of *PAUSED* on older (non 3301) model Toshiba CD-ROM drives. Such drives have never been shipped by SGI.

cd.**audio**
cd.**pnum**
cd.**index**
cd.**ptime**
cd.**atime**
cd.**catalog**
cd.**ident**
cd.**control**
Integer constants describing the various types of parser callbacks that can be set by the `addcallback()` method of CD parser objects (see below).

39.3.1 Player Objects

Player objects (returned by *open()*) have the following methods:

CD player.allowremoval()
Unlocks the eject button on the CD-ROM drive permitting the user to eject the caddy if desired.

CD player.bestreadsize()
Returns the best value to use for the *num_frames* parameter of the `readda()` method. Best is defined as the value that permits a continuous flow of data from the CD-ROM drive.

CD player.close()
Frees the resources associated with the player object. After calling `close()`, the methods of the object should no longer be used.

CD player.eject()
Ejects the caddy from the CD-ROM drive.

CD player.getstatus()
Returns information pertaining to the current state of the CD-ROM drive. The returned information is a tuple with the following values: *state*, *track*, *rtime*, *atime*, *ttime*, *first*, *last*, *scsi_audio*, *cur_block*. *rtime* is the time relative to the start of the current track; *atime* is the time relative to the beginning of the disc; *ttime* is the total time on the disc. For more information on the meaning of the values, see the man page *CDgetstatus(3dm)*. The value of *state* is one of the following: *ERROR*, *NODISC*, *READY*, *PLAYING*, *PAUSED*, *STILL*, or *CDROM*.

CD player.gettrackinfo(track)
Returns information about the specified track. The returned information is a tuple consisting of two elements, the start time of the track and the duration of the track.

CD player.msftoblock(min, sec, frame)
Converts a minutes, seconds, frames triple representing a time in absolute time code into the corresponding logical block number for the given CD-ROM drive. You should use *msftoframe()* rather than `msftoblock()` for comparing times. The logical block number differs from the frame number by an offset required by certain CD-ROM drives.

CD player.play(start, play)
Starts playback of an audio CD in the CD-ROM drive at the specified track. The audio output appears on the CD-ROM drive's headphone and audio jacks (if fitted). Play stops at the end of the disc. *start* is the number of the track at which to start playing the CD; if *play* is 0, the CD will be set to an initial paused state. The method `togglepause()` can then be used to commence play.

CD player.playabs(minutes, seconds, frames, play)
Like `play()`, except that the start is given in minutes, seconds, and frames instead of a track number.

CD player.playtrack(start, play)
Like `play()`, except that playing stops at the end of the track.

CD player.playtrackabs(track, minutes, seconds, frames, play)
Like `play()`, except that playing begins at the specified absolute time and ends at the end of the specified track.

CD player.preventremoval()
Locks the eject button on the CD-ROM drive thus preventing the user from arbitrarily ejecting the caddy.

CD player.readda(num_frames)
Reads the specified number of frames from an audio CD mounted in the CD-ROM drive. The return value is a string representing the audio frames. This string can be passed unaltered to the `parseframe()` method of the parser object.

CD player.seek(minutes, seconds, frames)
Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer

is set to an absolute time code location specified in *minutes*, *seconds*, and *frames*. The return value is the logical block number to which the pointer has been set.

CD player.seekblock(block)
Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to the specified logical block number. The return value is the logical block number to which the pointer has been set.

CD player.seektrack(track)
Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to the specified track. The return value is the logical block number to which the pointer has been set.

CD player.stop()
Stops the current playing operation.

CD player.togglepause()
Pauses the CD if it is playing, and makes it play if it is paused.

39.3.2 Parser Objects

Parser objects (returned by *createparser()*) have the following methods:

CD parser.addcallback(type, func, arg)
Adds a callback for the parser. The parser has callbacks for eight different types of data in the digital audio data stream. Constants for these types are defined at the *cd* module level (see above). The callback is called as follows: `func(arg, type, data)`, where *arg* is the user supplied argument, *type* is the particular type of callback, and *data* is the data returned for this *type* of callback. The type of the data depends on the *type* of callback as follows:

Type	Value
`audio`	String which can be passed unmodified to `al.writesamps()`.
`pnum`	Integer giving the program (track) number.
`index`	Integer giving the index number.
`ptime`	Tuple consisting of the program time in minutes, seconds, and frames.
`atime`	Tuple consisting of the absolute time in minutes, seconds, and frames.
`catalog`	String of 13 characters, giving the catalog number of the CD.
`ident`	String of 12 characters, giving the ISRC identification number of the recording. The string consists of two characters country code, three characters owner code, two characters giving the year, and five characters giving a serial number.
`control`	Integer giving the control bits from the CD subcode data

CD parser.deleteparser()
Deletes the parser and frees the memory it was using. The object should not be used after this call. This call is done automatically when the last reference to the object is removed.

CD parser.parseframe(frame)
Parses one or more frames of digital audio data from a CD such as returned by `readda()`. It determines which subcodes are present in the data. If these subcodes have changed since the last frame, then `parseframe()` executes a callback of the appropriate type passing to it the subcode data found in the frame. Unlike the C function, more than one frame of digital audio data can be passed to this method.

CD parser.removecallback(type)
Removes the callback for the given *type*.

CD parser.resetparser()
Resets the fields of the parser used for tracking subcodes to an initial state. `resetparser()` should be called after the disc has been changed.

39.4 fl —FORMS library for graphical user interfaces

2.6 版后已移除: The *fl* module has been removed in Python 3.

This module provides an interface to the FORMS Library by Mark Overmars. The source for the library can be retrieved by anonymous FTP from host `ftp.cs.ruu.nl`, directory `SGI/FORMS`. It was last tested with version 2.0b.

Most functions are literal translations of their C equivalents, dropping the initial `fl_` from their name. Constants used by the library are defined in module *FL* described below.

The creation of objects is a little different in Python than in C: instead of the 'current form' maintained by the library to which new FORMS objects are added, all functions that add a FORMS object to a form are methods of the Python object representing the form. Consequently, there are no Python equivalents for the C functions `fl_addto_form()` and `fl_end_form()`, and the equivalent of `fl_bgn_form()` is called *fl.make_form()*.

Watch out for the somewhat confusing terminology: FORMS uses the word *object* for the buttons, sliders etc. that you can place in a form. In Python, 'object' means any value. The Python interface to FORMS introduces two new Python object types: form objects (representing an entire form) and FORMS objects (representing one button, slider etc.). Hopefully this isn't too confusing.

There are no 'free objects' in the Python interface to FORMS, nor is there an easy way to add object classes written in Python. The FORMS interface to GL event handling is available, though, so you can mix FORMS with pure GL windows.

Please note: importing *fl* implies a call to the GL function `foreground()` and to the FORMS routine `fl_init()`.

39.4.1 Functions Defined in Module fl

Module *fl* defines the following functions. For more information about what they do, see the description of the equivalent C function in the FORMS documentation:

`fl.`**`make_form`**(*type*, *width*, *height*)
Create a form with given type, width and height. This returns a *form* object, whose methods are described below.

`fl.`**`do_forms`**()
The standard FORMS main loop. Returns a Python object representing the FORMS object needing interaction, or the special value `FL.EVENT`.

`fl.`**`check_forms`**()
Check for FORMS events. Returns what *do_forms()* above returns, or `None` if there is no event that immediately needs interaction.

`fl.`**`set_event_call_back`**(*function*)
Set the event callback function.

`fl.`**`set_graphics_mode`**(*rgbmode*, *doublebuffering*)
Set the graphics modes.

`fl.`**`get_rgbmode`**()
Return the current rgb mode. This is the value of the C global variable `fl_rgbmode`.

`fl.`**`show_message`**(*str1*, *str2*, *str3*)
Show a dialog box with a three-line message and an OK button.

`fl.`**`show_question`**(*str1*, *str2*, *str3*)
Show a dialog box with a three-line message and YES and NO buttons. It returns `1` if the user pressed YES, `0` if NO.

