Extending and Embedding Python

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

本文档描述了如何使用 C 或 C++ 编写模块以使用新模块来扩展 Python 解释器的功能。这些模块不仅可以定义新的函数，还可以定义新的对象类型及其方法。该文档还描述了如何将 Python 解释器嵌入到另一个应用程序中，以用作扩展语言。最后，它展示了如何编译和链接扩展模块，以便它们可以动态地（在运行时）加载到解释器中，如果底层操作系统支持此特性的话。

本文档假设你具备有关 Python 的基本知识。有关该语言的非正式介绍，请参阅 tutorial-index 。reference-index 给出了更正式的语言定义。library-index 包含现有的对象类型、函数和模块（内置和用 Python 编写）的文档，使语言具有广泛的应用范围。

关于整个 Python/C API 的详细介绍，请参阅独立的 c-api-index 。

注解： This guide only covers the basic tools for creating extensions provided as part of this version of CPython. Third party tools may offer simpler alternatives. Refer to the binary extensions section in the Python Packaging User Guide for more information.

CHAPTER 1

使用 C 或 C++ 扩展 Python

如果你会用 C，添加新的 Python 内置模块会很简单。以下两件不能用 Python 直接做的事，可以通过 *extension modules* 来实现：实现新的内置对象类型；调用 C 的库函数和系统调用。

为了支持扩展，Python API（应用程序编程接口）定义了一系列函数、宏和变量，可以访问 Python 运行时系统的大部分内容。Python 的 API 可以通过在一个 C 源文件中引用 `"Python.h"` 头文件来使用。

扩展模块的编写方式取决与你的目的以及系统设置；下面章节会详细介绍。

注解： The C extension interface is specific to CPython, and extension modules do not work on other Python implementations. In many cases, it is possible to avoid writing C extensions and preserve portability to other implementations. For example, if your use case is calling C library functions or system calls, you should consider using the `ctypes` module or the cffi library rather than writing custom C code. These modules let you write Python code to interface with C code and are more portable between implementations of Python than writing and compiling a C extension module.

1.1 一个简单的例子

Let’s create an extension module called `spam` (the favorite food of Monty Python fans…) and let’s say we want to create a Python interface to the C library function `system()` [1]. This function takes a null-terminated character string as argument and returns an integer. We want this function to be callable from Python as follows:

```
>>> import spam
>>> status = spam.system("ls -l")
```

首先创建一个 `spammodule.c` 文件。（传统上，如果一个模块叫 `spam`，则对应实现它的 C 文件叫 `spammodule.c`；如果这个模块名字非常长，比如 `spammify`，则这个模块的文件可以直接叫 `spammify.c`。）

The first line of our file can be:

[1] 这个函数的接口已经在标准模块 `os` 里了，这里作为一个简单而直接的例子。

```
#include <Python.h>
```

这会导入 Python API（如果你喜欢，你可以在这里添加描述模块目标和版权信息的注释)。

注解： 由于 Python 可能会定义一些能在某些系统上影响标准头文件的预处理器定义，因此在包含任何标准头文件之前，你 必须先包含 `Python.h`。

所有用户可见的符号都定义自爰 `Python.h` 中，并拥有前缀 `Py` 或 `PY` ，除了那些已经定义在 i 包准头文件的。为了方便，以及为了让 Python 解释器应用广泛，`"Python.h"` 也包含了少量标准头文件：`<stdio.h>` , `<string.h>` , `<errno.h>` , 和 `<stdlib.h>` 。如果后面的头文件在你的系统上不存在，还会直接声明函数 `malloc()` , `free()` 和 `realloc()` 。

下面添加 C 函数到扩展模块，当调用 `spam.system(string)` 时会做出响应，(我们稍后会看到调用)：

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i", sts);
}
```

有个直接翻译参数列表的方法 (举个例子，单独的 `"ls -l"`) 到要传递给 C 函数的参数。C 函数总是有两个参数，通常名字是 *self* 和 *args* 。

For module functions, the *self* argument is *NULL* or a pointer selected while initializing the module (see `Py_InitModule4()`). For a method, it would point to the object instance.

args 参数是指向一个 Python 的 tuple 对象的指针，其中包含参数。每个 tuple 项对应一个调用参数。这些参数也全都是 Python 对象—要在我们的 C 函数中使用它们就需要先将其转换为 C 值。Python API 中的函数 `PyArg_ParseTuple()` 会检查参数类型并将其转换为 C 值。它使用模板字符串确定需要的参数类型以及存储被转换的值的 C 变量类型。细节将稍后说明。

`PyArg_ParseTuple()` returns true (nonzero) if all arguments have the right type and its components have been stored in the variables whose addresses are passed. It returns false (zero) if an invalid argument list was passed. In the latter case it also raises an appropriate exception so the calling function can return *NULL* immediately (as we saw in the example).

1.2 关于错误和异常

An important convention throughout the Python interpreter is the following: when a function fails, it should set an exception condition and return an error value (usually a *NULL* pointer). Exceptions are stored in a static global variable inside the interpreter; if this variable is *NULL* no exception has occurred. A second global variable stores the "associated value" of the exception (the second argument to `raise`). A third variable contains the stack traceback in case the error originated in Python code. These three variables are the C equivalents of the Python variables `sys.exc_type`, `sys.exc_value` and `sys.exc_traceback` (see the section on module `sys` in the Python Library Reference). It is important to know about them to understand how errors are passed around.

Python API 中定义了一些函数来设置这些变量。

最常用的就是 PyErr_SetString()。其参数是异常对象和 C 字符串。异常对象一般是像 PyExc_ZeroDivisionError 这样的预定义对象。C 字符串指明异常原因，并被转换为一个 Python 字符串对象存储为异常的 "关联值"。

另一个有用的函数是 PyErr_SetFromErrno() ，仅接受一个异常对象，异常描述包含在全局变量 errno 中。最通用的函数还是 PyErr_SetObject() ，包含两个参数，分别为异常对象和异常描述。你不需要使用 Py_INCREF() 来增加传递到其他函数的参数对象的引用计数。

You can test non-destructively whether an exception has been set with PyErr_Occurred(). This returns the current exception object, or *NULL* if no exception has occurred. You normally don't need to call PyErr_Occurred() to see whether an error occurred in a function call, since you should be able to tell from the return value.

When a function *f* that calls another function *g* detects that the latter fails, *f* should itself return an error value (usually *NULL* or -1). It should *not* call one of the PyErr_*() functions —one has already been called by *g*. *f*'s caller is then supposed to also return an error indication to *its* caller, again *without* calling PyErr_*(), and so on —the most detailed cause of the error was already reported by the function that first detected it. Once the error reaches the Python interpreter's main loop, this aborts the currently executing Python code and tries to find an exception handler specified by the Python programmer.

（在某些情况下，当模块确实能够通过调用其它 PyErr_*() 函数给出更加详细的错误消息，并且在这些情况是可以这样做的。但是按照一般规则，这是不必要的，并可能导致有关错误原因的信息丢失：大多数操作会由于种种原因而失败。）

想要忽略由一个失败的函数调用所设置的异常，异常条件必须通过调用 PyErr_Clear() 显式地被清除。C 代码应当调用 PyErr_Clear() 的唯一情况是如果它不想将错误传给解释器而是想完全由自己来处理它（可能是尝试其他方法，或是假装没有出错）。

Every failing malloc() call must be turned into an exception —the direct caller of malloc() (or realloc()) must call PyErr_NoMemory() and return a failure indicator itself. All the object-creating functions (for example, PyInt_FromLong()) already do this, so this note is only relevant to those who call malloc() directly.

还要注意的是，除了 PyArg_ParseTuple() 等重要的例外，返回整数状态码的函数通常都是返回正值或零来表示成功，而以 -1 表示失败，如同 Unix 系统调用一样。

最后，当你返回一个错误指示器时要注意清理垃圾（通过为你已经创建的对象执行 Py_XDECREF() 或 Py_DECREF() 调用）！

选择引发哪个异常完全取决于你的喜好。所有内置的 Python 异常都有对应的预声明 C 对象，例如 PyExc_ZeroDivisionError，你可以直接使用它们。当然，你应当明智地选择异常—不要使用 PyExc_TypeError 来表示一个文件无法被打开 (那大概应该用 PyExc_IOError)。如果参数列表有问题，PyArg_ParseTuple() 函数通常会引发 PyExc_TypeError。如果你想要一个参数的值必须处于特定范围之内或必须满足其他条件，则适宜使用 PyExc_ValueError。

你也可以为你的模块定义一个唯一的新异常。需要在文件前部声明一个静态对象变量，如:

```
static PyObject *SpamError;
```

and initialize it in your module's initialization function (initspam()) with an exception object (leaving out the error checking for now):

```
PyMODINIT_FUNC
initspam(void)
{
    PyObject *m;

    m = Py_InitModule("spam", SpamMethods);
    if (m == NULL)
        return;
```

(下页继续)

(续上页)

```
    SpamError = PyErr_NewException("spam.error", NULL, NULL);
    Py_INCREF(SpamError);
    PyModule_AddObject(m, "error", SpamError);
}
```

Note that the Python name for the exception object is `spam.error`. The `PyErr_NewException()` function may create a class with the base class being `Exception` (unless another class is passed in instead of *NULL*), described in bltin-exceptions.

同样注意的是创建类保存了 `SpamError` 的一个引用，这是有意的。为了防止被垃圾回收掉，否则 `SpamError` 随时会成为野指针。

一会讨论 `PyMODINIT_FUNC` 作为函数返回类型的用法。

`spam.error` 异常可以在扩展模块中抛出，通过 `PyErr_SetString()` 函数调用，如下：

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    if (sts < 0) {
        PyErr_SetString(SpamError, "System command failed");
        return NULL;
    }
    return PyLong_FromLong(sts);
}
```

1.3 回到例子

回到前面的例子，你应该明白下面的代码:

```
if (!PyArg_ParseTuple(args, "s", &command))
    return NULL;
```

It returns *NULL* (the error indicator for functions returning object pointers) if an error is detected in the argument list, relying on the exception set by `PyArg_ParseTuple()`. Otherwise the string value of the argument has been copied to the local variable `command`. This is a pointer assignment and you are not supposed to modify the string to which it points (so in Standard C, the variable `command` should properly be declared as `const char *command`).

下一个语句使用 UNIX 系统函数 `system()` ，传递给他的参数是刚才从 `PyArg_ParseTuple()` 取出的:

```
sts = system(command);
```

Our `spam.system()` function must return the value of `sts` as a Python object. This is done using the function `Py_BuildValue()`, which is something like the inverse of `PyArg_ParseTuple()`: it takes a format string and an arbitrary number of C values, and returns a new Python object. More info on `Py_BuildValue()` is given later.

```
return Py_BuildValue("i", sts);
```

在这种情况下，会返回一个整数对象，(这个对象会在 Python 堆里面管理)。

如果你的 C 函数没有有用的返回值 (返回 `void` 的函数)，则必须返回 `None` 。(你可以用 `Py_RETUN_NONE` 宏来完成):

```
Py_INCREF(Py_None);
return Py_None;
```

`Py_None` is the C name for the special Python object `None`. It is a genuine Python object rather than a *NULL* pointer, which means "error" in most contexts, as we have seen.

1.4 模块方法表和初始化函数

为了展示 `spam_system()` 如何被 Python 程序调用。把函数声明为可以被 Python 调用，需要先定义一个方法表 "method table"。

```
static PyMethodDef SpamMethods[] = {
    ...
    {"system",  spam_system, METH_VARARGS,
     "Execute a shell command."},
    ...
    {NULL, NULL, 0, NULL}        /* Sentinel */
};
```

注意第三个参数 (`METH_VARARGS`) ，这个标志指定会使用 C 的调用惯例。可选值有 `METH_VARARGS` 、`METH_VARARGS | METH_KEYWORDS` 。值 0 代表使用 `PyArg_ParseTuple()` 的陈旧变量。

如果单独使用 `METH_VARARGS` ，函数会等待 Python 传来 tuple 格式的参数，并最终使用 `PyArg_ParseTuple()` 进行解析。

`METH_KEYWORDS` 值表示接受关键字参数。这种情况下 C 函数需要接受第三个 `PyObject *` 对象，表示字典参数，使用 `PyArg_ParseTupleAndKeywords()` 来解析出参数。

The method table must be passed to the interpreter in the module's initialization function. The initialization function must be named `initname()`, where *name* is the name of the module, and should be the only non-`static` item defined in the module file:

```
PyMODINIT_FUNC
initspam(void)
{
    (void) Py_InitModule("spam", SpamMethods);
}
```

Note that PyMODINIT_FUNC declares the function as `void` return type, declares any special linkage declarations required by the platform, and for C++ declares the function as `extern "C"`.

When the Python program imports module `spam` for the first time, `initspam()` is called. (See below for comments about embedding Python.) It calls `Py_InitModule()`, which creates a "module object" (which is inserted in the dictionary `sys.modules` under the key `"spam"`), and inserts built-in function objects into the newly created module based upon the table (an array of `PyMethodDef` structures) that was passed as its second argument. `Py_InitModule()` returns a pointer to the module object that it creates (which is unused here). It may abort with a fatal error for certain errors, or return *NULL* if the module could not be initialized satisfactorily.

When embedding Python, the `initspam()` function is not called automatically unless there's an entry in the `_PyImport_Inittab` table. The easiest way to handle this is to statically initialize your statically-linked modules by directly calling `initspam()` after the call to `Py_Initialize()`:

```
int
main(int argc, char *argv[])
{
    /* Pass argv[0] to the Python interpreter */
    Py_SetProgramName(argv[0]);

    /* Initialize the Python interpreter.  Required. */
    Py_Initialize();

    /* Add a static module */
    initspam();

    ...
```

An example may be found in the file `Demo/embed/demo.c` in the Python source distribution.

注解: Removing entries from `sys.modules` or importing compiled modules into multiple interpreters within a process (or following a `fork()` without an intervening `exec()`) can create problems for some extension modules. Extension module authors should exercise caution when initializing internal data structures. Note also that the `reload()` function can be used with extension modules, and will call the module initialization function (`initspam()` in the example), but will not load the module again if it was loaded from a dynamically loadable object file (`.so` on Unix, `.dll` on Windows).