`fl.`**`show_choice`**(*str1*, *str2*, *str3*, *but1*[, *but2*[, *but3*]])
Show a dialog box with a three-line message and up to three buttons. It returns the number of the button clicked by the user (`1`, `2` or `3`).

fl.**show_input**(*prompt*, *default*)
Show a dialog box with a one-line prompt message and text field in which the user can enter a string. The second argument is the default input string. It returns the string value as edited by the user.

fl.**show_file_selector**(*message*, *directory*, *pattern*, *default*)
Show a dialog box in which the user can select a file. It returns the absolute filename selected by the user, or `None` if the user presses Cancel.

fl.**get_directory**()
fl.**get_pattern**()
fl.**get_filename**()
These functions return the directory, pattern and filename (the tail part only) selected by the user in the last *`show_file_selector()`* call.

fl.**qdevice**(*dev*)
fl.**unqdevice**(*dev*)
fl.**isqueued**(*dev*)
fl.**qtest**()
fl.**qread**()
fl.**qreset**()
fl.**qenter**(*dev*, *val*)
fl.**get_mouse**()
fl.**tie**(*button*, *valuator1*, *valuator2*)
These functions are the FORMS interfaces to the corresponding GL functions. Use these if you want to handle some GL events yourself when using `fl.do_events()`. When a GL event is detected that FORMS cannot handle, *`fl.do_forms()`* returns the special value `FL.EVENT` and you should call *`fl.qread()`* to read the event from the queue. Don't use the equivalent GL functions!

fl.**color**()
fl.**mapcolor**()
fl.**getmcolor**()
See the description in the FORMS documentation of `fl_color()`, `fl_mapcolor()` and `fl_getmcolor()`.

39.4.2 Form Objects

Form objects (returned by *`make_form()`* above) have the following methods. Each method corresponds to a C function whose name is prefixed with `fl_`; and whose first argument is a form pointer; please refer to the official FORMS documentation for descriptions.

All the `add_*()` methods return a Python object representing the FORMS object. Methods of FORMS objects are described below. Most kinds of FORMS object also have some methods specific to that kind; these methods are listed here.

form.**show_form**(*placement*, *bordertype*, *name*)
Show the form.

form.**hide_form**()
Hide the form.

form.**redraw_form**()
Redraw the form.

form.**set_form_position**(*x*, *y*)
Set the form's position.

form.**freeze_form**()
Freeze the form.

form.**unfreeze_form**()
Unfreeze the form.

form.**activate_form**()
Activate the form.

form.**deactivate_form**()
Deactivate the form.

form.**bgn_group**()
Begin a new group of objects; return a group object.

form.**end_group**()
End the current group of objects.

form.**find_first**()
Find the first object in the form.

form.**find_last**()
Find the last object in the form.

form.**add_box**(*type*, *x*, *y*, *w*, *h*, *name*)
Add a box object to the form. No extra methods.

form.**add_text**(*type*, *x*, *y*, *w*, *h*, *name*)
Add a text object to the form. No extra methods.

form.**add_clock**(*type*, *x*, *y*, *w*, *h*, *name*)
Add a clock object to the form. —Method: `get_clock()`.

form.**add_button**(*type*, *x*, *y*, *w*, *h*, *name*)
Add a button object to the form. —Methods: `get_button()`, `set_button()`.

form.**add_lightbutton**(*type*, *x*, *y*, *w*, *h*, *name*)
Add a lightbutton object to the form. —Methods: `get_button()`, `set_button()`.

form.**add_roundbutton**(*type*, *x*, *y*, *w*, *h*, *name*)
Add a roundbutton object to the form. —Methods: `get_button()`, `set_button()`.

form.**add_slider**(*type*, *x*, *y*, *w*, *h*, *name*)
Add a slider object to the form. —Methods: `set_slider_value()`, `get_slider_value()`, `set_slider_bounds()`, `get_slider_bounds()`, `set_slider_return()`, `set_slider_size()`, `set_slider_precision()`, `set_slider_step()`.

form.**add_valslider**(*type*, *x*, *y*, *w*, *h*, *name*)
Add a valslider object to the form. —Methods: `set_slider_value()`, `get_slider_value()`, `set_slider_bounds()`, `get_slider_bounds()`, `set_slider_return()`, `set_slider_size()`, `set_slider_precision()`, `set_slider_step()`.

form.**add_dial**(*type*, *x*, *y*, *w*, *h*, *name*)
Add a dial object to the form. —Methods: `set_dial_value()`, `get_dial_value()`, `set_dial_bounds()`, `get_dial_bounds()`.

form.**add_positioner**(*type*, *x*, *y*, *w*, *h*, *name*)
Add a positioner object to the form. —Methods: `set_positioner_xvalue()`, `set_positioner_yvalue()`, `set_positioner_xbounds()`, `set_positioner_ybounds()`, `get_positioner_xvalue()`, `get_positioner_yvalue()`, `get_positioner_xbounds()`, `get_positioner_ybounds()`.

form.**add_counter**(*type*, *x*, *y*, *w*, *h*, *name*)
Add a counter object to the form. —Methods: `set_counter_value()`, `get_counter_value()`,

`set_counter_bounds()`, `set_counter_step()`, `set_counter_precision()`, `set_counter_return()`.

`form.`**`add_input`**(*type*, *x*, *y*, *w*, *h*, *name*)
Add an input object to the form. —Methods: `set_input()`, `get_input()`, `set_input_color()`, `set_input_return()`.

`form.`**`add_menu`**(*type*, *x*, *y*, *w*, *h*, *name*)
Add a menu object to the form. —Methods: `set_menu()`, `get_menu()`, `addto_menu()`.

`form.`**`add_choice`**(*type*, *x*, *y*, *w*, *h*, *name*)
Add a choice object to the form. —Methods: `set_choice()`, `get_choice()`, `clear_choice()`, `addto_choice()`, `replace_choice()`, `delete_choice()`, `get_choice_text()`, `set_choice_fontsize()`, `set_choice_fontstyle()`.

`form.`**`add_browser`**(*type*, *x*, *y*, *w*, *h*, *name*)
Add a browser object to the form. —Methods: `set_browser_topline()`, `clear_browser()`, `add_browser_line()`, `addto_browser()`, `insert_browser_line()`, `delete_browser_line()`, `replace_browser_line()`, `get_browser_line()`, `load_browser()`, `get_browser_maxline()`, `select_browser_line()`, `deselect_browser_line()`, `deselect_browser()`, `isselected_browser_line()`, `get_browser()`, `set_browser_fontsize()`, `set_browser_fontstyle()`, `set_browser_specialkey()`.

`form.`**`add_timer`**(*type*, *x*, *y*, *w*, *h*, *name*)
Add a timer object to the form. —Methods: `set_timer()`, `get_timer()`.

Form objects have the following data attributes; see the FORMS documentation:

Name	C Type	Meaning
`window`	int (read-only)	GL window id
`w`	float	form width
`h`	float	form height
`x`	float	form x origin
`y`	float	form y origin
`deactivated`	int	nonzero if form is deactivated
`visible`	int	nonzero if form is visible
`frozen`	int	nonzero if form is frozen
`doublebuf`	int	nonzero if double buffering on

39.4.3 FORMS Objects

Besides methods specific to particular kinds of FORMS objects, all FORMS objects also have the following methods:

`FORMS object.set_call_back(function, argument)`
Set the object's callback function and argument. When the object needs interaction, the callback function will be called with two arguments: the object, and the callback argument. (FORMS objects without a callback function are returned by *`fl.do_forms()`* or *`fl.check_forms()`* when they need interaction.) Call this method without arguments to remove the callback function.

`FORMS object.delete_object()`
Delete the object.

`FORMS object.show_object()`
Show the object.

FORMS object.hide_object()
Hide the object.

FORMS object.redraw_object()
Redraw the object.

FORMS object.freeze_object()
Freeze the object.

FORMS object.unfreeze_object()
Unfreeze the object.

FORMS objects have these data attributes; see the FORMS documentation:

Name	C Type	Meaning
objclass	int (read-only)	object class
type	int (read-only)	object type
boxtype	int	box type
x	float	x origin
y	float	y origin
w	float	width
h	float	height
col1	int	primary color
col2	int	secondary color
align	int	alignment
lcol	int	label color
lsize	float	label font size
label	string	label string
lstyle	int	label style
pushed	int (read-only)	(see FORMS docs)
focus	int (read-only)	(see FORMS docs)
belowmouse	int (read-only)	(see FORMS docs)
frozen	int (read-only)	(see FORMS docs)
active	int (read-only)	(see FORMS docs)
input	int (read-only)	(see FORMS docs)
visible	int (read-only)	(see FORMS docs)
radio	int (read-only)	(see FORMS docs)
automatic	int (read-only)	(see FORMS docs)

39.5 FL —Constants used with the fl module

2.6 版后已移除: The *FL* module has been removed in Python 3.

This module defines symbolic constants needed to use the built-in module *fl* (see above); they are equivalent to those defined in the C header file <forms.h> except that the name prefix FL_ is omitted. Read the module source for a complete list of the defined names. Suggested use:

```
import fl
from FL import *
```

39.6 flp —Functions for loading stored FORMS designs

2.6 版后已移除: The *flp* module has been removed in Python 3.

This module defines functions that can read form definitions created by the 'form designer' (**fdesign**) program that comes with the FORMS library (see module *fl* above).

For now, see the file `flp.doc` in the Python library source directory for a description.

XXX A complete description should be inserted here!

39.7 fm —*Font Manager* interface

2.6 版后已移除: The *fm* module has been removed in Python 3.

This module provides access to the IRIS *Font Manager* library. It is available only on Silicon Graphics machines. See also: *4Sight User's Guide*, section 1, chapter 5: "Using the IRIS Font Manager."

This is not yet a full interface to the IRIS Font Manager. Among the unsupported features are: matrix operations; cache operations; character operations (use string operations instead); some details of font info; individual glyph metrics; and printer matching.

It supports the following operations:

`fm.`**`init`**`()`
: Initialization function. Calls `fminit()`. It is normally not necessary to call this function, since it is called automatically the first time the *fm* module is imported.

`fm.`**`findfont`**`(`*fontname*`)`
: Return a font handle object. Calls `fmfindfont(fontname)`.

`fm.`**`enumerate`**`()`
: Returns a list of available font names. This is an interface to `fmenumerate()`.

`fm.`**`prstr`**`(`*string*`)`
: Render a string using the current font (see the `setfont()` font handle method below). Calls `fmprstr(string)`.

`fm.`**`setpath`**`(`*string*`)`
: Sets the font search path. Calls `fmsetpath(string)`. (XXX Does not work!?!)

`fm.`**`fontpath`**`()`
: Returns the current font search path.

Font handle objects support the following operations:

`font handle.scalefont(factor)`
: Returns a handle for a scaled version of this font. Calls `fmscalefont(fh, factor)`.

`font handle.setfont()`
: Makes this font the current font. Note: the effect is undone silently when the font handle object is deleted. Calls `fmsetfont(fh)`.

`font handle.getfontname()`
: Returns this font's name. Calls `fmgetfontname(fh)`.

`font handle.getcomment()`
: Returns the comment string associated with this font. Raises an exception if there is none. Calls `fmgetcomment(fh)`.

font handle.getfontinfo()
Returns a tuple giving some pertinent data about this font. This is an interface to `fmgetfontinfo()`. The returned tuple contains the following numbers: `(printermatched, fixed_width, xorig, yorig, xsize, ysize, height, nglyphs)`.

font handle.getstrwidth(string)
Returns the width, in pixels, of *string* when drawn in this font. Calls `fmgetstrwidth(fh, string)`.

39.8 gl —*Graphics Library* interface

2.6 版后已移除: The *gl* module has been removed in Python 3.

This module provides access to the Silicon Graphics *Graphics Library*. It is available only on Silicon Graphics machines.

警告: Some illegal calls to the GL library cause the Python interpreter to dump core. In particular, the use of most GL calls is unsafe before the first window is opened.

The module is too large to document here in its entirety, but the following should help you to get started. The parameter conventions for the C functions are translated to Python as follows:

- All (short, long, unsigned) int values are represented by Python integers.
- All float and double values are represented by Python floating point numbers. In most cases, Python integers are also allowed.
- All arrays are represented by one-dimensional Python lists. In most cases, tuples are also allowed.
- All string and character arguments are represented by Python strings, for instance, `winopen('Hi There!')` and `rotate(900, 'z')`.
- All (short, long, unsigned) integer arguments or return values that are only used to specify the length of an array argument are omitted. For example, the C call

```
lmdef(deftype, index, np, props)
```

  is translated to Python as

```
lmdef(deftype, index, props)
```

- Output arguments are omitted from the argument list; they are transmitted as function return values instead. If more than one value must be returned, the return value is a tuple. If the C function has both a regular return value (that is not omitted because of the previous rule) and an output argument, the return value comes first in the tuple. Examples: the C call

```
getmcolor(i, &red, &green, &blue)
```

  is translated to Python as

```
red, green, blue = getmcolor(i)
```

The following functions are non-standard or have special argument conventions:

gl.**varray**(*argument*)
Equivalent to but faster than a number of `v3d()` calls. The *argument* is a list (or tuple) of points. Each point must be a tuple of coordinates `(x, y, z)` or `(x, y)`. The points may be 2- or 3-dimensional but must all have the same dimension. Float and int values may be mixed however. The points are always converted to 3D double

precision points by assuming `z = 0.0` if necessary (as indicated in the man page), and for each point `v3d()` is called.

`gl.`**`nvarray`**`()`
Equivalent to but faster than a number of `n3f` and `v3f` calls. The argument is an array (list or tuple) of pairs of normals and points. Each pair is a tuple of a point and a normal for that point. Each point or normal must be a tuple of coordinates `(x, y, z)`. Three coordinates must be given. Float and int values may be mixed. For each pair, `n3f()` is called for the normal, and then `v3f()` is called for the point.

`gl.`**`vnarray`**`()`
Similar to `nvarray()` but the pairs have the point first and the normal second.

`gl.`**`nurbssurface`**`(`*s_k*, *t_k*, *ctl*, *s_ord*, *t_ord*, *type*`)`
Defines a nurbs surface. The dimensions of `ctl[][]` are computed as follows: `[len(s_k) - s_ord]`, `[len(t_k) - t_ord]`.

`gl.`**`nurbscurve`**`(`*knots*, *ctlpoints*, *order*, *type*`)`
Defines a nurbs curve. The length of ctlpoints is `len(knots) - order`.

`gl.`**`pwlcurve`**`(`*points*, *type*`)`
Defines a piecewise-linear curve. *points* is a list of points. *type* must be `N_ST`.

`gl.`**`pick`**`(`*n*`)`
`gl.`**`select`**`(`*n*`)`
The only argument to these functions specifies the desired size of the pick or select buffer.

`gl.`**`endpick`**`()`
`gl.`**`endselect`**`()`
These functions have no arguments. They return a list of integers representing the used part of the pick/select buffer. No method is provided to detect buffer overrun.

Here is a tiny but complete example GL program in Python:

```
import gl, GL, time

def main():
    gl.foreground()
    gl.prefposition(500, 900, 500, 900)
    w = gl.winopen('CrissCross')
    gl.ortho2(0.0, 400.0, 0.0, 400.0)
    gl.color(GL.WHITE)
    gl.clear()
    gl.color(GL.RED)
    gl.bgnline()
    gl.v2f(0.0, 0.0)
    gl.v2f(400.0, 400.0)
    gl.endline()
    gl.bgnline()
    gl.v2f(400.0, 0.0)
    gl.v2f(0.0, 400.0)
    gl.endline()
    time.sleep(5)

main()
```

参见:

PyOpenGL: The Python OpenGL Binding An interface to OpenGL is also available; see information about the **PyOpenGL** project online at http://pyopengl.sourceforge.net/. This may be a better option if support for SGI hardware from before about 1996 is not required.

39.9 DEVICE —Constants used with the gl module

2.6 版后已移除: The *DEVICE* module has been removed in Python 3.

This modules defines the constants used by the Silicon Graphics *Graphics Library* that C programmers find in the header file `<gl/device.h>`. Read the module source file for details.

39.10 GL —Constants used with the gl module

2.6 版后已移除: The *GL* module has been removed in Python 3.

This module contains constants used by the Silicon Graphics *Graphics Library* from the C header file `<gl/gl.h>`. Read the module source file for details.

39.11 imgfile —Support for SGI imglib files

2.6 版后已移除: The *imgfile* module has been removed in Python 3.

The *imgfile* module allows Python programs to access SGI imglib image files (also known as `.rgb` files). The module is far from complete, but is provided anyway since the functionality that there is enough in some cases. Currently, colormap files are not supported.

The module defines the following variables and functions:

exception imgfile.**error**
: This exception is raised on all errors, such as unsupported file type, etc.

imgfile.**getsizes**(*file*)
: This function returns a tuple `(x, y, z)` where *x* and *y* are the size of the image in pixels and *z* is the number of bytes per pixel. Only 3 byte RGB pixels and 1 byte greyscale pixels are currently supported.

imgfile.**read**(*file*)
: This function reads and decodes the image on the specified file, and returns it as a Python string. The string has either 1 byte greyscale pixels or 4 byte RGBA pixels. The bottom left pixel is the first in the string. This format is suitable to pass to `gl.lrectwrite()`, for instance.

imgfile.**readscaled**(*file*, *x*, *y*, *filter*[, *blur*])
: This function is identical to read but it returns an image that is scaled to the given *x* and *y* sizes. If the *filter* and *blur* parameters are omitted scaling is done by simply dropping or duplicating pixels, so the result will be less than perfect, especially for computer-generated images.