更多关于模块的现实的例子包含在 Python 源码包的 `Modules/xxmodule.c` 中。这些文件可以用作你的代码模板，或者学习。脚本 modulator.py 包含在源码发行版或 Windows 安装中，提供了一个简单的 GUI，用来声明需要实现的函数和对象，并且可以生成供填入的模板。脚本在 Tools/modulator/ 目录。查看 README 以了解用法。

1.5 编译和链接

在你能使用你的新写的扩展之前，你还需要做两件事情：使用 Python 系统来编译和链接。如果你使用动态加载，这取决于你使用的操作系统的动态加载机制；更多信息请参考编译扩展模块的章节（*使用 distutils 构建 C 和 C++ 扩展* 章节），以及在 Windows 上编译需要的额外信息（*在 Windows 平台编译 C 和 C++ 扩展* 章节）。

If you can't use dynamic loading, or if you want to make your module a permanent part of the Python interpreter, you will have to change the configuration setup and rebuild the interpreter. Luckily, this is very simple on Unix: just place your file (`spammodule.c` for example) in the `Modules/` directory of an unpacked source distribution, add a line to the file `Modules/Setup.local` describing your file:

```
spam spammodule.o
```

然后在顶层目录运行 **make** 来重新构建解释器。你也可以在 `Modules/` 子目录使用 **make**，但是你必须先重建 `Makefile` 文件，然后运行 '**make** Makefile' 命令。（你每次修改 `Setup` 文件都需要这样操作。）

If your module requires additional libraries to link with, these can be listed on the line in the configuration file as well, for instance:

```
spam spammodule.o -lX11
```

1.6 在 C 中调用 Python 函数

迄今为止，我们一直把注意力集中于让 Python 调用 C 函数，其实反过来也很有用，就是用 C 调用 Python 函数。这在回调函数中尤其有用。如果一个 C 接口使用回调，那么就要实现这个回调机制。

幸运的是，Python 解释器是比较方便回调的，并给标准 Python 函数提供了标准接口。(这里就不再详述解析 Python 字符串作为输入的方式，如果有兴趣可以参考 `Python/pythonmain.c` 中的 `-c` 命令行代码。)

调用 Python 函数很簡单，首先 Python 程序要传递 Python 函数对象。应该提供个函数 (或其他接口) 来实现。当调用这个函数时，用全局变量保存 Python 函数对象的指针，还要调用 (`Py_INCREF()`) 来增加引用计数，当然不用全局变量也没什么关系。举个例子，如下函数可能是模块定义的一部分：

```
static PyObject *my_callback = NULL;

static PyObject *
my_set_callback(PyObject *dummy, PyObject *args)
{
    PyObject *result = NULL;
    PyObject *temp;

    if (PyArg_ParseTuple(args, "O:set_callback", &temp)) {
        if (!PyCallable_Check(temp)) {
            PyErr_SetString(PyExc_TypeError, "parameter must be callable");
            return NULL;
        }
        Py_XINCREF(temp);         /* Add a reference to new callback */
        Py_XDECREF(my_callback);  /* Dispose of previous callback */
        my_callback = temp;       /* Remember new callback */
        /* Boilerplate to return "None" */
        Py_INCREF(Py_None);
        result = Py_None;
    }
    return result;
}
```

这个函数必须使用 `METH_VARARGS` 标志注册到解释器，这在模块方法表和初始化函数 章节会描述。`PyArg_ParseTuple()` 函数及其参数的文档在提取扩展函数的参数 。

The macros `Py_XINCREF()` and `Py_XDECREF()` increment/decrement the reference count of an object and are safe in the presence of *NULL* pointers (but note that *temp* will not be *NULL* in this context). More info on them in section 引用计数.

Later, when it is time to call the function, you call the C function `PyObject_CallObject()`. This function has two arguments, both pointers to arbitrary Python objects: the Python function, and the argument list. The argument list must always be a tuple object, whose length is the number of arguments. To call the Python function with no arguments, pass in NULL, or an empty tuple; to call it with one argument, pass a singleton tuple. `Py_BuildValue()` returns a tuple when its format string consists of zero or more format codes between parentheses. For example:

```
int arg;
PyObject *arglist;
PyObject *result;
...
arg = 123;
...
/* Time to call the callback */
arglist = Py_BuildValue("(i)", arg);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
```

PyObject_CallObject() 返回 Python 对象指针，这也是 Python 函数的返回值。PyObject_CallObject() 是一个对其参数"引用计数无关"的函数。例子中新的元组创建用于参数列表，并且在 PyObject_CallObject() 之后立即使用了 Py_DECREF() 。

PyEval_CallObject() 的返回值总是"新"的：要么是一个新建的对象；要么是已有对象，但增加了引用计数。所以除非你想把结果保存在全局变量中，你需要对这个值使用 Py_DECREF()，即使你对里面的内容（特别!）不感兴趣。

Before you do this, however, it is important to check that the return value isn't *NULL*. If it is, the Python function terminated by raising an exception. If the C code that called PyObject_CallObject() is called from Python, it should now return an error indication to its Python caller, so the interpreter can print a stack trace, or the calling Python code can handle the exception. If this is not possible or desirable, the exception should be cleared by calling PyErr_Clear(). For example:

```
if (result == NULL)
    return NULL; /* Pass error back */
...use result...
Py_DECREF(result);
```

依赖于具体的回调函数，你还要提供一个参数列表到 PyEval_CallObject() 。在某些情况下参数列表是由 Python 程序提供的，通过接口再传到回调函数对象。这样就可以不改变形式直接传递。另外一些时候你要构造一个新的元组来传递参数。最简单的方法就是 Py_BuildValue() 函数构造 tuple。举个例子，你要传递一个事件代码时可以用如下代码:

```
PyObject *arglist;
...
arglist = Py_BuildValue("(l)", eventcode);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

注意 Py_DECREF(arglist) 所在处会立即调用，在错误检查之前。当然还要注意一些常规的错误，比如 Py_BuildValue() 可能会遭遇内存不足等等。

当你调用函数时还需要注意，用关键字参数调用 PyObject_Call() ，需要支持普通参数和关键字参数。有如如上例子中，我们使用 Py_BuildValue() 来构造字典。

```
PyObject *dict;
...
dict = Py_BuildValue("{s:i}", "name", val);
result = PyObject_Call(my_callback, NULL, dict);
Py_DECREF(dict);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

1.7 提取扩展函数的参数

函数 PyArg_ParseTuple() 的声明如下：

```
int PyArg_ParseTuple(PyObject *arg, char *format, ...);
```

参数 *arg* 必须是一个元组对象，包含从 Python 传递给 C 函数的参数列表。*format* 参数必须是一个格式字符串，语法请参考 Python C/API 手册中的 arg-parsing。剩余参数是各个变量的地址，类型要与格式字符串对应。

注意 PyArg_ParseTuple() 会检测他需要的 Python 参数类型，却无法检测传递给他的 C 变量地址，如果这里出错了，可能会在内存中随机写入东西，小心。

注意任何由调用者提供的 Python 对象引用是 借来的引用；不要递减它们的引用计数！

一些调用的例子：

```
int ok;
int i, j;
long k, l;
const char *s;
int size;

ok = PyArg_ParseTuple(args, ""); /* No arguments */
    /* Python call: f() */
```

```
ok = PyArg_ParseTuple(args, "s", &s); /* A string */
    /* Possible Python call: f('whoops!') */
```

```
ok = PyArg_ParseTuple(args, "lls", &k, &l, &s); /* Two longs and a string */
    /* Possible Python call: f(1, 2, 'three') */
```

```
ok = PyArg_ParseTuple(args, "(ii)s#", &i, &j, &s, &size);
    /* A pair of ints and a string, whose size is also returned */
    /* Possible Python call: f((1, 2), 'three') */
```

```
{
    const char *file;
    const char *mode = "r";
    int bufsize = 0;
    ok = PyArg_ParseTuple(args, "s|si", &file, &mode, &bufsize);
    /* A string, and optionally another string and an integer */
    /* Possible Python calls:
       f('spam')
       f('spam', 'w')
       f('spam', 'wb', 100000) */
}
```

```
{
    int left, top, right, bottom, h, v;
    ok = PyArg_ParseTuple(args, "((ii)(ii))(ii)",
             &left, &top, &right, &bottom, &h, &v);
    /* A rectangle and a point */
    /* Possible Python call:
       f(((0, 0), (400, 300)), (10, 10)) */
}
```

```
{
    Py_complex c;
    ok = PyArg_ParseTuple(args, "D:myfunction", &c);
    /* a complex, also providing a function name for errors */
    /* Possible Python call: myfunction(1+2j) */
}
```

1.8 给扩展函数的关键字参数

函数 `PyArg_ParseTupleAndKeywords()` 声明如下：

```
int PyArg_ParseTupleAndKeywords(PyObject *arg, PyObject *kwdict,
                                char *format, char *kwlist[], ...);
```

The *arg* and *format* parameters are identical to those of the `PyArg_ParseTuple()` function. The *kwdict* parameter is the dictionary of keywords received as the third parameter from the Python runtime. The *kwlist* parameter is a *NULL*-terminated list of strings which identify the parameters; the names are matched with the type information from *format* from left to right. On success, `PyArg_ParseTupleAndKeywords()` returns true, otherwise it returns false and raises an appropriate exception.

注解： 嵌套的元组在使用关键字参数时无法生效，不在 *kwlist* 中的关键字参数会导致 `TypeError` 异常。

如下例子是使用关键字参数的例子模块，作者是 Geoff Philbrick (philbrick@hks.com):

```
#include "Python.h"

static PyObject *
keywdarg_parrot(PyObject *self, PyObject *args, PyObject *keywds)
{
    int voltage;
    char *state = "a stiff";
    char *action = "voom";
    char *type = "Norwegian Blue";

    static char *kwlist[] = {"voltage", "state", "action", "type", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, keywds, "i|sss", kwlist,
                                     &voltage, &state, &action, &type))
        return NULL;

    printf("-- This parrot wouldn't %s if you put %i Volts through it.\n",
           action, voltage);
    printf("-- Lovely plumage, the %s -- It's %s!\n", type, state);

    Py_INCREF(Py_None);

    return Py_None;
}

static PyMethodDef keywdarg_methods[] = {
    /* The cast of the function is necessary since PyCFunction values
     * only take two PyObject* parameters, and keywdarg_parrot() takes
     * three.
```

(下页继续)

(续上页)

```
     */
    {"parrot", (PyCFunction)keywdarg_parrot, METH_VARARGS | METH_KEYWORDS,
     "Print a lovely skit to standard output."},
    {NULL, NULL, 0, NULL}   /* sentinel */
};
```

```
void
initkeywdarg(void)
{
  /* Create the module and add the functions */
  Py_InitModule("keywdarg", keywdarg_methods);
}
```

1.9 构造任意值

这个函数与 `PyArg_ParseTuple()` 很相似，声明如下：

```
PyObject *Py_BuildValue(char *format, ...);
```

接受一个格式字符串，与 `PyArg_ParseTuple()` 相同，但是参数必须是原变量的地址指针 (输入给函数，而非输出)。最终返回一个 Python 对象适合于返回 C 函数调用给 Python 代码。

一个与 `PyArg_ParseTuple()` 的不同是，后面可能需要的要求返回一个元组 (Python 参数里诶包总是在内部描述为元组)，比如用于传递给其他 Python 函数以参数。`Py_BuildValue()` 并不总是生成元组，在多于 1 个格式字符串时会生成元组，而如果格式字符串为空则返回 `None` ，一个参数则直接返回该参数的对象。如果要求强制生成一个长度为 0 的元组，或包含一个元素的元组，需要在格式字符串中加上括号。

例子 (左侧是调用，右侧是 Python 值结果)：

```
Py_BuildValue("")                        None
Py_BuildValue("i", 123)                  123
Py_BuildValue("iii", 123, 456, 789)      (123, 456, 789)
Py_BuildValue("s", "hello")              'hello'
Py_BuildValue("ss", "hello", "world")    ('hello', 'world')
Py_BuildValue("s#", "hello", 4)          'hell'
Py_BuildValue("()")                      ()
Py_BuildValue("(i)", 123)                (123,)
Py_BuildValue("(ii)", 123, 456)          (123, 456)
Py_BuildValue("(i,i)", 123, 456)         (123, 456)
Py_BuildValue("[i,i]", 123, 456)         [123, 456]
Py_BuildValue("{s:i,s:i}",
              "abc", 123, "def", 456)    {'abc': 123, 'def': 456}
Py_BuildValue("((ii)(ii)) (ii)",
              1, 2, 3, 4, 5, 6)          (((1, 2), (3, 4)), (5, 6))
```

1.10 引用计数

在 C/C++ 语言中，程序员负责动态分配和回收堆 heap 当中的内存。在 C 里，通过函数 `malloc()` 和 `free()` 来完成。在 C++ 里是操作 `new` 和 `delete` 来实现相同的功能。

每个由 `malloc()` 分配的内存块，最终都要由 `free()` 退回到可用内存池里面去。而调用 `free()` 的时机非常重要，如果一个内存块忘了 `free()` 则会导致内存泄漏，这块内存在程序结束前将无法重新使用。这叫做 *内存泄漏*。而如果对同一内存块 `free()` 了以后，另外一个指针再次访问，则再次使用 `malloc()` 复用这块内存会导致冲突。这叫做 *野指针*。等同于使用未初始化的数据，core dump，错误结果，神秘的崩溃等。

内存泄露往往发生在一些并不常见的代码流程上面。比如一个函数申请了内存以后，做了些计算，然后释放内存块。现在一些对函数的修改可能增加对计算的测试并检测错误条件，然后过早的从函数返回了。这很容易忘记在退出前释放内存，特别是后期修改的代码。这种内存泄漏，一旦引入，通常很长时间都难以检测到，错误退出被调用的频度较低，而现代电脑又有非常巨大的虚拟内存，所以泄漏仅在长期运行或频繁调用泄漏函数时才会变得明显。因此，有必要避免内存泄漏，通过代码规范会策略来最小化此类错误。

Python 通过 `malloc()` 和 `free()` 包含大量的内存分配和释放，同样需要避免内存泄漏和野指针。他选择的方法就是 *引用计数*。其原理比较简单：每个对象都包含一个计数器，计数器的增减与对象引用的增减直接相关，当引用计数为 0 时，表示对象已经没有存在的意义了，对象就可以删除了。

另一个叫法是 *自动垃圾回收*。(有时引用计数也被看作是垃圾回收策略，于是这里的"自动"用以区分两者)。自动垃圾回收的优点是用户不需要明确的调用 `free()`。(另一个优点是改善速度或内存使用，然而这并不难)。缺点是对 C，没有可移植的自动垃圾回收器，而引用计数则可以可移植的实现 (只要 `malloc()` 和 `free()` 函数是可用的，这也是 C 标准担保的)。也许以后有一天会出现可移植的自动垃圾回收器，但在此前我们必须与引用计数一起工作。

Python 使用传统的引用计数实现，也提供了循环监测器，用以检测引用循环。这使得应用无需担心直接或间接的创建了循环引用，这是引用计数垃圾收集的一个弱点。引用循环是对象 (可能直接) 的引用了本身，所以循环中的每个对象的引用计数都不是 0。典型的引用计数实现无法回收处于引用循环中的对象，或者被循环所引用的对象，哪怕没有循环以外的引用了。

The cycle detector is able to detect garbage cycles and can reclaim them so long as there are no finalizers implemented in Python (`__del__()` methods). When there are such finalizers, the detector exposes the cycles through the `gc` module (specifically, the `garbage` variable in that module). The `gc` module also exposes a way to run the detector (the `collect()` function), as well as configuration interfaces and the ability to disable the detector at runtime. The cycle detector is considered an optional component; though it is included by default, it can be disabled at build time using the `--without-cycle-gc` option to the **`configure`** script on Unix platforms (including Mac OS X) or by removing the definition of `WITH_CYCLE_GC` in the `pyconfig.h` header on other platforms. If the cycle detector is disabled in this way, the `gc` module will not be available.