  Alternatively, you can specify a filter to use to smooth the image after scaling. The filter forms supported are `'impulse'`, `'box'`, `'triangle'`, `'quadratic'` and `'gaussian'`. If a filter is specified *blur* is an optional parameter specifying the blurriness of the filter. It defaults to `1.0`.

  readscaled() makes no attempt to keep the aspect ratio correct, so that is the users' responsibility.

imgfile.**ttob**(*flag*)
: This function sets a global flag which defines whether the scan lines of the image are read or written from bottom to top (flag is zero, compatible with SGI GL) or from top to bottom(flag is one, compatible with X). The default is zero.

imgfile.**write**(*file*, *data*, *x*, *y*, *z*)
: This function writes the RGB or greyscale data in *data* to image file *file*. *x* and *y* give the size of the image, *z* is 1 for 1 byte greyscale images or 3 for RGB images (which are stored as 4 byte values of which only the lower three bytes are used). These are the formats returned by `gl.lrectread()`.

39.12 jpeg —Read and write JPEG files

2.6 版后已移除: The *jpeg* module has been removed in Python 3.

The module *jpeg* provides access to the jpeg compressor and decompressor written by the Independent JPEG Group (IJG). JPEG is a standard for compressing pictures; it is defined in ISO 10918. For details on JPEG or the Independent JPEG Group software refer to the JPEG standard or the documentation provided with the software.

A portable interface to JPEG image files is available with the Python Imaging Library (PIL) by Fredrik Lundh. Information on PIL is available at http://www.pythonware.com/products/pil/.

The *jpeg* module defines an exception and some functions.

exception jpeg.**error**
Exception raised by *compress()* and *decompress()* in case of errors.

jpeg.**compress**(*data*, *w*, *h*, *b*)
Treat data as a pixmap of width *w* and height *h*, with *b* bytes per pixel. The data is in SGI GL order, so the first pixel is in the lower-left corner. This means that `gl.lrectread()` return data can immediately be passed to *compress()*. Currently only 1 byte and 4 byte pixels are allowed, the former being treated as greyscale and the latter as RGB color. *compress()* returns a string that contains the compressed picture, in JFIF format.

jpeg.**decompress**(*data*)
Data is a string containing a picture in JFIF format. It returns a tuple `(data, width, height, bytesperpixel)`. Again, the data is suitable to pass to `gl.lrectwrite()`.

jpeg.**setoption**(*name*, *value*)
Set various options. Subsequent *compress()* and *decompress()* calls will use these options. The following options are available:

Option	Effect
`'forcegray'`	Force output to be grayscale, even if input is RGB.
`'quality'`	Set the quality of the compressed image to a value between `0` and `100` (default is `75`). This only affects compression.
`'optimize'`	Perform Huffman table optimization. Takes longer, but results in smaller compressed image. This only affects compression.
`'smooth'`	Perform inter-block smoothing on uncompressed image. Only useful for low- quality images. This only affects decompression.

参见:

JPEG Still Image Data Compression Standard The canonical reference for the JPEG image format, by Pennebaker and Mitchell.

Information Technology - Digital Compression and Coding of Continuous-tone Still Images - Requirements and Guidelines
The ISO standard for JPEG is also published as ITU T.81. This is available online in PDF form.

CHAPTER 40

SunOS Specific Services

The modules described in this chapter provide interfaces to features that are unique to SunOS 5 (also known as Solaris version 2).

40.1 sunaudiodev —Access to Sun audio hardware

2.6 版后已移除: The *sunaudiodev* module has been removed in Python 3.

This module allows you to access the Sun audio interface. The Sun audio hardware is capable of recording and playing back audio data in u-LAW format with a sample rate of 8K per second. A full description can be found in the *audio(7I)* manual page.

The module *SUNAUDIODEV* defines constants which may be used with this module.

This module defines the following variables and functions:

exception sunaudiodev.**error**
: This exception is raised on all errors. The argument is a string describing what went wrong.

sunaudiodev.**open**(*mode*)
: This function opens the audio device and returns a Sun audio device object. This object can then be used to do I/O on. The *mode* parameter is one of `'r'` for record-only access, `'w'` for play-only access, `'rw'` for both and `'control'` for access to the control device. Since only one process is allowed to have the recorder or player open at the same time it is a good idea to open the device only for the activity needed. See *audio(7I)* for details.

  As per the manpage, this module first looks in the environment variable `AUDIODEV` for the base audio device filename. If not found, it falls back to `/dev/audio`. The control device is calculated by appending "ctl" to the base audio device.

40.1.1 Audio Device Objects

The audio device objects are returned by *open()* define the following methods (except `control` objects which only provide `getinfo()`, `setinfo()`, `fileno()`, and `drain()`):

audio device.close()
This method explicitly closes the device. It is useful in situations where deleting the object does not immediately close it since there are other references to it. A closed device should not be used again.

audio device.fileno()
Returns the file descriptor associated with the device. This can be used to set up `SIGPOLL` notification, as described below.

audio device.drain()
This method waits until all pending output is processed and then returns. Calling this method is often not necessary: destroying the object will automatically close the audio device and this will do an implicit drain.

audio device.flush()
This method discards all pending output. It can be used avoid the slow response to a user's stop request (due to buffering of up to one second of sound).

audio device.getinfo()
This method retrieves status information like input and output volume, etc. and returns it in the form of an audio status object. This object has no methods but it contains a number of attributes describing the current device status. The names and meanings of the attributes are described in `<sun/audioio.h>` and in the *audio(7I)* manual page. Member names are slightly different from their C counterparts: a status object is only a single structure. Members of the `play` substructure have `o_` prepended to their name and members of the `record` structure have `i_`. So, the C member `play.sample_rate` is accessed as `o_sample_rate`, `record.gain` as `i_gain` and `monitor_gain` plainly as `monitor_gain`.

audio device.ibufcount()
This method returns the number of samples that are buffered on the recording side, i.e. the program will not block on a `read()` call of so many samples.

audio device.obufcount()
This method returns the number of samples buffered on the playback side. Unfortunately, this number cannot be used to determine a number of samples that can be written without blocking since the kernel output queue length seems to be variable.

audio device.read(size)
This method reads *size* samples from the audio input and returns them as a Python string. The function blocks until enough data is available.

audio device.setinfo(status)
This method sets the audio device status parameters. The *status* parameter is a device status object as returned by `getinfo()` and possibly modified by the program.

audio device.write(samples)
Write is passed a Python string containing audio samples to be played. If there is enough buffer space free it will immediately return, otherwise it will block.

The audio device supports asynchronous notification of various events, through the SIGPOLL signal. Here's an example of how you might enable this in Python:

```
def handle_sigpoll(signum, frame):
    print 'I got a SIGPOLL update'

import fcntl, signal, STROPTS

```

(下页继续)

(续上页)

```
signal.signal(signal.SIGPOLL, handle_sigpoll)
fcntl.ioctl(audio_obj.fileno(), STROPTS.I_SETSIG, STROPTS.S_MSG)
```

40.2 SUNAUDIODEV —Constants used with sunaudiodev

2.6 版后已移除: The *SUNAUDIODEV* module has been removed in Python 3.

This is a companion module to *sunaudiodev* which defines useful symbolic constants like MIN_GAIN, MAX_GAIN, SPEAKER, etc. The names of the constants are the same names as used in the C include file <sun/audioio.h>, with the leading string AUDIO_ stripped.

CHAPTER 41

未创建文档的模块

以下是目前未创建文档模块的速览列表，但其它们应创建文档。欢迎为他们提供文档！（通过电子邮件发送到 docs@python.org ）

本章的想法和原内容取自 Fredrik Lundh 的帖子；本章的具体内容已经大幅修改。

41.1 Miscellaneous useful utilities

Some of these are very old and/or not very robust; marked with "hmm."

ihooks —Import hook support (for *rexec*; may become obsolete). Removed in Python 3.x.

41.2 平台特定模块

这些模块用于实现*os.path* 模块，除此之外没有文档。几乎没有必要创建这些文档。

ntpath —Implementation of *os.path* on Win32, Win64, WinCE, and OS/2 platforms.

posixpath —在 POSIX 上实现*os.path* 。

bsddb185 —Backwards compatibility module for systems which still use the Berkeley DB 1.85 module. It is normally only available on certain BSD Unix-based systems. It should never be used directly.

41.3 Multimedia

audiodev —Platform-independent API for playing audio data. Removed in Python 3.x.

linuxaudiodev —Play audio data on the Linux audio device. Replaced in Python 2.3 by the *ossaudiodev* module. Removed in Python 3.x.

sunaudio —Interpret Sun audio headers (may become obsolete or a tool/demo). Removed in Python 3.x.

toaiff —Convert "arbitrary" sound files to AIFF files; should probably become a tool or demo. Requires the external program **sox**. Removed in Python 3.x.

41.4 Undocumented Mac OS modules

41.4.1 applesingle —AppleSingle decoder

2.6 版后已移除.

41.4.2 buildtools —Helper module for BuildApplet and Friends

2.4 版后已移除.

41.4.3 cfmfile —Code Fragment Resource module

cfmfile is a module that understands Code Fragments and the accompanying "cfrg" resources. It can parse them and merge them, and is used by BuildApplication to combine all plugin modules to a single executable.

2.4 版后已移除.

41.4.4 icopen —Internet Config replacement for open()

Importing *icopen* will replace the built-in *open()* with a version that uses Internet Config to set file type and creator for new files.

2.6 版后已移除.

41.4.5 macerrors —Mac OS Errors

macerrors contains constant definitions for many Mac OS error codes.

2.6 版后已移除.

41.4.6 macresource —Locate script resources

macresource helps scripts finding their resources, such as dialogs and menus, without requiring special case code for when the script is run under MacPython, as a MacPython applet or under OSX Python.

2.6 版后已移除.

41.4.7 Nav —NavServices calls

A low-level interface to Navigation Services.

2.6 版后已移除.

41.4.8 PixMapWrapper —Wrapper for PixMap objects

PixMapWrapper wraps a PixMap object with a Python object that allows access to the fields by name. It also has methods to convert to and from PIL images.

2.6 版后已移除.

41.4.9 videoreader —Read QuickTime movies

videoreader reads and decodes QuickTime movies and passes a stream of images to your program. It also provides some support for audio tracks.

2.6 版后已移除.

41.4.10 W —Widgets built on FrameWork

The *W* widgets are used extensively in the **IDE**.

2.6 版后已移除.

41.5 Obsolete

These modules are not normally available for import; additional work must be done to make them available.

These extension modules written in C are not built by default. Under Unix, these must be enabled by uncommenting the appropriate lines in Modules/Setup in the build tree and either rebuilding Python if the modules are statically linked, or building and installing the shared object if using dynamically-loaded extensions.

timing —Measure time intervals to high resolution (use *time.clock()* instead). Removed in Python 3.x.

41.6 SGI-specific Extension modules

The following are SGI specific, and may be out of touch with the current version of reality.

cl —Interface to the SGI compression library.

sv —Interface to the "simple video" board on SGI Indigo (obsolete hardware). Removed in Python 3.x.

APPENDIX A

术语对照表

>>> 交互式终端中默认的 Python 提示符。往往会显示于能以交互方式在解释器里执行的样例代码之前。

... The default Python prompt of the interactive shell when entering code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

2to3 一个将 Python 2.x 代码转换为 Python 3.x 代码的工具，能够处理大部分通过解析源码并遍历解析树可检测到的不兼容问题。

2to3 包含在标准库中，模块名为`lib2to3`；并提供一个独立入口点 `Tools/scripts/2to3`。参见*2to3 - 自动将 Python 2 代码转为 Python 3 代码*。

abstract base class –抽象基类 Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections` module), numbers (in the `numbers` module), and streams (in the `io` module). You can create your own ABCs with the `abc` module.

argument –参数 A value passed to a *function* (or *method*) when calling the function. There are two types of arguments:

- *关键字参数*: 在函数调用中前面带有标识符（例如 `name=`）或者作为包含在前面带有 `**` 的字典里的值传入。举例来说，`3` 和 `5` 在以下对`complex()` 的调用中均属于关键字参数:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *位置参数*: 不属于关键字参数的参数。位置参数可出现于参数列表的开头以及/或者作为前面带有 `*` 的*iterable* 里的元素被传入。举例来说，`3` 和 `5` 在以下调用中均属于位置参数:

```
complex(3, 5)
complex(*(3, 5))
```

参数会被赋值给函数体中对应的局部变量。有关赋值规则参见 calls 一节。根据语法，任何表达式都可用来表示一个参数；最终算出的值会被赋给对应的局部变量。

See also the *parameter* glossary entry and the FAQ question on the difference between arguments and parameters.

attribute –属性 关联到一个对象的值，可以使用点号表达式通过其名称来引用。例如，如果一个对象 *o* 具有一个属性 *a*，就可以用 *o.a* 来引用它。

BDFL Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python’s creator.

bytes-like object –字节类对象 An object that supports the buffer protocol, like *str*, *bytearray* or *memoryview*. Bytes-like objects can be used for various operations that expect binary data, such as compression, saving to a binary file or sending over a socket. Some operations need the binary data to be mutable, in which case not all bytes-like objects can apply.

bytecode –字节码 Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This “intermediate language” is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

字节码指令列表可以在*dis 模块* 的文档中查看。

class –类 用来创建用户定义对象的模板。类定义通常包含对该类的实例进行操作的方法定义。

classic class Any class which does not inherit from *object*. See *new-style class*. Classic classes have been removed in Python 3.

coercion –强制类型转换 The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` built-in function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

complex number –复数 对普通实数系统的扩展，其中所有数字都被表示为一个实部和一个虚部的和。虚数是虚数单位（`-1` 的平方根）的实倍数，通常在数学中写为 `i`，在工程学中写为 `j`。Python 内置了对复数的支持，采用工程学标记方式；虚部带有一个 `j` 后缀，例如 `3+1j`。如果需要*math* 模块内对象的对应复数版本，请使用*cmath*，复数的使用是一个比较高级的数学特性。如果你感觉没有必要，忽略它们也几乎不会有任何问题。

context manager –上下文管理器 在 `with` 语句中使用，通过定义 `__enter__()` 和 `__exit__()` 方法来控制环境状态的对象。参见 **PEP 343**。

CPython Python 编程语言的规范实现，在 python.org 上发布。”CPython” 一词用于在必要时将此实现与其他实现例如 Jython 或 IronPython 相区别。

decorator –装饰器 返回值为另一个函数的函数，通常使用 `@wrapper` 语法形式来进行函数变换。装饰器的常见例子包括*classmethod()* 和*staticmethod()*。

装饰器语法只是一种语法糖，以下两个函数定义在语义上完全等价:

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

同的样概念也适用于类，但通常较少这样使用。有关装饰器的详情可参见 函数定义和 类定义的文档。

descriptor –描述器 Any *new-style* object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

有关描述符的方法的详情可参看 descriptors。

dictionary –字典 An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

dictionary view –字典视图 The objects returned from *dict.viewkeys()*, *dict.viewvalues()*, and *dict.viewitems()* are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See 字典视图对象.

docstring –文档字符串 作为类、函数或模块之内的第一个表达式出现的字符串字面值。它在代码执行时会被忽略，但会被解释器识别并放入所在类、函数或模块的 `__doc__` 属性中。由于它可用于代码内省，因此是对象存放文档的规范位置。

duck-typing –鸭子类型 指一种编程风格，它并不依靠查找对象类型来确定其是否具有正确的接口，而是直接调用或使用其方法或属性（“看起来像鸭子，叫起来也像鸭子，那么肯定就是鸭子。”）由于强调接口而非特定类型，设计良好的代码可通过允许多态替代来提升灵活性。鸭子类型避免使用*type()*或*isinstance()*检测。(但要注意鸭子类型可以使用抽象基类 作为补充。) 而往往会采用*hasattr()*检测或是*EAFP* 编程。

EAFP “求原谅比求许可更容易”的英文缩写。这种 Python 常用代码编写风格会假定所需的键或属性存在，并在假定错误时捕获异常。这种简洁快速风格的特点就是大量运用 `try` 和 `except` 语句。于其相对的则是所谓*LBYL* 风格，常见于 C 等许多其他语言。

expression –表达式 A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statement*s which cannot be used as expressions, such as `print` or `if`. Assignments are also statements, not expressions.