1.10.1 Python 中的引用计数

有两个宏 `Py_INCREF(x)` 和 `Py_DECREF(x)`，会处理引用计数的增减。`Py_DECREF()` 也会在引用计数到达 0 时释放对象。为了灵活，并不会直接调用 `free()`，而是通过对象的 *类型对象*的函数指针来调用。为了这个目的 (或其他的)，每个对象同时包含一个指向自身类型对象的指针。

最大的问题依旧：何时使用 `Py_INCREF(x)` 和 `Py_DECREF(x)`？我们首先引入一些概念。没有人"拥有"一个对象，你可以 *拥有一个引用*到一个对象。一个对象的引用计数定义为拥有引用的数量。引用的拥有者有责任调用 `Py_DECREF()`，在引用不再需要时。引用的拥有关系可以被传递。有三种办法来处置拥有的引用：传递、存储、调用 `Py_DECREF()`。忘记处置一个拥有的引用会导致内存泄漏。

还可以 *借用*[2] 一个对象的引用。借用的引用不应该调用 `Py_DECREF()`。借用者必须确保不能持有对象超过拥有者借出的时间。在拥有者处置对象后使用借用的引用是有风险的，应该完全避免[3]。

[2] 术语"借用"一个引用是不完全正确的：拥有者仍然有引用的拷贝。
[3] 检查引用计数至少为 1 **没有用**，引用计数本身可以在已经释放的内存里，并有可能被其他对象所用。

借用相对于引用的优点是你无需担心整条路径上代码的引用，或者说，通过借用你无需担心内存泄漏的风险。借用的缺点是一些看起来正确代码上的借用可能会在拥有者处置后使用对象。

借用可以变为拥有引用，通过调用 `Py_INCREF()` 。这不会影响已经借出的拥有者的状态。这回创建一个新的拥有引用，并给予完全的拥有者责任 (新的拥有者必须恰当的处置引用，就像之前的拥有者那样)。

1.10.2 拥有规则

当一个对象引用传递进出一个函数时，函数的接口应该指定拥有关系的传递是否包含引用。

Most functions that return a reference to an object pass on ownership with the reference. In particular, all functions whose function it is to create a new object, such as `PyInt_FromLong()` and `Py_BuildValue()`, pass ownership to the receiver. Even if the object is not actually new, you still receive ownership of a new reference to that object. For instance, `PyInt_FromLong()` maintains a cache of popular values and can return a reference to a cached item.

很多另一个对象提取对象的函数，也会传递引用关系，例如 `PyObject_GetAttrString()` 。这里的情况不够清晰，一些不太常用的例程是例外的 `PyTuple_GetItem()` ，`PyList_GetItem()` ，`PyDict_GetItem()` ，`PyDict_GetItemString()` 都是返回从元组、列表、字典里借用的引用。

函数 `PyImport_AddModule()` 也会返回借用的引用，哪怕可能会返回创建的对象：这个可能因为一个拥有的引用对象是存储在 `sys.modules` 里。

当你传递一个对象引用到另一个函数时，通常函数是借用出去的。如果需要存储，就使用 `Py_INCREF()` 来变成独立的拥有者。这个规则有两个重要的例外：`PyTuple_SetItem()` 和 `PyList_SetItem()` 。这些函数接受传递来的引用关系，哪怕会失败！(注意 `PyDict_SetItem()` 及其同类不会接受引用关系，他们是”正常的”)。

当一个 C 函数被 Python 调用时，会从调用方传来的参数借用引用。调用者拥有对象的引用，所以借用的引用生命周期可以保证到函数返回。只要当借用的引用需要存储或传递时，就必须转换为拥有的引用，通过调用 `Py_INCREF()` 。

Python 调用从 C 函数返回的对象引用时必须是拥有的引用—拥有关系被从函数传递给调用者。

1.10.3 危险的薄冰

有少数情况下，借用的引用看起来无害，但却可能导致问题。这通常是因为解释器的隐式调用，并可能导致引用拥有者处置这个引用。

首先需要特别注意的情况是使用 `Py_DECREF()` 到一个无关对象，而这个对象的引用是借用自一个列表的元素。举个实例：

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    PyList_SetItem(list, 1, PyInt_FromLong(0L));
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

这个函数首先借用一个引用 `list[0]` ，然后替换 `list[1]` 为值 `0` ，最后打印借用的引用。看起来无害是吧，但却不是。

我们跟着控制流进入 `PyList_SetItem()` 。列表拥有者引用了其所有成员，所以当成员 1 被替换时，就必须处置原来的成员 1。现在假设原来的成员 1 是用户定义类的实例，且假设这个类定义了 `__del__()` 方法。如果这个类实例的引用计数是 1，那么处置动作就会调用 `__del__()` 方法。

既然是 Python 写的，`__del__()` 方法可以执行任意 Python 代码。是否可能在 `bug()` 的 `item` 废止引用呢，是的。假设列表传递到 `bug()` 会被 `__del__()` 方法所访问，就可以执行一个语句来实现 `del list[0]`，然后假设这是最后一个对对象的引用，就需要释放内存，从而使得 `item` 无效化。

解决方法是，当你知道了问题的根源，就容易了：临时增加引用计数。正确版本的函数代码如下：

```
void
no_bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    Py_INCREF(item);
    PyList_SetItem(list, 1, PyInt_FromLong(0L));
    PyObject_Print(item, stdout, 0);
    Py_DECREF(item);
}
```

这是个真实的故事。一个旧版本的 Python 包含了这个 bug 的变种，而一些人花费了大量时间在 C 调试器上去寻找为什么 `__del__()` 方法会失败。

这个问题的第二种情况是借用的引用涉及线程的变种。通常，Python 解释器里多个线程无法进入对方的路径，因为有个全局锁保护着 Python 整个对象空间。但可以使用宏 `Py_BEGIN_ALLOW_THREADS` 来临时释放这个锁，重新获取锁用 `Py_END_ALLOW_THREADS`。这通常围绕在阻塞 I/O 调用外，使得其他线程可以在等待 I/O 期间使用处理器。显然，如下函数会跟之前那个有一样的问题：

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);
    Py_BEGIN_ALLOW_THREADS
    ...some blocking I/O call...
    Py_END_ALLOW_THREADS
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

1.10.4 NULL 指针

In general, functions that take object references as arguments do not expect you to pass them *NULL* pointers, and will dump core (or cause later core dumps) if you do so. Functions that return object references generally return *NULL* only to indicate that an exception occurred. The reason for not testing for *NULL* arguments is that functions often pass the objects they receive on to other function —if each function were to test for *NULL*, there would be a lot of redundant tests and the code would run more slowly.

It is better to test for *NULL* only at the "source:" when a pointer that may be *NULL* is received, for example, from `malloc()` or from a function that may raise an exception.

The macros `Py_INCREF()` and `Py_DECREF()` do not check for *NULL* pointers —however, their variants `Py_XINCREF()` and `Py_XDECREF()` do.

The macros for checking for a particular object type (`Pytype_Check()`) don't check for *NULL* pointers —again, there is much code that calls several of these in a row to test an object against various different expected types, and this would generate redundant tests. There are no variants with *NULL* checking.

The C function calling mechanism guarantees that the argument list passed to C functions (`args` in the examples) is never *NULL* —in fact it guarantees that it is always a tuple[4].

It is a severe error to ever let a *NULL* pointer "escape" to the Python user.

[4] 当你使用"旧式"风格调用约定时，这些保证不成立，尽管这依旧存在于很多旧代码中。

1.11 在 C++ 中编写扩展

还可以在 C++ 中编写扩展模块，只是有些限制。如果主程序 (Python 解释器) 是使用 C 编译器来编译和链接的，全局或静态对象的构造器就不能使用。而如果是 C++ 编译器来链接的就没有这个问题。函数会被 Python 解释器调用 (通常就是模块初始化函数) 必须声明为 `extern "C"`。而是否在 `extern "C" {...}` 里包含 Python 头文件则不是那么重要，因为如果定义了符号 `__cplusplus` 则已经是这么声明的了 (所有现代 C++ 编译器都会定义这个符号)。

1.12 给扩展模块提供 C API

很多扩展模块提供了新的函数和类型供 Python 使用，但有时扩展模块里的代码也可以被其他扩展模块使用。例如，一个扩展模块可以实现一个类型"collection"看起来是没有顺序的。就像是 Python 列表类型，拥有 C API 允许扩展模块来创建和维护列表，这个新的集合类型可以有一堆 C 函数用于给其他扩展模块直接使用。

开始看起来很简单：只需要编写函数 (无需声明为 `static`)，提供恰当的头文件，以及 C API 的文档。实际上在所有扩展模块都是静态链接到 Python 解释器时也是可以正常工作的。当模块以共享库链接时，一个模块中的符号定义对另一个模块不可见。可见的细节依赖于操作系统，一些系统的 Python 解释器使用全局命名空间 (例如 Windows)，有些则在链接时需要一个严格的已导入符号列表 (一个例子是 AIX)，或者提供可选的不同策略 (如 Unix 系列)。即便是符号是全局可见的，你要调用的模块也可能尚未加载。

可移植性需要不能对符号可见性做任何假设。这意味着扩展模块里的所有符号都应该声明为 `static`，除了模块的初始化函数，来避免与其他扩展模块的命名冲突 (在段落模块方法表和初始化函数 中讨论)。这意味着符号应该 *必须*通过其他导出方式来供其他扩展模块访问。

Python 提供了一个特别的机制来传递 C 级别信息 (指针)，从一个扩展模块到另一个：Capsules。一个 Capsule 是一个 Python 数据类型，会保存指针 (`void *`)。Capsule 只能通过其 C API 来创建和访问，但可以像其他 Python 对象一样的传递。通常，我们可以指定一个扩展模块命名空间的名字。其他扩展模块可以导入这个模块，获取这个名字的值，然后从 Capsule 获取指针。

Capsule 可以用多种方式导出 C API 给扩展模块。每个函数可以用自己的 Capsule，或者所有 C API 指针可以存储在一个数组里，数组地址再发布给 Capsule。存储和获取指针也可以用多种方式，供客户端模块使用。

Whichever method you choose, it's important to name your Capsules properly. The function `PyCapsule_New()` takes a name parameter (`const char *`); you're permitted to pass in a *NULL* name, but we strongly encourage you to specify a name. Properly named Capsules provide a degree of runtime type-safety; there is no feasible way to tell one unnamed Capsule from another.

通常来说，Capsule 用于暴露 C API，其名字应该遵循如下规范：

```
modulename.attributename
```

便利函数 `PyCapsule_Import()` 可以方便的载入通过 Capsule 提供的 C API，仅在 Capsule 的名字匹配时。这个行为为 C API 用户提供了高度的确定性来载入正确的 C API。

如下例子展示了将大部分负担交由导出模块作者的方法，适用于常用的库模块。其会存储所有 C API 指针 (例子里只有一个) 在 `void` 指针的数组里，并使其值变为 Capsule。对应的模块头文件提供了宏来管理导入模块和获取 C API 指针；客户端模块只需要在访问 C API 前调用这个宏即可。

导出的模块修改自 `spam` 模块，来自一个简单的例子 段落。函数 `spam.system()` 不会直接调用 C 库函数 `system()`，但一个函数 `PySpam_System()` 会负责调用，当然现实中会更复杂些 (例如添加"spam"到每个命令)。函数 `PySpam_System()` 也会导出给其他扩展模块。

函数 `PySpam_System()` 是个纯 C 函数，声明 `static` 就像其他地方那样:

```
static int
PySpam_System(const char *command)
{
    return system(command);
}
```

函数 spam_system() 按照如下方式修改:

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = PySpam_System(command);
    return Py_BuildValue("i", sts);
}
```

在模块开头，在此行后:

```
#include "Python.h"
```

添加另外两行:

```
#define SPAM_MODULE
#include "spammodule.h"
```

#define 用于告知头文件需要包含给导出的模块，而不是客户端模块。最终，模块的初始化函数必须负责初始化 C API 指针数组:

```
PyMODINIT_FUNC
initspam(void)
{
    PyObject *m;
    static void *PySpam_API[PySpam_API_pointers];
    PyObject *c_api_object;

    m = Py_InitModule("spam", SpamMethods);
    if (m == NULL)
        return;

    /* Initialize the C API pointer array */
    PySpam_API[PySpam_System_NUM] = (void *)PySpam_System;

    /* Create a Capsule containing the API pointer array's address */
    c_api_object = PyCapsule_New((void *)PySpam_API, "spam._C_API", NULL);

    if (c_api_object != NULL)
        PyModule_AddObject(m, "_C_API", c_api_object);
}
```

Note that PySpam_API is declared static; otherwise the pointer array would disappear when initspam() terminates!

头文件 spammodule.h 里的一堆工作，看起来如下所示:

```
#ifndef Py_SPAMMODULE_H
#define Py_SPAMMODULE_H
#ifdef __cplusplus
extern "C" {
#endif

/* Header file for spammodule */

/* C API functions */
#define PySpam_System_NUM 0
#define PySpam_System_RETURN int
#define PySpam_System_PROTO (const char *command)

/* Total number of C API pointers */
#define PySpam_API_pointers 1


#ifdef SPAM_MODULE
/* This section is used when compiling spammodule.c */

static PySpam_System_RETURN PySpam_System PySpam_System_PROTO;

#else
/* This section is used in modules that use spammodule's API */

static void **PySpam_API;

#define PySpam_System \
 (*(PySpam_System_RETURN (*)PySpam_System_PROTO) PySpam_API[PySpam_System_NUM])

/* Return -1 on error, 0 on success.
 * PyCapsule_Import will set an exception if there's an error.
 */
static int
import_spam(void)
{
    PySpam_API = (void **)PyCapsule_Import("spam._C_API", 0);
    return (PySpam_API != NULL) ? 0 : -1;
}

#endif

#ifdef __cplusplus
}
#endif

#endif /* !defined(Py_SPAMMODULE_H) */
```

客户端模块必须在其初始化函数里按顺序调用函数 `import_spam()` (或其他宏) 才能访问函数 `PySpam_System()` 。

```
PyMODINIT_FUNC
initclient(void)
{
    PyObject *m;

    m = Py_InitModule("client", ClientMethods);
```

(下页继续)

(续上页)

```
    if (m == NULL)
        return;
    if (import_spam() < 0)
        return;
    /* additional initialization can happen here */
}
```

这种方法的主要缺点是，文件 spammodule.h 过于复杂。当然，对每个要导出的函数，基本结构是相似的，所以只需要学习一次。

最后需要提醒的是 Capsule 提供了额外的功能，用于存储在 Capsule 里的指针的内存分配和释放。细节参考 Python/C API 参考手册的章节 capsules 和 Capsule 的实现 (在 Python 源码发行包的 Include/pycapsule.h 和 Objects/pycapsule.c)。

备注

CHAPTER 2

Defining New Types

As mentioned in the last chapter, Python allows the writer of an extension module to define new types that can be manipulated from Python code, much like strings and lists in core Python.

This is not hard; the code for all extension types follows a pattern, but there are some details that you need to understand before you can get started.

注解: The way new types are defined changed dramatically (and for the better) in Python 2.2. This document documents how to define new types for Python 2.2 and later. If you need to support older versions of Python, you will need to refer to older versions of this documentation.

2.1 The Basics

The Python runtime sees all Python objects as variables of type `PyObject*`. A `PyObject` is not a very magnificent object - it just contains the refcount and a pointer to the object's "type object". This is where the action is; the type object determines which (C) functions get called when, for instance, an attribute gets looked up on an object or it is multiplied by another object. These C functions are called "type methods".

So, if you want to define a new object type, you need to create a new type object.

This sort of thing can only be explained by example, so here's a minimal, but complete, module that defines a new type:

```
#include <Python.h>

typedef struct {
    PyObject_HEAD
    /* Type-specific fields go here. */
} noddy_NoddyObject;

static PyTypeObject noddy_NoddyType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "noddy.Noddy",             /* tp_name */
```

(下页继续)

(续上页)

```
    sizeof(noddy_NoddyObject), /* tp_basicsize */
    0,                         /* tp_itemsize */
    0,                         /* tp_dealloc */
    0,                         /* tp_print */
    0,                         /* tp_getattr */
    0,                         /* tp_setattr */
    0,                         /* tp_compare */
    0,                         /* tp_repr */
    0,                         /* tp_as_number */
    0,                         /* tp_as_sequence */
    0,                         /* tp_as_mapping */
    0,                         /* tp_hash */
    0,                         /* tp_call */
    0,                         /* tp_str */
    0,                         /* tp_getattro */
    0,                         /* tp_setattro */
    0,                         /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT,        /* tp_flags */
    "Noddy objects",           /* tp_doc */
};

static PyMethodDef noddy_methods[] = {
    {NULL}  /* Sentinel */
};

#ifndef PyMODINIT_FUNC        /* declarations for DLL import/export */
#define PyMODINIT_FUNC void
#endif
PyMODINIT_FUNC
initnoddy(void)
{
    PyObject* m;

    noddy_NoddyType.tp_new = PyType_GenericNew;
    if (PyType_Ready(&noddy_NoddyType) < 0)
        return;

    m = Py_InitModule3("noddy", noddy_methods,
                       "Example module that creates an extension type.");

    Py_INCREF(&noddy_NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&noddy_NoddyType);
}
```

Now that's quite a bit to take in at once, but hopefully bits will seem familiar from the last chapter.

The first bit that will be new is:

```
typedef struct {
    PyObject_HEAD
} noddy_NoddyObject;
```

This is what a Noddy object will contain—in this case, nothing more than every Python object contains, namely a refcount and a pointer to a type object. These are the fields the `PyObject_HEAD` macro brings in. The reason for the macro is to standardize the layout and to enable special debugging fields in debug builds. Note that there is no semicolon after the `PyObject_HEAD` macro; one is included in the macro definition. Be wary of adding one by accident; it's easy to do from habit, and your compiler might not complain, but someone else's probably will! (On Windows, MSVC is known to call this an error and refuse to compile the code.)

For contrast, let' s take a look at the corresponding definition for standard Python integers:

```
typedef struct {
    PyObject_HEAD
    long ob_ival;
} PyIntObject;
```

Moving on, we come to the crunch —the type object.

```
static PyTypeObject noddy_NoddyType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "noddy.Noddy",             /* tp_name */
    sizeof(noddy_NoddyObject), /* tp_basicsize */
    0,                         /* tp_itemsize */
    0,                         /* tp_dealloc */
    0,                         /* tp_print */
    0,                         /* tp_getattr */
    0,                         /* tp_setattr */
    0,                         /* tp_compare */
    0,                         /* tp_repr */
    0,                         /* tp_as_number */
    0,                         /* tp_as_sequence */
    0,                         /* tp_as_mapping */
    0,                         /* tp_hash */
    0,                         /* tp_call */
    0,                         /* tp_str */
    0,                         /* tp_getattro */
    0,                         /* tp_setattro */
    0,                         /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT,        /* tp_flags */
    "Noddy objects",           /* tp_doc */
};
```

Now if you go and look up the definition of `PyTypeObject` in `object.h` you' ll see that it has many more fields that the definition above. The remaining fields will be filled with zeros by the C compiler, and it' s common practice to not specify them explicitly unless you need them.

This is so important that we' re going to pick the top of it apart still further:

```
PyVarObject_HEAD_INIT(NULL, 0)
```

This line is a bit of a wart; what we' d like to write is:

```
PyVarObject_HEAD_INIT(&PyType_Type, 0)
```

as the type of a type object is "type", but this isn' t strictly conforming C and some compilers complain. Fortunately, this member will be filled in for us by `PyType_Ready()`.

```
"noddy.Noddy",              /* tp_name */
```

The name of our type. This will appear in the default textual representation of our objects and in some error messages, for example:

```
>>> "" + noddy.new_noddy()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot add type "noddy.Noddy" to string
```

Note that the name is a dotted name that includes both the module name and the name of the type within the module. The module in this case is `noddy` and the type is `Noddy`, so we set the type name to `noddy.Noddy`. One side effect of using an undotted name is that the pydoc documentation tool will not list the new type in the module documentation.

```
sizeof(noddy_NoddyObject),   /* tp_basicsize */
```

This is so that Python knows how much memory to allocate when you call `PyObject_New()`.

注解: If you want your type to be subclassable from Python, and your type has the same `tp_basicsize` as its base type, you may have problems with multiple inheritance. A Python subclass of your type will have to list your type first in its `__bases__`, or else it will not be able to call your type' s `__new__()` method without getting an error. You can avoid this problem by ensuring that your type has a larger value for `tp_basicsize` than its base type does. Most of the time, this will be true anyway, because either your base type will be `object`, or else you will be adding data members to your base type, and therefore increasing its size.

```
0,                          /* tp_itemsize */
```

This has to do with variable length objects like lists and strings. Ignore this for now.

Skipping a number of type methods that we don' t provide, we set the class flags to `Py_TPFLAGS_DEFAULT`.

```
Py_TPFLAGS_DEFAULT,        /* tp_flags */
```

All types should include this constant in their flags. It enables all of the members defined by the current version of Python.

We provide a doc string for the type in `tp_doc`.

```
"Noddy objects",           /* tp_doc */
```

Now we get into the type methods, the things that make your objects different from the others. We aren' t going to implement any of these in this version of the module. We' ll expand this example later to have more interesting behavior.

For now, all we want to be able to do is to create new `Noddy` objects. To enable object creation, we have to provide a `tp_new` implementation. In this case, we can just use the default implementation provided by the API function `PyType_GenericNew()`. We' d like to just assign this to the `tp_new` slot, but we can' t, for portability sake, On some platforms or compilers, we can' t statically initialize a structure member with a function defined in another C module, so, instead, we' ll assign the `tp_new` slot in the module initialization function just before calling `PyType_Ready()`:

```
noddy_NoddyType.tp_new = PyType_GenericNew;
if (PyType_Ready(&noddy_NoddyType) < 0)
    return;
```

All the other type methods are *NULL*, so we' ll go over them later —that' s for a later section!

Everything else in the file should be familiar, except for some code in `initnoddy()`:

```
if (PyType_Ready(&noddy_NoddyType) < 0)
    return;
```

This initializes the `Noddy` type, filing in a number of members, including `ob_type` that we initially set to *NULL*.

```
PyModule_AddObject(m, "Noddy", (PyObject *)&noddy_NoddyType);
```

This adds the type to the module dictionary. This allows us to create `Noddy` instances by calling the `Noddy` class:

```
>>> import noddy
>>> mynoddy = noddy.Noddy()
```

That’s it! All that remains is to build it; put the above code in a file called `noddy.c` and

```
from distutils.core import setup, Extension
setup(name="noddy", version="1.0",
      ext_modules=[Extension("noddy", ["noddy.c"])])
```

in a file called `setup.py`; then typing

```
$ python setup.py build
```

at a shell should produce a file `noddy.so` in a subdirectory; move to that directory and fire up Python —you should be able to `import noddy` and play around with Noddy objects.

That wasn’t so hard, was it?

Of course, the current Noddy type is pretty uninteresting. It has no data and doesn’t do anything. It can’t even be subclassed.

2.1.1 Adding data and methods to the Basic example

Let’s extend the basic example to add some data and methods. Let’s also make the type usable as a base class. We’ll create a new module, `noddy2` that adds these capabilities:

```
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} Noddy;

static void
Noddy_dealloc(Noddy* self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject*)self);
}

static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }

        self->last = PyString_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
```

(下页继续)

(续上页)

```
            return NULL;
        }

        self->number = 0;
    }

    return (PyObject *)self;
}

static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist,
                                      &first, &last,
                                      &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }

    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }

    return 0;
}


static PyMemberDef Noddy_members[] = {
    {"first", T_OBJECT_EX, offsetof(Noddy, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(Noddy, last), 0,
     "last name"},
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL}  /* Sentinel */
};

static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
```

(下页继续)

(续上页)

```
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }

    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);
    if (args == NULL)
        return NULL;

    result = PyString_Format(format, args);
    Py_DECREF(args);

    return result;
}

static PyMethodDef Noddy_methods[] = {
    {"name", (PyCFunction)Noddy_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL}  /* Sentinel */
};

static PyTypeObject NoddyType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "noddy.Noddy",             /* tp_name */
    sizeof(Noddy),             /* tp_basicsize */
    0,                         /* tp_itemsize */
    (destructor)Noddy_dealloc, /* tp_dealloc */
    0,                         /* tp_print */
    0,                         /* tp_getattr */
    0,                         /* tp_setattr */
    0,                         /* tp_compare */
    0,                         /* tp_repr */
    0,                         /* tp_as_number */
    0,                         /* tp_as_sequence */
    0,                         /* tp_as_mapping */
    0,                         /* tp_hash */
    0,                         /* tp_call */
    0,                         /* tp_str */
    0,                         /* tp_getattro */
    0,                         /* tp_setattro */
    0,                         /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT |
        Py_TPFLAGS_BASETYPE,   /* tp_flags */
    "Noddy objects",           /* tp_doc */
    0,                         /* tp_traverse */
```

(下页继续)

(续上页)

```
    0,                         /* tp_clear */
    0,                         /* tp_richcompare */
    0,                         /* tp_weaklistoffset */
    0,                         /* tp_iter */
    0,                         /* tp_iternext */
    Noddy_methods,             /* tp_methods */
    Noddy_members,             /* tp_members */
    0,                         /* tp_getset */
    0,                         /* tp_base */
    0,                         /* tp_dict */
    0,                         /* tp_descr_get */
    0,                         /* tp_descr_set */
    0,                         /* tp_dictoffset */
    (initproc)Noddy_init,      /* tp_init */
    0,                         /* tp_alloc */
    Noddy_new,                 /* tp_new */
};

static PyMethodDef module_methods[] = {
    {NULL}  /* Sentinel */
};

#ifndef PyMODINIT_FUNC        /* declarations for DLL import/export */
#define PyMODINIT_FUNC void
#endif
PyMODINIT_FUNC
initnoddy2(void)
{
    PyObject* m;

    if (PyType_Ready(&NoddyType) < 0)
        return;

    m = Py_InitModule3("noddy2", module_methods,
                       "Example module that creates an extension type.");

    if (m == NULL)
        return;

    Py_INCREF(&NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&NoddyType);
}
```

This version of the module has a number of changes.

We’ve added an extra include:

```
#include <structmember.h>
```

This include provides declarations that we use to handle attributes, as described a bit later.

The name of the `Noddy` object structure has been shortened to `Noddy`. The type object name has been shortened to `NoddyType`.

The `Noddy` type now has three data attributes, *first*, *last*, and *number*. The *first* and *last* variables are Python strings containing first and last names. The *number* attribute is an integer.

The object structure is updated accordingly:

```
typedef struct {
    PyObject_HEAD
    PyObject *first;
    PyObject *last;
    int number;
} Noddy;
```

Because we now have data to manage, we have to be more careful about object allocation and deallocation. At a minimum, we need a deallocation method:

```
static void
Noddy_dealloc(Noddy* self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject*)self);
}
```

which is assigned to the `tp_dealloc` member:

```
(destructor)Noddy_dealloc, /*tp_dealloc*/
```

This method decrements the reference counts of the two Python attributes. We use `Py_XDECREF()` here because the `first` and `last` members could be *NULL*. It then calls the `tp_free` member of the object' s type to free the object' s memory. Note that the object' s type might not be `NoddyType`, because the object may be an instance of a subclass.

We want to make sure that the first and last names are initialized to empty strings, so we provide a new method:

```
static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL)
          {
            Py_DECREF(self);
            return NULL;
          }

        self->last = PyString_FromString("");
        if (self->last == NULL)
          {
            Py_DECREF(self);
            return NULL;
          }

        self->number = 0;
    }

    return (PyObject *)self;
}
```

and install it in the `tp_new` member:

```
Noddy_new,                  /* tp_new */
```

The new member is responsible for creating (as opposed to initializing) objects of the type. It is exposed in Python as the `__new__()` method. See the paper titled "Unifying types and classes in Python" for a detailed discussion of the `__new__()` method. One reason to implement a new method is to assure the initial values of instance variables. In this case, we use the new method to make sure that the initial values of the members `first` and `last` are not *NULL*. If we didn't care whether the initial values were *NULL*, we could have used `PyType_GenericNew()` as our new method, as we did before. `PyType_GenericNew()` initializes all of the instance variable members to *NULL*.

The new method is a static method that is passed the type being instantiated and any arguments passed when the type was called, and that returns the new object created. New methods always accept positional and keyword arguments, but they often ignore the arguments, leaving the argument handling to initializer methods. Note that if the type supports subclassing, the type passed may not be the type being defined. The new method calls the tp_alloc slot to allocate memory. We don't fill the `tp_alloc` slot ourselves. Rather `PyType_Ready()` fills it for us by inheriting it from our base class, which is `object` by default. Most types use the default allocation.

注解: If you are creating a co-operative `tp_new` (one that calls a base type's `tp_new` or `__new__()`), you must *not* try to determine what method to call using method resolution order at runtime. Always statically determine what type you are going to call, and call its `tp_new` directly, or via `type->tp_base->tp_new`. If you do not do this, Python subclasses of your type that also inherit from other Python-defined classes may not work correctly. (Specifically, you may not be able to create instances of such subclasses without getting a `TypeError`.)

We provide an initialization function:

```
static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist,
                                      &first, &last,
                                      &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }

    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }

    return 0;
}
```

by filling the `tp_init` slot.

```
(initproc)Noddy_init,           /* tp_init */
```

The `tp_init` slot is exposed in Python as the `__init__()` method. It is used to initialize an object after it's created. Unlike the new method, we can't guarantee that the initializer is called. The initializer isn't called when unpickling objects and it can be overridden. Our initializer accepts arguments to provide initial values for our instance. Initializers always accept positional and keyword arguments.

Initializers can be called multiple times. Anyone can call the `__init__()` method on our objects. For this reason, we have to be extra careful when assigning the new values. We might be tempted, for example to assign the `first` member like this:

```
if (first) {
    Py_XDECREF(self->first);
    Py_INCREF(first);
    self->first = first;
}
```

But this would be risky. Our type doesn't restrict the type of the `first` member, so it could be any kind of object. It could have a destructor that causes code to be executed that tries to access the `first` member. To be paranoid and protect ourselves against this possibility, we almost always reassign members before decrementing their reference counts. When don't we have to do this?

- when we absolutely know that the reference count is greater than 1
- when we know that deallocation of the object[1] will not cause any calls back into our type's code
- when decrementing a reference count in a `tp_dealloc` handler when garbage-collections is not supported[2]

We want to expose our instance variables as attributes. There are a number of ways to do that. The simplest way is to define member definitions:

```
static PyMemberDef Noddy_members[] = {
    {"first", T_OBJECT_EX, offsetof(Noddy, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(Noddy, last), 0,
     "last name"},
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL}  /* Sentinel */
};
```

and put the definitions in the `tp_members` slot:

```
Noddy_members,             /* tp_members */
```

Each member definition has a member name, type, offset, access flags and documentation string. See the 泛型属性管理 section below for details.

A disadvantage of this approach is that it doesn't provide a way to restrict the types of objects that can be assigned to the Python attributes. We expect the first and last names to be strings, but any Python objects can be assigned. Further, the attributes can be deleted, setting the C pointers to *NULL*. Even though we can make sure the members are initialized to non-*NULL* values, the members can be set to *NULL* if the attributes are deleted.

We define a single method, `name()`, that outputs the objects name as the concatenation of the first and last names.

[1] This is true when we know that the object is a basic type, like a string or a float.

[2] We relied on this in the `tp_dealloc` handler in this example, because our type doesn't support garbage collection. Even if a type supports garbage collection, there are calls that can be made to "untrack" the object from garbage collection, however, these calls are advanced and not covered here.

```
static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }

    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);
    if (args == NULL)
        return NULL;

    result = PyString_Format(format, args);
    Py_DECREF(args);

    return result;
}
```

The method is implemented as a C function that takes a `Noddy` (or `Noddy` subclass) instance as the first argument. Methods always take an instance as the first argument. Methods often take positional and keyword arguments as well, but in this case we don't take any and don't need to accept a positional argument tuple or keyword argument dictionary. This method is equivalent to the Python method:

```
def name(self):
   return "%s %s" % (self.first, self.last)
```

Note that we have to check for the possibility that our `first` and `last` members are *NULL*. This is because they can be deleted, in which case they are set to *NULL*. It would be better to prevent deletion of these attributes and to restrict the attribute values to be strings. We'll see how to do that in the next section.

Now that we've defined the method, we need to create an array of method definitions:

```
static PyMethodDef Noddy_methods[] = {
    {"name", (PyCFunction)Noddy_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL}  /* Sentinel */
};
```

and assign them to the `tp_methods` slot:

```
Noddy_methods,             /* tp_methods */
```

Note that we used the `METH_NOARGS` flag to indicate that the method is passed no arguments.

Finally, we'll make our type usable as a base class. We've written our methods carefully so far so that they don't make any assumptions about the type of the object being created or used, so all we need to do is to add the `Py_TPFLAGS_BASETYPE` to our class flag definition:

```
Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /*tp_flags*/
```

We rename `initnoddy()` to `initnoddy2()` and update the module name passed to `Py_InitModule3()`.

Finally, we update our `setup.py` file to build the new module:

```
from distutils.core import setup, Extension
setup(name="noddy", version="1.0",
      ext_modules=[
         Extension("noddy", ["noddy.c"]),
         Extension("noddy2", ["noddy2.c"]),
         ])
```

2.1.2 Providing finer control over data attributes

In this section, we'll provide finer control over how the `first` and `last` attributes are set in the `Noddy` example. In the previous version of our module, the instance variables `first` and `last` could be set to non-string values or even deleted. We want to make sure that these attributes always contain strings.

```
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first;
    PyObject *last;
    int number;
} Noddy;

static void
Noddy_dealloc(Noddy* self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject*)self);
}

static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }

        self->last = PyString_FromString("");
        if (self->last == NULL) {
```

(下页继续)

(续上页)

```
            Py_DECREF(self);
            return NULL;
        }

        self->number = 0;
    }

    return (PyObject *)self;
}

static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|SSi", kwlist,
                                      &first, &last,
                                      &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }

    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }

    return 0;
}

static PyMemberDef Noddy_members[] = {
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL}  /* Sentinel */
};

static PyObject *
Noddy_getfirst(Noddy *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Noddy_setfirst(Noddy *self, PyObject *value, void *closure)
{
    if (value == NULL) {
```

(下页继续)

(续上页)

```
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }

    if (! PyString_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a string");
        return -1;
    }

    Py_DECREF(self->first);
    Py_INCREF(value);
    self->first = value;

    return 0;
}

static PyObject *
Noddy_getlast(Noddy *self, void *closure)
{
    Py_INCREF(self->last);
    return self->last;
}

static int
Noddy_setlast(Noddy *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }

    if (! PyString_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The last attribute value must be a string");
        return -1;
    }

    Py_DECREF(self->last);
    Py_INCREF(value);
    self->last = value;

    return 0;
}

static PyGetSetDef Noddy_getseters[] = {
    {"first",
     (getter)Noddy_getfirst, (setter)Noddy_setfirst,
     "first name",
     NULL},
    {"last",
     (getter)Noddy_getlast, (setter)Noddy_setlast,
     "last name",
     NULL},
    {NULL}  /* Sentinel */
};
```

(下页继续)

(续上页)

```
static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);
    if (args == NULL)
        return NULL;

    result = PyString_Format(format, args);
    Py_DECREF(args);

    return result;
}

static PyMethodDef Noddy_methods[] = {
    {"name", (PyCFunction)Noddy_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL}  /* Sentinel */
};

static PyTypeObject NoddyType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "noddy.Noddy",             /* tp_name */
    sizeof(Noddy),             /* tp_basicsize */
    0,                         /* tp_itemsize */
    (destructor)Noddy_dealloc, /* tp_dealloc */
    0,                         /* tp_print */
    0,                         /* tp_getattr */
    0,                         /* tp_setattr */
    0,                         /* tp_compare */
    0,                         /* tp_repr */
    0,                         /* tp_as_number */
    0,                         /* tp_as_sequence */
    0,                         /* tp_as_mapping */
    0,                         /* tp_hash */
    0,                         /* tp_call */
    0,                         /* tp_str */
    0,                         /* tp_getattro */
    0,                         /* tp_setattro */
    0,                         /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT |
        Py_TPFLAGS_BASETYPE,   /* tp_flags */
    "Noddy objects",           /* tp_doc */
    0,                         /* tp_traverse */
    0,                         /* tp_clear */
    0,                         /* tp_richcompare */
```

(下页继续)

(续上页)

```
    0,                         /* tp_weaklistoffset */
    0,                         /* tp_iter */
    0,                         /* tp_iternext */
    Noddy_methods,             /* tp_methods */
    Noddy_members,             /* tp_members */
    Noddy_getseters,           /* tp_getset */
    0,                         /* tp_base */
    0,                         /* tp_dict */
    0,                         /* tp_descr_get */
    0,                         /* tp_descr_set */
    0,                         /* tp_dictoffset */
    (initproc)Noddy_init,      /* tp_init */
    0,                         /* tp_alloc */
    Noddy_new,                 /* tp_new */
};

static PyMethodDef module_methods[] = {
    {NULL}  /* Sentinel */
};

#ifndef PyMODINIT_FUNC        /* declarations for DLL import/export */
#define PyMODINIT_FUNC void
#endif
PyMODINIT_FUNC
initnoddy3(void)
{
    PyObject* m;

    if (PyType_Ready(&NoddyType) < 0)
        return;

    m = Py_InitModule3("noddy3", module_methods,
                       "Example module that creates an extension type.");

    if (m == NULL)
        return;

    Py_INCREF(&NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&NoddyType);
}
```

To provide greater control, over the `first` and `last` attributes, we' ll use custom getter and setter functions. Here are the functions for getting and setting the `first` attribute:

```
Noddy_getfirst(Noddy *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Noddy_setfirst(Noddy *self, PyObject *value, void *closure)
{
  if (value == NULL) {
    PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
    return -1;
```

(下页继续)

(续上页)

```
  }

  if (! PyString_Check(value)) {
    PyErr_SetString(PyExc_TypeError,
                    "The first attribute value must be a string");
    return -1;
  }

  Py_DECREF(self->first);
  Py_INCREF(value);
  self->first = value;

  return 0;
}
```

The getter function is passed a Noddy object and a "closure", which is void pointer. In this case, the closure is ignored. (The closure supports an advanced usage in which definition data is passed to the getter and setter. This could, for example, be used to allow a single set of getter and setter functions that decide the attribute to get or set based on data in the closure.)

The setter function is passed the Noddy object, the new value, and the closure. The new value may be *NULL*, in which case the attribute is being deleted. In our setter, we raise an error if the attribute is deleted or if the attribute value is not a string.

We create an array of PyGetSetDef structures:

```
static PyGetSetDef Noddy_getseters[] = {
    {"first",
     (getter)Noddy_getfirst, (setter)Noddy_setfirst,
     "first name",
     NULL},
    {"last",
     (getter)Noddy_getlast, (setter)Noddy_setlast,
     "last name",
     NULL},
    {NULL}  /* Sentinel */
};
```

and register it in the tp_getset slot:

```
Noddy_getseters,           /* tp_getset */
```

to register our attribute getters and setters.

The last item in a PyGetSetDef structure is the closure mentioned above. In this case, we aren't using the closure, so we just pass *NULL*.

We also remove the member definitions for these attributes:

```
static PyMemberDef Noddy_members[] = {
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL}  /* Sentinel */
};
```

We also need to update the tp_init handler to only allow strings[3] to be passed:

[3] We now know that the first and last members are strings, so perhaps we could be less careful about decrementing their reference counts, however,

```
static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|SSi", kwlist,
                                      &first, &last,
                                      &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }

    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }

    return 0;
}
```

With these changes, we can assure that the `first` and `last` members are never *NULL* so we can remove checks for *NULL* values in almost all cases. This means that most of the `Py_XDECREF()` calls can be converted to `Py_DECREF()` calls. The only place we can't change these calls is in the deallocator, where there is the possibility that the initialization of these members failed in the constructor.

We also rename the module initialization function and module name in the initialization function, as we did before, and we add an extra definition to the `setup.py` file.

2.1.3 Supporting cyclic garbage collection

Python has a cyclic-garbage collector that can identify unneeded objects even when their reference counts are not zero. This can happen when objects are involved in cycles. For example, consider:

```
>>> l = []
>>> l.append(l)
>>> del l
```

In this example, we create a list that contains itself. When we delete it, it still has a reference from itself. Its reference count doesn't drop to zero. Fortunately, Python's cyclic-garbage collector will eventually figure out that the list is garbage and free it.

In the second version of the `Noddy` example, we allowed any kind of object to be stored in the `first` or `last` attributes[4]. This means that `Noddy` objects can participate in cycles:

we accept instances of string subclasses. Even though deallocating normal strings won't call back into our objects, we can't guarantee that deallocating an instance of a string subclass won't call back into our objects.

[4] Even in the third version, we aren't guaranteed to avoid cycles. Instances of string subclasses are allowed and string subclasses could allow cycles even if normal strings don't.

```
>>> import noddy2
>>> n = noddy2.Noddy()
>>> l = [n]
>>> n.first = l
```

This is pretty silly, but it gives us an excuse to add support for the cyclic-garbage collector to the `Noddy` example. To support cyclic garbage collection, types need to fill two slots and set a class flag that enables these slots:

```
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first;
    PyObject *last;
    int number;
} Noddy;

static int
Noddy_traverse(Noddy *self, visitproc visit, void *arg)
{
    int vret;

    if (self->first) {
        vret = visit(self->first, arg);
        if (vret != 0)
            return vret;
    }
    if (self->last) {
        vret = visit(self->last, arg);
        if (vret != 0)
            return vret;
    }

    return 0;
}

static int
Noddy_clear(Noddy *self)
{
    PyObject *tmp;

    tmp = self->first;
    self->first = NULL;
    Py_XDECREF(tmp);

    tmp = self->last;
    self->last = NULL;
    Py_XDECREF(tmp);

    return 0;
}

static void
Noddy_dealloc(Noddy* self)
{
    PyObject_GC_UnTrack(self);
```

(下页继续)

(续上页)

```
    Noddy_clear(self);
    Py_TYPE(self)->tp_free((PyObject*)self);
}

static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }

        self->last = PyString_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }

        self->number = 0;
    }

    return (PyObject *)self;
}

static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist,
                                      &first, &last,
                                      &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }

    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }

    return 0;
```

(下页继续)

(续上页)

```
}

static PyMemberDef Noddy_members[] = {
    {"first", T_OBJECT_EX, offsetof(Noddy, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(Noddy, last), 0,
     "last name"},
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL}  /* Sentinel */
};

static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }

    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);
    if (args == NULL)
        return NULL;

    result = PyString_Format(format, args);
    Py_DECREF(args);

    return result;
}

static PyMethodDef Noddy_methods[] = {
    {"name", (PyCFunction)Noddy_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL}  /* Sentinel */
};

static PyTypeObject NoddyType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "noddy.Noddy",             /* tp_name */
    sizeof(Noddy),             /* tp_basicsize */
```

(下页继续)

(续上页)

```
    0,                         /* tp_itemsize */
    (destructor)Noddy_dealloc, /* tp_dealloc */
    0,                         /* tp_print */
    0,                         /* tp_getattr */
    0,                         /* tp_setattr */
    0,                         /* tp_compare */
    0,                         /* tp_repr */
    0,                         /* tp_as_number */
    0,                         /* tp_as_sequence */
    0,                         /* tp_as_mapping */
    0,                         /* tp_hash */
    0,                         /* tp_call */
    0,                         /* tp_str */
    0,                         /* tp_getattro */
    0,                         /* tp_setattro */
    0,                         /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT |
        Py_TPFLAGS_BASETYPE |
        Py_TPFLAGS_HAVE_GC,    /* tp_flags */
    "Noddy objects",           /* tp_doc */
    (traverseproc)Noddy_traverse,   /* tp_traverse */
    (inquiry)Noddy_clear,           /* tp_clear */
    0,                         /* tp_richcompare */
    0,                         /* tp_weaklistoffset */
    0,                         /* tp_iter */
    0,                         /* tp_iternext */
    Noddy_methods,             /* tp_methods */
    Noddy_members,             /* tp_members */
    0,                         /* tp_getset */
    0,                         /* tp_base */
    0,                         /* tp_dict */
    0,                         /* tp_descr_get */
    0,                         /* tp_descr_set */
    0,                         /* tp_dictoffset */
    (initproc)Noddy_init,      /* tp_init */
    0,                         /* tp_alloc */
    Noddy_new,                 /* tp_new */
};

static PyMethodDef module_methods[] = {
    {NULL}  /* Sentinel */
};

#ifndef PyMODINIT_FUNC        /* declarations for DLL import/export */
#define PyMODINIT_FUNC void
#endif
PyMODINIT_FUNC
initnoddy4(void)
{
    PyObject* m;

    if (PyType_Ready(&NoddyType) < 0)
        return;

    m = Py_InitModule3("noddy4", module_methods,
                       "Example module that creates an extension type.");
```

(下页继续)

(续上页)

```
    if (m == NULL)
        return;

    Py_INCREF(&NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&NoddyType);
}
```

The traversal method provides access to subobjects that could participate in cycles:

```
static int
Noddy_traverse(Noddy *self, visitproc visit, void *arg)
{
    int vret;

    if (self->first) {
        vret = visit(self->first, arg);
        if (vret != 0)
            return vret;
    }
    if (self->last) {
        vret = visit(self->last, arg);
        if (vret != 0)
            return vret;
    }

    return 0;
}
```

For each subobject that can participate in cycles, we need to call the `visit()` function, which is passed to the traversal method. The `visit()` function takes as arguments the subobject and the extra argument *arg* passed to the traversal method. It returns an integer value that must be returned if it is non-zero.

Python 2.4 and higher provide a `Py_VISIT()` macro that automates calling visit functions. With `Py_VISIT()`, `Noddy_traverse()` can be simplified:

```
static int
Noddy_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->first);
    Py_VISIT(self->last);
    return 0;
}
```

注解: Note that the `tp_traverse` implementation must name its arguments exactly *visit* and *arg* in order to use `Py_VISIT()`. This is to encourage uniformity across these boring implementations.

We also need to provide a method for clearing any subobjects that can participate in cycles.

```
static int
Noddy_clear(Noddy *self)
{
    PyObject *tmp;

    tmp = self->first;
```

(下页继续)

(续上页)

```
    self->first = NULL;
    Py_XDECREF(tmp);

    tmp = self->last;
    self->last = NULL;
    Py_XDECREF(tmp);

    return 0;
}
```

Notice the use of a temporary variable in `Noddy_clear()`. We use the temporary variable so that we can set each member to *NULL* before decrementing its reference count. We do this because, as was discussed earlier, if the reference count drops to zero, we might cause code to run that calls back into the object. In addition, because we now support garbage collection, we also have to worry about code being run that triggers garbage collection. If garbage collection is run, our `tp_traverse` handler could get called. We can't take a chance of having `Noddy_traverse()` called when a member's reference count has dropped to zero and its value hasn't been set to *NULL*.

Python 2.4 and higher provide a `Py_CLEAR()` that automates the careful decrementing of reference counts. With `Py_CLEAR()`, the `Noddy_clear()` function can be simplified:

```
static int
Noddy_clear(Noddy *self)
{
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}
```

Note that `Noddy_dealloc()` may call arbitrary functions through `__del__` method or weakref callback. It means circular GC can be triggered inside the function. Since GC assumes reference count is not zero, we need to untrack the object from GC by calling `PyObject_GC_UnTrack()` before clearing members. Here is reimplemented deallocator which uses `PyObject_GC_UnTrack()` and `Noddy_clear()`.

```
static void
Noddy_dealloc(Noddy* self)
{
    PyObject_GC_UnTrack(self);
    Noddy_clear(self);
    Py_TYPE(self)->tp_free((PyObject*)self);
}
```

Finally, we add the `Py_TPFLAGS_HAVE_GC` flag to the class flags:

```
Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC, /* tp_flags */
```

That's pretty much it. If we had written custom `tp_alloc` or `tp_free` slots, we'd need to modify them for cyclic-garbage collection. Most extensions will use the versions automatically provided.

2.1.4 Subclassing other types

It is possible to create new extension types that are derived from existing types. It is easiest to inherit from the built in types, since an extension can easily use the `PyTypeObject` it needs. It can be difficult to share these `PyTypeObject` structures between extension modules.

In this example we will create a `Shoddy` type that inherits from the built-in `list` type. The new type will be completely compatible with regular lists, but will have an additional `increment()` method that increases an internal counter.

```
>>> import shoddy
>>> s = shoddy.Shoddy(range(3))
>>> s.extend(s)
>>> print len(s)
6
>>> print s.increment()
1
>>> print s.increment()
2
```

```
#include <Python.h>

typedef struct {
    PyListObject list;
    int state;
} Shoddy;


static PyObject *
Shoddy_increment(Shoddy *self, PyObject *unused)
{
    self->state++;
    return PyInt_FromLong(self->state);
}


static PyMethodDef Shoddy_methods[] = {
    {"increment", (PyCFunction)Shoddy_increment, METH_NOARGS,
     PyDoc_STR("increment state counter")},
    {NULL,          NULL},
};

static int
Shoddy_init(Shoddy *self, PyObject *args, PyObject *kwds)
{
    if (PyList_Type.tp_init((PyObject *)self, args, kwds) < 0)
        return -1;
    self->state = 0;
    return 0;
}


static PyTypeObject ShoddyType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "shoddy.Shoddy",         /* tp_name */
    sizeof(Shoddy),          /* tp_basicsize */
    0,                       /* tp_itemsize */
    0,                       /* tp_dealloc */
```

(下页继续)

(续上页)

```
    0,                          /* tp_print */
    0,                          /* tp_getattr */
    0,                          /* tp_setattr */
    0,                          /* tp_compare */
    0,                          /* tp_repr */
    0,                          /* tp_as_number */
    0,                          /* tp_as_sequence */
    0,                          /* tp_as_mapping */
    0,                          /* tp_hash */
    0,                          /* tp_call */
    0,                          /* tp_str */
    0,                          /* tp_getattro */
    0,                          /* tp_setattro */
    0,                          /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT |
        Py_TPFLAGS_BASETYPE,    /* tp_flags */
    0,                          /* tp_doc */
    0,                          /* tp_traverse */
    0,                          /* tp_clear */
    0,                          /* tp_richcompare */
    0,                          /* tp_weaklistoffset */
    0,                          /* tp_iter */
    0,                          /* tp_iternext */
    Shoddy_methods,             /* tp_methods */
    0,                          /* tp_members */
    0,                          /* tp_getset */
    0,                          /* tp_base */
    0,                          /* tp_dict */
    0,                          /* tp_descr_get */
    0,                          /* tp_descr_set */
    0,                          /* tp_dictoffset */
    (initproc)Shoddy_init,      /* tp_init */
    0,                          /* tp_alloc */
    0,                          /* tp_new */
};

PyMODINIT_FUNC
initshoddy(void)
{
    PyObject *m;

    ShoddyType.tp_base = &PyList_Type;
    if (PyType_Ready(&ShoddyType) < 0)
        return;

    m = Py_InitModule3("shoddy", NULL, "Shoddy module");
    if (m == NULL)
        return;

    Py_INCREF(&ShoddyType);
    PyModule_AddObject(m, "Shoddy", (PyObject *) &ShoddyType);
}
```

As you can see, the source code closely resembles the `Noddy` examples in previous sections. We will break down the main differences between them.

```
typedef struct {
    PyListObject list;
    int state;
} Shoddy;
```

The primary difference for derived type objects is that the base type’ s object structure must be the first value. The base type will already include the `PyObject_HEAD()` at the beginning of its structure.

When a Python object is a `Shoddy` instance, its *PyObject** pointer can be safely cast to both *PyListObject** and *Shoddy**.

```
static int
Shoddy_init(Shoddy *self, PyObject *args, PyObject *kwds)
{
    if (PyList_Type.tp_init((PyObject *)self, args, kwds) < 0)
       return -1;
    self->state = 0;
    return 0;
}
```

In the `__init__` method for our type, we can see how to call through to the `__init__` method of the base type.

This pattern is important when writing a type with custom `new` and `dealloc` methods. The `new` method should not actually create the memory for the object with `tp_alloc`, that will be handled by the base class when calling its `tp_new`.

When filling out the `PyTypeObject()` for the `Shoddy` type, you see a slot for `tp_base()`. Due to cross platform compiler issues, you can’ t fill that field directly with the `PyList_Type()`; it can be done later in the module’ s `init()` function.

```
PyMODINIT_FUNC
initshoddy(void)
{
    PyObject *m;

    ShoddyType.tp_base = &PyList_Type;
    if (PyType_Ready(&ShoddyType) < 0)
        return;

    m = Py_InitModule3("shoddy", NULL, "Shoddy module");
    if (m == NULL)
        return;

    Py_INCREF(&ShoddyType);
    PyModule_AddObject(m, "Shoddy", (PyObject *) &ShoddyType);
}
```

Before calling `PyType_Ready()`, the type structure must have the `tp_base` slot filled in. When we are deriving a new type, it is not necessary to fill out the `tp_alloc` slot with `PyType_GenericNew()` –the allocate function from the base type will be inherited.

After that, calling `PyType_Ready()` and adding the type object to the module is the same as with the basic `Noddy` examples.

2.2 Type Methods

本章节目标是提供一个各种你可以实现的类型方法及其功能的简短介绍。

这是 C 类型 PyTypeObject 的定义，省略了只用于调试构建的字段:

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    char *tp_name; /* For printing, in format "<module>.<name>" */
    int tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    printfunc tp_print;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    cmpfunc tp_compare;
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    long tp_flags;

    char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    long tp_weaklistoffset;

    /* Added in release 2.2 */
```

(下页继续)

(续上页)

```
    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
    struct PyGetSetDef *tp_getset;
    struct _typeobject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    long tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
    newfunc tp_new;
    freefunc tp_free; /* Low-level free-memory routine */
    inquiry tp_is_gc; /* For PyObject_IS_GC */
    PyObject *tp_bases;
    PyObject *tp_mro; /* method resolution order */
    PyObject *tp_cache;
    PyObject *tp_subclasses;
    PyObject *tp_weaklist;

} PyTypeObject;
```

Now that’s a *lot* of methods. Don’t worry too much though - if you have a type you want to define, the chances are very good that you will only implement a handful of these.