extension module –扩展模块 以 C 或 C++ 编写的模块，使用 Python 的 C API 来与语言核心以及用户代码进行交互。

file object –文件对象 对外提供面向文件 API 以使用下层资源的对象（带有 `read()` 或 `write()` 这样的方法）。根据其创建方式的不同，文件对象可以处理对真实磁盘文件，对其他类型存储，或是对通讯设备的访问（例如标准输入/输出、内存缓冲区、套接字、管道等等）。文件对象也被称为 *文件类对象* 或 *流*。

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the *io* module. The canonical way to create a file object is by using the *open()* function.

file-like object –文件类对象 *file object* 的同义词。

finder –查找器 An object that tries to find the *loader* for a module. It must implement a method named `find_module()`. See **PEP 302** for details.

floor division –向下取整除法 向下舍入到最接近的整数的数学除法。向下取整除法的运算符是 `//`。例如，表达式 `11 // 4` 的计算结果是 `2`，而与之相反的是浮点数的真正除法返回 `2.75`。注意 `(-11) // 4` 会返回 `-3` 因为这是 `-2.75` 向下舍入得到的结果。见 **PEP 238**。

function –函数 可以向调用者返回某个值的一组语句。还可以向其传入零个或多个参数 并在函数体执行中被使用。另见*parameter*, *method* 和 function 等节。

__future__ A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression `11/4` currently evaluates to `2`. If the module in which it

is executed had enabled *true division* by executing:

```
from __future__ import division
```

the expression `11/4` would evaluate to `2.75`. By importing the *__future__* module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection –**垃圾回收** The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles.

generator –**生成器** A function which returns an iterator. It looks like a normal function except that it contains `yield` statements for producing a series of values usable in a for-loop or that can be retrieved one at a time with the *next()* function. Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression –**生成器表达式** An expression that returns an iterator. It looks like a normal expression followed by a `for` expression defining a loop variable, range, and an optional `if` expression. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

GIL 参见*global interpreter lock*。

global interpreter lock –**全局解释器锁** *CPython* 解释器所采用的一种机制，它确保同一时刻只有一个线程在执行 Python *bytecode*。此机制通过设置对象模型（包括*dict* 等重要内置类型）针对并发访问的隐式安全简化了 CPython 实现。给整个解释器加锁使得解释器多线程运行更方便，其代价则是牺牲了在多处理器上的并行性。

不过，某些标准库或第三方库的扩展模块被设计为在执行计算密集型任务如压缩或哈希时释放 GIL。此外，在执行 I/O 操作时也总是会释放 GIL。

创建一个（以更精细粒度来锁定共享数据的）“自由线程”解释器的努力从未获得成功，因为这会牺牲在普通单处理器情况下的性能。据信克服这种性能问题的措施将导致实现变得更复杂，从而更难以维护。

hashable –**可哈希** An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` or `__cmp__()` method). Hashable objects which compare equal must have the same hash value.

可哈希性使得对象能够作为字典键或集合成员使用，因为这些数据结构要在内部使用哈希值。

All of Python’s immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal (except with themselves), and their hash value is derived from their *id()*.

IDLE Python 的 IDE，“集成开发与学习环境”的英文缩写。是 Python 标准发行版附带的基本编程器和解释器环境。

immutable –**不可变** 具有固定值的对象。不可变对象包括数字、字符串和元组。这样的对象不能被改变。如果必须存储一个不同的值，则必须创建新的对象。它们在需要常量哈希值的地方起着重要作用，例如作为字典中的键。

integer division Mathematical division discarding any remainder. For example, the expression `11/4` currently evaluates to `2` in contrast to the `2.75` returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is

another numeric type (such as a *float*), the result will be coerced (see *coercion*) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also *__future__*.

importing –导入 令一个模块中的 Python 代码能为另一个模块中的 Python 代码所使用的过程。

importer –导入器 查找并加载模块的对象；此对象既属于*finder* 又属于*loader*。

interactive –交互 Python 带有一个交互式解释器，即你可以在解释器提示符后输入语句和表达式，立即执行并查看其结果。只需不带参数地启动 `python` 命令（也可以在你的计算机开始菜单中选择相应菜单项）。在测试新想法或检验模块和包的时候用这种方式会非常方便（请记得使用 `help(x)`）。

interpreted –解释型 Python 一是种解释型语言，与之相对的是编译型语言，虽然两者的区别由于字节码编译器的存在而会有所模糊。这意味着源文件可以直接运行而不必显式地创建可执行文件再运行。解释型语言通常具有比编译型语言更短的开发/调试周期，但是其程序往往运行得更慢。参见*interactive*。

iterable –可迭代对象 An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, *str*, and *tuple*) and some non-sequence types like *dict* and *file* and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed (*zip()*, *map()*, ···). When an iterable object is passed as an argument to the built-in function *iter()*, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call *iter()* or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator –迭代器 An object representing a stream of data. Repeated calls to the iterator's `next()` method return successive items in the stream. When no more data are available a *StopIteration* exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise *StopIteration* again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the *iter()* function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

更多信息可查看迭代器类型。

key function –键函数 键函数或称整理函数，是能够返回用于排序或排位的值的可调用对象。例如，*locale.strxfrm()* 可用于生成一个符合特定区域排序约定的排序键。

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include *min()*, *max()*, *sorted()*, `list.sort()`, *heapq.nsmallest()*, *heapq.nlargest()*, and *itertools.groupby()*.

There are several ways to create a key function. For example. the *str.lower()* method can serve as a key function for case insensitive sorts. Alternatively, an ad-hoc key function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, the *operator* module provides three key function constructors: *attrgetter()*, *itemgetter()*, and *methodcaller()*. See the Sorting HOW TO for examples of how to create and use key functions.

keyword argument –关键字参数 参见*argument*。

lambda 由一个单独*expression* 构成的匿名内联函数，表达式会在调用时被求值。创建 lambda 函数的句法为 `lambda [parameters]: expression`

LBYL "先查看后跳跃"的英文缩写。这种代码编写风格会在进行调用或查找之前显式地检查前提条件。此风格与*EAFP* 方式恰成对比，其特点是大量使用 `if` 语句。

在多线程环境中，LBYL 方式会导致"查看"和"跳跃"之间发生条件竞争风险。例如，以下代码 `if key in mapping: return mapping[key]` 可能由于在检查操作之后其他线程从 *mapping* 中移除了 *key* 而出错。这种问题可通过加锁或使用 EAFP 方式来解决。

list –列表 Python 内置的一种*sequence*。虽然名为列表，但更类似于其他语言中的数组而非链接列表，因为访问元素的时间复杂度为 O(1)。

list comprehension –列表推导式 A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

loader –加载器 An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See **PEP 302** for details.

magic method –魔术方法 *special method* 的非正式同义词。

mapping –映射 A container object that supports arbitrary key lookups and implements the methods specified in the *Mapping* or *MutableMapping abstract base classes*. Examples include *dict*, *collections.defaultdict*, *collections.OrderedDict* and *collections.Counter*.

metaclass –元类 一种用于创建类的类。类定义包含类名、类字典和基类列表。元类负责接受上述三个参数并创建相应的类。大部分面向对象的编程语言都会提供一个默认实现。Python 的特别之处在于可以创建自定义元类。大部分用户永远不需要这个工具，但当需要出现时，元类可提供强大而优雅的解决方案。它们已被用于记录属性访问日志、添加线程安全性、跟踪对象创建、实现单例，以及其他许多任务。

更多详情参见 metaclasses。

method 方法 在类内部定义的函数。如果作为该类的实例的一个属性来调用，方法将会获取实例对象作为其第一个*argument* (通常命名为 `self`)。参见*function* 和*nested scope*。

method resolution order –方法解析顺序 方法解析顺序就是在查找成员时搜索全部基类所用的先后顺序。请查看 Python 2.3 方法解析顺序 了解自 2.3 版起 Python 解析器所用相关算法的详情。

module 模块 此对象是 Python 代码的一种组织单位。各模块具有独立的命名空间，可包含任意 Python 对象。模块可通过*importing* 操作被加载到 Python 中。

另见*package*。

MRO 参见*method resolution order*。

mutable –可变 可变对象可以在其*id()* 保持固定的情况下改变其取值。另请参见*immutable*。

named tuple –具名元组 Any tuple-like class whose indexable elements are also accessible using named attributes (for example, *time.localtime()* returns a tuple-like object where the *year* is accessible either with an index such as `t[0]` or with a named attribute like `t.tm_year`).

A named tuple can be a built-in type such as *time.struct_time*, or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function *collections.namedtuple()*. The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

namespace –命名空间 The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and *os.open()* are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing *random.seed()* or *itertools.izip()* makes it clear that those functions are implemented by the *random* and *itertools* modules, respectively.

nested scope –嵌套作用域 The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

new-style class –新式类 Any class which inherits from *object*. This includes all built-in types like `list` and *dict*. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, and `__getattribute__()`.

More information can be found in newstyle.

object –对象 任何具有状态（属性或值）以及预定义行为（方法）的数据。object 也是任何*new-style class* 的最顶层基类名。

package –包 一种可包含子模块或递归地包含子包的 Python *module*。从技术上说，包是带有 `__path__` 属性的 Python 模块。

parameter –形参 A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are four types of parameters:

- *positional-or-keyword*：位置或关键字，指定一个可以作为位置参数 传入也可以作为关键字参数 传入的实参。这是默认的形参类型，例如下面的 *foo* 和 *bar*:

```
def func(foo, bar=None): ...
```

- *positional-only*：仅限位置，指定一个只能按位置传入的参数。Python 中没有定义仅限位置形参的语法。但是一些内置函数有仅限位置形参（比如*abs()*）。
- *var-positional*：可变位置，指定可以提供由一个任意数量的位置参数构成的序列（附加在其他形参已接受的位置参数之后）。这种形参可通过在形参名称前加缀 `*` 来定义，例如下面的 *args*:

```
def func(*args, **kwargs): ...
```

- *var-keyword*：可变关键字，指定可以提供任意数量的关键字参数（附加在其他形参已接受的关键字参数之后）。这种形参可通过在形参名称前加缀 `**` 来定义，例如上面的 *kwargs*。

形参可以同时指定可选和必选参数，也可以为某些可选参数指定默认值。

See also the *argument* glossary entry, the FAQ question on the difference between arguments and parameters, and the function section.

PEP "Python 增强提议"的英文缩写。一个 PEP 就是一份设计文档，用来向 Python 社区提供信息，或描述一个 Python 的新增特性及其进度或环境。PEP 应当提供精确的技术规格和所提议特性的原理说明。

PEP 应被作为提出主要新特性建议、收集社区对特定问题反馈以及为必须加入 Python 的设计决策编写文档的首选机制。PEP 的作者有责任在社区内部建立共识，并应将不同意见也记入文档。

参见 **PEP 1**。

positional argument –位置参数 参见*argument*。

Python 3000 Python 3.x 发布路线的昵称（这个名字在版本 3 的发布还遥遥无期的时候就已出现了）。有时也被缩写为"Py3k"。

Pythonic 指一个思路或一段代码紧密遵循了 Python 语言最常用的风格和理念，而不是使用其他语言中通用的概念来实现代码。例如，Python 的常用风格是使用 `for` 语句循环来遍历一个可迭代对象中的所有元素。许多其他语言没有这样的结构，因此不熟悉 Python 的人有时会选择使用一个数字计数器:

```
for i in range(len(food)):
    print food[i]
```

而相应的更简洁更 Pythonic 的方法是这样的:

```
for piece in food:
    print piece
```

reference count –引用计数 对特定对象的引用的数量。当一个对象的引用计数降为零时，所分配资源将被释放。引用计数对 Python 代码来说通常是不可见的，但它是*CPython* 实现的一个关键元素。*sys* 模块定义了一个*getrefcount()* 函数，程序员可调用它来返回特定对象的引用计数。

__slots__ A declaration inside a *new-style class* that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence –序列 An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a *len()* method that returns the length of the sequence. Some built-in sequence types are `list`, *str*, *tuple*, and *unicode*. Note that *dict* also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

slice –切片 An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses *slice* objects internally (or in older versions, `__getslice__()` and `__setslice__()`).

special method –特殊方法 一种由 Python 隐式调用的方法，用来对某个类型执行特定操作例如相加等等。这种方法的名称的首尾都为双下划线。特殊方法的文档参见 specialnames。

statement –语句 语句是程序段（一个代码“块”）的组成单位。一条语句可以是一个*expression* 或某个带有关键字的结构，例如 `if`、`while` 或 `for`。

struct sequence A tuple with named elements. Struct sequences expose an interface similiar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like *_make()* or *_asdict()*. Examples of struct sequences include *sys.float_info* and the return value of *os.stat()*.

triple-quoted string –三引号字符串 首尾各带三个连续双引号（"）或者单引号（'）的字符串。它们在功能上与首尾各用一个引号标注的字符串没有什么不同，但是有多种用处。它们允许你在字符串内包含未经转义的单引号和双引号，并且可以跨越多行而无需使用连接符，在编写文档字符串时特别好用。

type –类型 类型决定一个 Python 对象属于什么种类；每个对象都具有一种类型。要知道对象的类型，可以访问它的*__class__* 属性，或是通过 `type(obj)` 来获取。

universal newlines –通用换行 A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See **PEP 278** and **PEP 3116**, as well as *str.splitlines()* for an additional use.

virtual environment –虚拟环境 一种采用协作式隔离的运行时环境，允许 Python 用户和应用程序在安装和升级 Python 分发包时不会干扰到同一系统上运行的其他 Python 应用程序的行为。

virtual machine –虚拟机 一台完全通过软件定义的计算机。Python 虚拟机可执行字节码编译器所生成的*bytecode*。

Zen of Python –Python 之禅 列出 Python 设计的原则与哲学，有助于理解与使用这种语言。查看其具体内容可在交互模式提示符中输入“`import this`”。

APPENDIX B

文档说明

这些文档生成自 reStructuredText 原文档，由 Sphinx （一个专门为 Python 文档写的文档生成器）创建。

本文档和它所用工具链的开发完全是由志愿者完成的，这和 Python 本身一样。如果您想参与进来，请阅读 reporting-bugs 了解如何参与。我们随时欢迎新的志愿者！

特别鸣谢：

- Fred L. Drake, Jr.，创造了用于早期 Python 文档的工具链，以及撰写了非常多的文档；
- Docutils 软件包 项目，创建了 reStructuredText 文本格式和 Docutils 软件套件；
- Fredrik Lundh，Sphinx 从他的 Alternative Python Reference 项目中获得了很多好的想法。

B.1 Python 文档的贡献者

有很多对 Python 语言，Python 标准库和 Python 文档有贡献的人，随 Python 源代码发布的 Misc/ACKS 文件列出了部分贡献者。

有了 Python 社区的输入和贡献，Python 才有了如此出色的文档 - 谢谢你们！

APPENDIX C

历史和许可证

C.1 该软件的历史

Python 由荷兰数学和计算机科学研究学会（CWI，见 https://www.cwi.nl/ ）的 Guido van Rossum 于 1990 年代初设计，作为一门叫做 ABC 的语言的替代品。尽管 Python 包含了许多来自其他人的贡献，Guido 仍是其主要作者。

1995 年，Guido 在弗吉尼亚州的国家创新研究公司（CNRI，见 https://www.cnri.reston.va.us/ ）继续他在 Python 上的工作，并在那里发布了该软件的多个版本。

2000 年五月，Guido 和 Python 核心开发团队转到 BeOpen.com 并组建了 BeOpen PythonLabs 团队。同年十月，PythonLabs 团队转到 Digital Creations (现为 Zope Corporation；见 https://www.zope.org/)。2001 年，Python 软件基金会 (PSF，见 https://www.python.org/psf/) 成立，这是一个专为拥有 Python 相关知识产权而创建的非营利组织。Zope Corporation 现在是 PSF 的赞助成员。

所有的 Python 版本都是开源的（有关开源的定义参阅 https://opensource.org/ ）。历史上，绝大多数 Python 版本是 GPL 兼容的；下表总结了各个版本情况。

发布版本	源自	年份	所有者	GPL 兼容？
0.9.0 至 1.2	n/a	1991-1995	CWI	是
1.3 至 1.5.2	1.2	1995-1999	CNRI	是
1.6	1.5.2	2000	CNRI	否
2.0	1.6	2000	BeOpen.com	否
1.6.1	1.6	2001	CNRI	否
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 及更高	2.1.1	2001 至今	PSF	是

注解： GPL 兼容并不意味着 Python 在 GPL 下发布。与 GPL 不同，所有 Python 许可证都允许您分发修改后

的版本，而无需开源所做的更改。GPL 兼容的许可证使得 Python 可以与其它在 GPL 下发布的软件结合使用；但其它的许可证则不行。

感谢众多在 Guido 指导下工作的外部志愿者，使得这些发布成为可能。

C.2 获取或以其他方式使用 Python 的条款和条件

C.2.1 用于 PYTHON 2.7.18 的 PSF 许可协议

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   2.7.18 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 2.7.18 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2020 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 2.7.18 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 2.7.18 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
   2.7.18.