As you probably expect by now, we’re going to go over this and give more information about the various handlers. We won’t go in the order they are defined in the structure, because there is a lot of historical baggage that impacts the ordering of the fields; be sure your type initialization keeps the fields in the right order! It’s often easiest to find an example that includes all the fields you need (even if they’re initialized to `0`) and then change the values to suit your new type.

```
char *tp_name; /* For printing */
```

The name of the type - as mentioned in the last section, this will appear in various places, almost entirely for diagnostic purposes. Try to choose something that will be helpful in such a situation!

```
int tp_basicsize, tp_itemsize; /* For allocation */
```

These fields tell the runtime how much memory to allocate when new objects of this type are created. Python has some built-in support for variable length structures (think: strings, lists) which is where the `tp_itemsize` field comes in. This will be dealt with later.

```
char *tp_doc;
```

这里你可以放置一段字符串（或者它的地址），当你想在 Python 脚本引用 `obj.__doc__` 时返回这段文档字符串。

Now we come to the basic type methods—the ones most extension types will implement.

2.2.1 终结和内存释放

```
destructor tp_dealloc;
```

当您的类型实例的引用计数减少为零并且 Python 解释器想要回收它时，将调用此函数。如果你的类型有内存可供释放或执行其他清理，你可以把它放在这里。对象本身也需要在这里释放。以下是此函数的示例：

```
static void
newdatatype_dealloc(newdatatypeobject * obj)
{
    free(obj->obj_UnderlyingDatatypePtr);
    Py_TYPE(obj)->tp_free(obj);
}
```

One important requirement of the deallocator function is that it leaves any pending exceptions alone. This is important since deallocators are frequently called as the interpreter unwinds the Python stack; when the stack is unwound due to an exception (rather than normal returns), nothing is done to protect the deallocators from seeing that an exception has already been set. Any actions which a deallocator performs which may cause additional Python code to be executed may detect that an exception has been set. This can lead to misleading errors from the interpreter. The proper way to protect against this is to save a pending exception before performing the unsafe action, and restoring it when done. This can be done using the `PyErr_Fetch()` and `PyErr_Restore()` functions:

```
static void
my_dealloc(PyObject *obj)
{
    MyObject *self = (MyObject *) obj;
    PyObject *cbresult;

    if (self->my_callback != NULL) {
        PyObject *err_type, *err_value, *err_traceback;
        int have_error = PyErr_Occurred() ? 1 : 0;

        if (have_error)
            PyErr_Fetch(&err_type, &err_value, &err_traceback);

        cbresult = PyObject_CallObject(self->my_callback, NULL);
        if (cbresult == NULL)
            PyErr_WriteUnraisable(self->my_callback);
        else
            Py_DECREF(cbresult);

        if (have_error)
            PyErr_Restore(err_type, err_value, err_traceback);

        Py_DECREF(self->my_callback);
    }
    Py_TYPE(obj)->tp_free((PyObject*)self);
}
```

2.2.2 对象展示

In Python, there are three ways to generate a textual representation of an object: the `repr()` function (or equivalent back-tick syntax), the `str()` function, and the `print` statement. For most objects, the `print` statement is equivalent to the `str()` function, but it is possible to special-case printing to a `FILE*` if necessary; this should only be done if efficiency is identified as a problem and profiling suggests that creating a temporary string object to be written to a file is too expensive.

These handlers are all optional, and most types at most need to implement the `tp_str` and `tp_repr` handlers.

```
reprfunc tp_repr;
reprfunc tp_str;
printfunc tp_print;
```

`tp_repr` 处理程序应该返回一个字符串对象，其中包含调用它的实例的表示形式。下面是一个简单的例子:

```
static PyObject *
newdatatype_repr(newdatatypeobject * obj)
{
    return PyString_FromFormat("Repr-ified_newdatatype{{size:\%d}}",
                               obj->obj_UnderlyingDatatypePtr->size);
}
```

如果没有指定 `tp_repr` 处理程序，解释器将提供一个使用 `tp_name` 的表示形式以及对象的惟一标识值。

The `tp_str` handler is to `str()` what the `tp_repr` handler described above is to `repr()`; that is, it is called when Python code calls `str()` on an instance of your object. Its implementation is very similar to the `tp_repr` function, but the resulting string is intended for human consumption. If `tp_str` is not specified, the `tp_repr` handler is used instead.

下面是一个简单的例子:

```
static PyObject *
newdatatype_str(newdatatypeobject * obj)
{
    return PyString_FromFormat("Stringified_newdatatype{{size:\%d}}",
                               obj->obj_UnderlyingDatatypePtr->size);
}
```

The print function will be called whenever Python needs to "print" an instance of the type. For example, if 'node' is an instance of type TreeNode, then the print function is called when Python code calls:

```
print node
```

There is a flags argument and one flag, `Py_PRINT_RAW`, and it suggests that you print without string quotes and possibly without interpreting escape sequences.

The print function receives a file object as an argument. You will likely want to write to that file object.

Here is a sample print function:

```
static int
newdatatype_print(newdatatypeobject *obj, FILE *fp, int flags)
{
    if (flags & Py_PRINT_RAW) {
        fprintf(fp, "<{newdatatype object--size: %d}>",
                obj->obj_UnderlyingDatatypePtr->size);
    }
```

(下页继续)

(续上页)

```
    else {
        fprintf(fp, "\"<{newdatatype object--size: %d}>\"",
                obj->obj_UnderlyingDatatypePtr->size);
    }
    return 0;
}
```

2.2.3 Attribute Management

For every object which can support attributes, the corresponding type must provide the functions that control how the attributes are resolved. There needs to be a function which can retrieve attributes (if any are defined), and another to set attributes (if setting attributes is allowed). Removing an attribute is a special case, for which the new value passed to the handler is *NULL*.

Python supports two pairs of attribute handlers; a type that supports attributes only needs to implement the functions for one pair. The difference is that one pair takes the name of the attribute as a `char*`, while the other accepts a `PyObject*`. Each type can use whichever pair makes more sense for the implementation' s convenience.

```
getattrfunc  tp_getattr;        /* char * version */
setattrfunc  tp_setattr;
/* ... */
getattrofunc tp_getattrofunc;   /* PyObject * version */
setattrofunc tp_setattrofunc;
```

If accessing attributes of an object is always a simple operation (this will be explained shortly), there are generic implementations which can be used to provide the `PyObject*` version of the attribute management functions. The actual need for type-specific attribute handlers almost completely disappeared starting with Python 2.2, though there are many examples which have not been updated to use some of the new generic mechanism that is available.

泛型属性管理

2.2 新版功能.

Most extension types only use *simple* attributes. So, what makes the attributes simple? There are only a couple of conditions that must be met:

1. The name of the attributes must be known when `PyType_Ready()` is called.
2. 不需要特殊的处理来记录属性是否被查找或设置，也不需要根据值采取操作。

请注意，此列表不对属性的值、值的计算时间或相关数据的存储方式施加任何限制。

When `PyType_Ready()` is called, it uses three tables referenced by the type object to create *descriptor*s which are placed in the dictionary of the type object. Each descriptor controls access to one attribute of the instance object. Each of the tables is optional; if all three are *NULL*, instances of the type will only have attributes that are inherited from their base type, and should leave the `tp_getattro` and `tp_setattro` fields *NULL* as well, allowing the base type to handle attributes.

表被声明为 object:: 类型的三个字段:

```
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
```

If `tp_methods` is not *NULL*, it must refer to an array of `PyMethodDef` structures. Each entry in the table is an instance of this structure:

```
typedef struct PyMethodDef {
    const char  *ml_name;       /* method name */
    PyCFunction  ml_meth;       /* implementation function */
    int          ml_flags;      /* flags */
    const char  *ml_doc;        /* docstring */
} PyMethodDef;
```

One entry should be defined for each method provided by the type; no entries are needed for methods inherited from a base type. One additional entry is needed at the end; it is a sentinel that marks the end of the array. The `ml_name` field of the sentinel must be *NULL*.

XXX Need to refer to some unified discussion of the structure fields, shared with the next section.

The second table is used to define attributes which map directly to data stored in the instance. A variety of primitive C types are supported, and access may be read-only or read-write. The structures in the table are defined as:

```
typedef struct PyMemberDef {
    char *name;
    int   type;
    int   offset;
    int   flags;
    char *doc;
} PyMemberDef;
```

For each entry in the table, a *descriptor* will be constructed and added to the type which will be able to extract a value from the instance structure. The `type` field should contain one of the type codes defined in the `structmember.h` header; the value will be used to determine how to convert Python values to and from C values. The `flags` field is used to store flags which control how the attribute can be accessed.

XXX Need to move some of this to a shared section!

以下标志常量定义在:file: ‘structmember.h ‘; 它们可以使用 bitwise-OR 组合。

常数	含义
`READONLY`	没有可写的
`RO`	Shorthand for `READONLY`.
`READ_RESTRICTED`	Not readable in restricted mode.
`WRITE_RESTRICTED`	Not writable in restricted mode.
`RESTRICTED`	在受限模式下不可读，也不可写。

An interesting advantage of using the `tp_members` table to build descriptors that are used at runtime is that any attribute defined this way can have an associated doc string simply by providing the text in the table. An application can use the introspection API to retrieve the descriptor from the class object, and get the doc string using its `__doc__` attribute.

As with the `tp_methods` table, a sentinel entry with a `name` value of *NULL* is required.

Type-specific Attribute Management

For simplicity, only the `char*` version will be demonstrated here; the type of the name parameter is the only difference between the `char*` and `PyObject*` flavors of the interface. This example effectively does the same thing as the generic example above, but does not use the generic support added in Python 2.2. The value in showing this is two-fold: it demonstrates how basic attribute management can be done in a way that is portable to older versions of Python, and explains how the handler functions are called, so that if you do need to extend their functionality, you' ll understand what needs to be done.

The `tp_getattr` handler is called when the object requires an attribute look-up. It is called in the same situations where the `__getattr__()` method of a class would be called.

A likely way to handle this is (1) to implement a set of functions (such as `newdatatype_getSize()` and `newdatatype_setSize()` in the example below), (2) provide a method table listing these functions, and (3) provide a getattr function that returns the result of a lookup in that table. The method table uses the same structure as the `tp_methods` field of the type object.

Here is an example:

```
static PyMethodDef newdatatype_methods[] = {
    {"getSize", (PyCFunction)newdatatype_getSize, METH_VARARGS,
     "Return the current size."},
    {"setSize", (PyCFunction)newdatatype_setSize, METH_VARARGS,
     "Set the size."},
    {NULL, NULL, 0, NULL}           /* sentinel */
};

static PyObject *
newdatatype_getattr(newdatatypeobject *obj, char *name)
{
    return Py_FindMethod(newdatatype_methods, (PyObject *)obj, name);
}
```

The `tp_setattr` handler is called when the `__setattr__()` or `__delattr__()` method of a class instance would be called. When an attribute should be deleted, the third parameter will be *NULL*. Here is an example that simply raises an exception; if this were really all you wanted, the `tp_setattr` handler should be set to *NULL*.

```
static int
newdatatype_setattr(newdatatypeobject *obj, char *name, PyObject *v)
{
    (void)PyErr_Format(PyExc_RuntimeError, "Read-only attribute: \%s", name);
    return -1;
}
```

2.2.4 Object Comparison

```
cmpfunc tp_compare;
```

The `tp_compare` handler is called when comparisons are needed and the object does not implement the specific rich comparison method which matches the requested comparison. (It is always used if defined and the `PyObject_Compare()` or `PyObject_Cmp()` functions are used, or if `cmp()` is used from Python.) It is analogous to the `__cmp__()` method. This function should return `-1` if *obj1* is less than *obj2*, `0` if they are equal, and `1` if *obj1* is greater than *obj2*. (It was previously allowed to return arbitrary negative or positive integers for less than and greater than, respectively; as of Python 2.2, this is no longer allowed. In the future, other return values may be assigned a different meaning.)

A `tp_compare` handler may raise an exception. In this case it should return a negative value. The caller has to test for the exception using `PyErr_Occurred()`.

Here is a sample implementation:

```
static int
newdatatype_compare(newdatatypeobject * obj1, newdatatypeobject * obj2)
{
    long result;

    if (obj1->obj_UnderlyingDatatypePtr->size <
        obj2->obj_UnderlyingDatatypePtr->size) {
        result = -1;
    }
    else if (obj1->obj_UnderlyingDatatypePtr->size >
             obj2->obj_UnderlyingDatatypePtr->size) {
        result = 1;
    }
    else {
        result = 0;
    }
    return result;
}
```

2.2.5 Abstract Protocol Support

Python supports a variety of *abstract* ‘protocols;’ the specific interfaces provided to use these interfaces are documented in abstract.

A number of these abstract interfaces were defined early in the development of the Python implementation. In particular, the number, mapping, and sequence protocols have been part of Python since the beginning. Other protocols have been added over time. For protocols which depend on several handler routines from the type implementation, the older protocols have been defined as optional blocks of handlers referenced by the type object. For newer protocols there are additional slots in the main type object, with a flag bit being set to indicate that the slots are present and should be checked by the interpreter. (The flag bit does not indicate that the slot values are non-*NULL*. The flag may be set to indicate the presence of a slot, but a slot may still be unfilled.)

```
PyNumberMethods   *tp_as_number;
PySequenceMethods *tp_as_sequence;
PyMappingMethods  *tp_as_mapping;
```

If you wish your object to be able to act like a number, a sequence, or a mapping object, then you place the address of a structure that implements the C type `PyNumberMethods`, `PySequenceMethods`, or `PyMappingMethods`, respectively. It is up to you to fill in this structure with appropriate values. You can find examples of the use of each of these in the `Objects` directory of the Python source distribution.

```
hashfunc tp_hash;
```

This function, if you choose to provide it, should return a hash number for an instance of your data type. Here is a moderately pointless example:

```
static long
newdatatype_hash(newdatatypeobject *obj)
{
    long result;
    result = obj->obj_UnderlyingDatatypePtr->size;
```

(下页继续)

(续上页)

```
    result = result * 3;
    return result;
}
```

```
ternaryfunc tp_call;
```

This function is called when an instance of your data type is "called", for example, if `obj1` is an instance of your data type and the Python script contains `obj1('hello')`, the `tp_call` handler is invoked.

This function takes three arguments:

1. *arg1* is the instance of the data type which is the subject of the call. If the call is `obj1('hello')`, then *arg1* is `obj1`.
2. *arg2* is a tuple containing the arguments to the call. You can use `PyArg_ParseTuple()` to extract the arguments.
3. *arg3* is a dictionary of keyword arguments that were passed. If this is non-*NULL* and you support keyword arguments, use `PyArg_ParseTupleAndKeywords()` to extract the arguments. If you do not want to support keyword arguments and this is non-*NULL*, raise a `TypeError` with a message saying that keyword arguments are not supported.