4. PSF is making Python 2.7.18 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 2.7.18 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.18
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.18, 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 2.7.18, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 用于 PYTHON 2.0 的 BEOPEN.COM 许可协议

BEOPEN PYTHON 开源许可协议第 1 版

```
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
```

(下页继续)

(续上页)

```
   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.
```

C.2.3 用于 PYTHON 1.6.1 的 CNRI 许可协议

```
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
```

(下页继续)

(续上页)

```
   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.
```

C.2.4 用于 PYTHON 0.9.0 至 1.2 的 CWI 许可协议

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.3 被收录软件的许可证与鸣谢

本节是 Python 发行版中收录的第三方软件的许可和致谢清单，该清单是不完整且不断增长的。

C.3.1 Mersenne Twister

_random 模块包含基于 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 下载的代码。以下是原始代码的完整注释（声明）：

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.
```

(下页继续)

(续上页)

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 套接字

socket 模块使用 `getaddrinfo()` 和 `getnameinfo()` 函数，这些函数源代码在 WIDE 项目（http://www.wide.ad.jp/）的单独源文件中。

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
```

(下页继续)

(续上页)

```
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Floating point exception control

The source for the *fpectl* module includes the following notice:

```
  ---------------------------------------------------------------------
 /                       Copyright (c) 1996.                           \
|          The Regents of the University of California.                  |
|                         All rights reserved.                          |
|                                                                       |
|   Permission to use, copy, modify, and distribute this software for   |
|   any purpose without fee is hereby granted, provided that this en-   |
|   tire notice is included in all copies of any software which is or   |
|   includes  a  copy  or  modification  of  this software and in all   |
|   copies of the supporting documentation for such software.           |
|                                                                       |
|   This  work was produced at the University of California, Lawrence   |
|   Livermore National Laboratory under  contract  no.  W-7405-ENG-48   |
|   between  the  U.S.  Department  of  Energy and The Regents of the   |
|   University of California for the operation of UC LLNL.              |
|                                                                       |
|                              DISCLAIMER                               |
|                                                                       |
|   This  software was prepared as an account of work sponsored by an   |
|   agency of the United States Government. Neither the United States   |
|   Government  nor the University of California nor any of their em-   |
|   ployees, makes any warranty, express or implied, or  assumes  any   |
|   liability  or  responsibility  for the accuracy, completeness, or   |
|   usefulness of any information,  apparatus,  product,  or  process   |
|   disclosed,   or  represents  that  its  use  would  not  infringe   |
|   privately-owned rights. Reference herein to any specific  commer-   |
|   cial  products,  process,  or  service  by trade name, trademark,   |
|   manufacturer, or otherwise, does not  necessarily  constitute  or   |
|   imply  its endorsement, recommendation, or favoring by the United   |
|   States Government or the University of California. The views  and   |
|   opinions  of authors expressed herein do not necessarily state or   |
|   reflect those of the United States Government or  the  University   |
|   of  California,  and shall not be used for advertising or product   |
 \  endorsement purposes.                                              /
  ---------------------------------------------------------------------
```

C.3.4 MD5 message digest algorithm

The source code for the *md5* module contains the following notice:

```
Copyright (C) 1999, 2002 Aladdin Enterprises.  All rights reserved.

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
      http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite
(section A.5) but excluding the rest of Appendix A.  It does not include
any code or documentation that is identified in the RFC as being
copyrighted.

The original and principal author of md5.h is L. Peter Deutsch
<ghost@aladdin.com>.  Other authors are noted in the change history
that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed
      references to Ghostscript; clarified derivation from RFC 1321;
      now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
      added conditionalization for C++ compilation from Martin
      Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
```

C.3.5 异步套接字服务

asynchat and *asyncore* 模块包含以下声明:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.6 Cookie 管理

The *Cookie* module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.7 执行追踪

trace 模块包含以下声明:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.8 UUencode 与 UUdecode 函数

uu 模块包含以下声明:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
```

(下页继续)

(续上页)

```
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.9 XML 远程过程调用

The *xmlrpclib* module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.10 test_epoll

The `test_epoll` contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(下页继续)

(续上页)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.11 Select kqueue

The *select* and contains the following notice for the kqueue interface:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.12 strtod and dtoa

`Python/dtoa.c` 文件提供了 C 语言的 dtoa 和 strtod 函数，用于将 C 语言的双精度型和字符串进行转换，该文件由 David M. Gay 的同名文件派生而来，当前可从 http://www.netlib.org/fp/ 下载。2009 年 3 月 16 日检索到的原始文件包含以下版权和许可声明:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
```

(下页继续)

(续上页)

```
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***********************************************************/
```

C.3.13 OpenSSL

如果操作系统可用，则*hashlib*, *posix*, *ssl*, *crypt* 模块使用 OpenSSL 库来提高性能。此外，适用于 Python 的 Windows 和 Mac OS X 安装程序可能包括 OpenSSL 库的拷贝，所以在此处也列出了 OpenSSL 许可证的拷贝:

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   *
   * 1. Redistributions of source code must retain the above copyright
   *    notice, this list of conditions and the following disclaimer.
   *
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in
   *    the documentation and/or other materials provided with the
   *    distribution.
   *
   * 3. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
```

(下页继续)

(续上页)

```
   *    acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
   *
   * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
   * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
   * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
   * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
   * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
   * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
   * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
   * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
   * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
   * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
   * OF THE POSSIBILITY OF SUCH DAMAGE.
   * ====================================================================
   *
   * This product includes cryptographic software written by Eric Young
   * (eay@cryptsoft.com).  This product includes software written by Tim
   * Hudson (tjh@cryptsoft.com).
   *
   */

Original SSLeay License
-----------------------

  /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
   * All rights reserved.
   *
   * This package is an SSL implementation written
   * by Eric Young (eay@cryptsoft.com).
   * The implementation was written so as to conform with Netscapes SSL.
   *
   * This library is free for commercial and non-commercial use as long as
   * the following conditions are aheared to.  The following conditions
   * apply to all code found in this distribution, be it the RC4, RSA,
   * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
   * included with this distribution is covered by the same copyright terms
   * except that the holder is Tim Hudson (tjh@cryptsoft.com).
   *
   * Copyright remains Eric Young's, and as such any Copyright notices in
   * the code are not to be removed.
   * If this package is used in a product, Eric Young should be given attribution
   * as the author of the parts of the library used.
   * This can be in the form of a textual message at program startup or
   * in documentation (online or textual) provided with the package.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   * 1. Redistributions of source code must retain the copyright
   *    notice, this list of conditions and the following disclaimer.
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in the
   *    documentation and/or other materials provided with the distribution.
```

(下页继续)

(续上页)

```
   * 3. All advertising materials mentioning features or use of this software
   *    must display the following acknowledgement:
   *    "This product includes cryptographic software written by
   *     Eric Young (eay@cryptsoft.com)"
   *    The word 'cryptographic' can be left out if the rouines from the library
   *    being used are not cryptographic related :-).
   * 4. If you include any Windows specific code (or a derivative thereof) from
   *    the apps directory (application code) you must include an acknowledgement:
   *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
   *
   * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
   * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
   * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
   * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
   * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
   * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
   * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
   * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
   * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
   * SUCH DAMAGE.
   *
   * The licence and distribution terms for any publically available version or
   * derivative of this code cannot be changed.  i.e. this code cannot simply be
   * copied and put under another distribution licence
   * [including the GNU Public Licence.]
   */
```

C.3.14 expat

除非使用 `--with-system-expat` 配置了构建，否则 `pyexpat` 扩展都是用包含 expat 源的拷贝构建的:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.15 libffi

除非使用 --with-system-libffi 配置了构建，否则 _ctypes 扩展都是包含 libffi 源的拷贝构建的:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.16 zlib

如果系统上找到的 zlib 版本太旧而无法用于构建，则使用包含 zlib 源代码的拷贝来构建*zlib* 扩展:

```
Copyright (C) 1995-2010 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

APPENDIX D

Copyright

Python 与这份文档：

有关完整的许可证和许可信息，参见历史和许可证。

Bibliography

[C99] ISO/IEC 9899:1999. "Programming languages –C." A public draft of this standard is available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

Python 模块索引

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

W

X

Z

索引

非字母

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z