Here is a desultory example of the implementation of the call function.

```
/* Implement the call function.
 *    obj1 is the instance receiving the call.
 *    obj2 is a tuple containing the arguments to the call, in this
 *         case 3 strings.
 */
static PyObject *
newdatatype_call(newdatatypeobject *obj, PyObject *args, PyObject *other)
{
    PyObject *result;
    char *arg1;
    char *arg2;
    char *arg3;

    if (!PyArg_ParseTuple(args, "sss:call", &arg1, &arg2, &arg3)) {
        return NULL;
    }
    result = PyString_FromFormat(
        "Returning -- value: [\%d] arg1: [\%s] arg2: [\%s] arg3: [\%s]\n",
        obj->obj_UnderlyingDatatypePtr->size,
        arg1, arg2, arg3);
    printf("\%s", PyString_AS_STRING(result));
    return result;
}
```

XXX some fields need to be added here…

```
/* Added in release 2.2 */
/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;
```

These functions provide support for the iterator protocol. Any object which wishes to support iteration over its contents (which may be generated during iteration) must implement the `tp_iter` handler. Objects which are returned by a `tp_iter` handler must implement both the `tp_iter` and `tp_iternext` handlers. Both handlers take exactly one parameter, the instance for which they are being called, and return a new reference. In the case of an error, they should set an exception and return *NULL*.

For an object which represents an iterable collection, the `tp_iter` handler must return an iterator object. The iterator object is responsible for maintaining the state of the iteration. For collections which can support multiple iterators which do not interfere with each other (as lists and tuples do), a new iterator should be created and returned. Objects which can only be iterated over once (usually due to side effects of iteration) should implement this handler by returning a new reference to themselves, and should also implement the `tp_iternext` handler. File objects are an example of such an iterator.

Iterator objects should implement both handlers. The `tp_iter` handler should return a new reference to the iterator (this is the same as the `tp_iter` handler for objects which can only be iterated over destructively). The `tp_iternext` handler should return a new reference to the next object in the iteration if there is one. If the iteration has reached the end, it may return *NULL* without setting an exception or it may set `StopIteration`; avoiding the exception can yield slightly better performance. If an actual error occurs, it should set an exception and return *NULL*.

2.2.6 Weak Reference Support

One of the goals of Python's weak-reference implementation is to allow any type to participate in the weak reference mechanism without incurring the overhead on those objects which do not benefit by weak referencing (such as numbers).

For an object to be weakly referencable, the extension must include a `PyObject*` field in the instance structure for the use of the weak reference mechanism; it must be initialized to *NULL* by the object's constructor. It must also set the `tp_weaklistoffset` field of the corresponding type object to the offset of the field. For example, the instance type is defined with the following structure:

```
typedef struct {
    PyObject_HEAD
    PyClassObject *in_class;       /* The class object */
    PyObject      *in_dict;        /* A dictionary */
    PyObject      *in_weakreflist; /* List of weak references */
} PyInstanceObject;
```

The statically-declared type object for instances is defined this way:

```
PyTypeObject PyInstance_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "module.instance",

    /* Lots of stuff omitted for brevity... */

    Py_TPFLAGS_DEFAULT,                          /* tp_flags */
    0,                                           /* tp_doc */
    0,                                           /* tp_traverse */
    0,                                           /* tp_clear */
    0,                                           /* tp_richcompare */
    offsetof(PyInstanceObject, in_weakreflist), /* tp_weaklistoffset */
};
```

The type constructor is responsible for initializing the weak reference list to *NULL*:

```
static PyObject *
instance_new() {
    /* Other initialization stuff omitted for brevity */

```

(下页继续)

(续上页)

```
    self->in_weakreflist = NULL;

    return (PyObject *) self;
}
```

The only further addition is that the destructor needs to call the weak reference manager to clear any weak references. This is only required if the weak reference list is non-*NULL*:

```
static void
instance_dealloc(PyInstanceObject *inst)
{
    /* Allocate temporaries if needed, but do not begin
       destruction just yet.
     */

    if (inst->in_weakreflist != NULL)
        PyObject_ClearWeakRefs((PyObject *) inst);

    /* Proceed with object destruction normally. */
}
```

2.2.7 更多建议

Remember that you can omit most of these functions, in which case you provide `0` as a value. There are type definitions for each of the functions you must provide. They are in `object.h` in the Python include directory that comes with the source distribution of Python.

In order to learn how to implement any specific method for your new data type, do the following: Download and unpack the Python source distribution. Go the `Objects` directory, then search the C source files for `tp_` plus the function you want (for example, `tp_print` or `tp_compare`). You will find examples of the function you want to implement.

When you need to verify that an object is an instance of the type you are implementing, use the `PyObject_TypeCheck()` function. A sample of its use might be something like the following:

```
if (! PyObject_TypeCheck(some_object, &MyType)) {
    PyErr_SetString(PyExc_TypeError, "arg #1 not a mything");
    return NULL;
}
```

CHAPTER 3

使用 distutils 构建 C 和 C++ 扩展

Starting in Python 1.4, Python provides, on Unix, a special make file for building make files for building dynamically-linked extensions and custom interpreters. Starting with Python 2.0, this mechanism (known as related to Makefile.pre.in, and Setup files) is no longer supported. Building custom interpreters was rarely used, and extension modules can be built using distutils.

Building an extension module using distutils requires that distutils is installed on the build machine, which is included in Python 2.x and available separately for Python 1.5. Since distutils also supports creation of binary packages, users don't necessarily need a compiler and distutils to install the extension.

一个 distutils 包包含了一个驱动脚本 `setup.py` 。这是个纯 Python 文件，大多数时候也很简单，看起来如下：

```
from distutils.core import setup, Extension

module1 = Extension('demo',
                    sources = ['demo.c'])

setup (name = 'PackageName',
       version = '1.0',
       description = 'This is a demo package',
       ext_modules = [module1])
```

通过文件 `setup.py` ，和文件 `demo.c` ，运行如下

```
python setup.py build
```

这会编译 `demo.c` ，然后产生一个扩展模块叫做 `demo` 在目录 `build` 里。依赖于系统，模块文件会放在某个子目录形如 `build/lib.system` ，名字可能是 `demo.so` 或 `demo.pyd` 。

In the `setup.py`, all execution is performed by calling the `setup` function. This takes a variable number of keyword arguments, of which the example above uses only a subset. Specifically, the example specifies meta-information to build packages, and it specifies the contents of the package. Normally, a package will contain of addition modules, like Python source modules, documentation, subpackages, etc. Please refer to the distutils documentation in distutils-index to learn more about the features of distutils; this section explains building extension modules only.

It is common to pre-compute arguments to `setup()`, to better structure the driver script. In the example above, the ext_modules argument to setup() is a list of extension modules, each of which is an instance of the Extension. In the example, the instance defines an extension named demo which is build by compiling a single source file, demo.c.

更多时候，构建一个扩展会复杂的多，需要额外的预处理器定义和库。如下例子展示了这些。

```
from distutils.core import setup, Extension

module1 = Extension('demo',
                    define_macros = [('MAJOR_VERSION', '1'),
                                     ('MINOR_VERSION', '0')],
                    include_dirs = ['/usr/local/include'],
                    libraries = ['tcl83'],
                    library_dirs = ['/usr/local/lib'],
                    sources = ['demo.c'])

setup (name = 'PackageName',
       version = '1.0',
       description = 'This is a demo package',
       author = 'Martin v. Loewis',
       author_email = 'martin@v.loewis.de',
       url = 'https://docs.python.org/extending/building',
       long_description = '''
This is really just a demo package.
''',
       ext_modules = [module1])
```

In this example, setup() is called with additional meta-information, which is recommended when distribution packages have to be built. For the extension itself, it specifies preprocessor defines, include directories, library directories, and libraries. Depending on the compiler, distutils passes this information in different ways to the compiler. For example, on Unix, this may result in the compilation commands

```
gcc -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -fPIC -DMAJOR_VERSION=1 -DMINOR_
↪VERSION=0 -I/usr/local/include -I/usr/local/include/python2.2 -c demo.c -o build/
↪temp.linux-i686-2.2/demo.o

gcc -shared build/temp.linux-i686-2.2/demo.o -L/usr/local/lib -ltcl83 -o build/lib.
↪linux-i686-2.2/demo.so
```

这些行代码仅用于展示目的；distutils 用户应该相信 distutils 能正确调用。

3.1 发布你的扩展模块

当一个扩展已经成功的构建过，有三种方式使用。

最终用户通常想要安装模块，可以这么运行

```
python setup.py install
```

模块维护者应该制作源码包；要实现可以运行

```
python setup.py sdist
```

In some cases, additional files need to be included in a source distribution; this is done through a MANIFEST.in file; see the distutils documentation for details.

如果源码发行包成功构建了，维护者也可以创建二进制发行包。依赖于平台，一个可用的命令如下

```
python setup.py bdist_wininst
python setup.py bdist_rpm
python setup.py bdist_dumb
```

CHAPTER 4

在 Windows 平台编译 C 和 C++ 扩展

This chapter briefly explains how to create a Windows extension module for Python using Microsoft Visual C++, and follows with more detailed background information on how it works. The explanatory material is useful for both the Windows programmer learning to build Python extensions and the Unix programmer interested in producing software which can be successfully built on both Unix and Windows.

Module authors are encouraged to use the distutils approach for building extension modules, instead of the one described in this section. You will still need the C compiler that was used to build Python; typically Microsoft Visual C++.

注解： This chapter mentions a number of filenames that include an encoded Python version number. These filenames are represented with the version number shown as `XY`; in practice, `'X'` will be the major version number and `'Y'` will be the minor version number of the Python release you’ re working with. For example, if you are using Python 2.2.1, `XY` will actually be `22`.

4.1 A Cookbook Approach

There are two approaches to building extension modules on Windows, just as there are on Unix: use the `distutils` package to control the build process, or do things manually. The distutils approach works well for most extensions; documentation on using `distutils` to build and package extension modules is available in distutils-index. If you find you really need to do things manually, it may be instructive to study the project file for the winsound standard library module.

4.2 Differences Between Unix and Windows

Unix and Windows use completely different paradigms for run-time loading of code. Before you try to build a module that can be dynamically loaded, be aware of how your system works.

In Unix, a shared object (`.so`) file contains code to be used by the program, and also the names of functions and data that it expects to find in the program. When the file is joined to the program, all references to those functions and data in the file' s code are changed to point to the actual locations in the program where the functions and data are placed in memory. This is basically a link operation.

In Windows, a dynamic-link library (`.dll`) file has no dangling references. Instead, an access to functions or data goes through a lookup table. So the DLL code does not have to be fixed up at runtime to refer to the program' s memory; instead, the code already uses the DLL' s lookup table, and the lookup table is modified at runtime to point to the functions and data.

In Unix, there is only one type of library file (`.a`) which contains code from several object files (`.o`). During the link step to create a shared object file (`.so`), the linker may find that it doesn' t know where an identifier is defined. The linker will look for it in the object files in the libraries; if it finds it, it will include all the code from that object file.

In Windows, there are two types of library, a static library and an import library (both called `.lib`). A static library is like a Unix `.a` file; it contains code to be included as necessary. An import library is basically used only to reassure the linker that a certain identifier is legal, and will be present in the program when the DLL is loaded. So the linker uses the information from the import library to build the lookup table for using identifiers that are not included in the DLL. When an application or a DLL is linked, an import library may be generated, which will need to be used for all future DLLs that depend on the symbols in the application or DLL.

Suppose you are building two dynamic-load modules, B and C, which should share another block of code A. On Unix, you would *not* pass `A.a` to the linker for `B.so` and `C.so`; that would cause it to be included twice, so that B and C would each have their own copy. In Windows, building `A.dll` will also build `A.lib`. You *do* pass `A.lib` to the linker for B and C. `A.lib` does not contain code; it just contains information which will be used at runtime to access A' s code.

In Windows, using an import library is sort of like using `import spam`; it gives you access to spam' s names, but does not create a separate copy. On Unix, linking with a library is more like `from spam import *`; it does create a separate copy.

4.3 Using DLLs in Practice

Windows Python is built in Microsoft Visual C++; using other compilers may or may not work (though Borland seems to). The rest of this section is MSVC++ specific.

When creating DLLs in Windows, you must pass `pythonXY.lib` to the linker. To build two DLLs, spam and ni (which uses C functions found in spam), you could use these commands:

```
cl /LD /I/python/include spam.c ../libs/pythonXY.lib
cl /LD /I/python/include ni.c spam.lib ../libs/pythonXY.lib
```

The first command created three files: `spam.obj`, `spam.dll` and `spam.lib`. `Spam.dll` does not contain any Python functions (such as `PyArg_ParseTuple()`), but it does know how to find the Python code thanks to `pythonXY.lib`.

The second command created `ni.dll` (and `.obj` and `.lib`), which knows how to find the necessary functions from spam, and also from the Python executable.

Not every identifier is exported to the lookup table. If you want any other modules (including Python) to be able to see your identifiers, you have to say `_declspec(dllexport)`, as in `void _declspec(dllexport) initspam(void)` or `PyObject _declspec(dllexport) *NiGetSpamData(void)`.

Developer Studio will throw in a lot of import libraries that you do not really need, adding about 100K to your executable. To get rid of them, use the Project Settings dialog, Link tab, to specify *ignore default libraries*. Add the correct `msvcrtxx.lib` to the list of libraries.

CHAPTER 5

在其它应用程序嵌入 Python

The previous chapters discussed how to extend Python, that is, how to extend the functionality of Python by attaching a library of C functions to it. It is also possible to do it the other way around: enrich your C/C++ application by embedding Python in it. Embedding provides your application with the ability to implement some of the functionality of your application in Python rather than C or C++. This can be used for many purposes; one example would be to allow users to tailor the application to their needs by writing some scripts in Python. You can also use it yourself if some of the functionality can be written in Python more easily.

Embedding Python is similar to extending it, but not quite. The difference is that when you extend Python, the main program of the application is still the Python interpreter, while if you embed Python, the main program may have nothing to do with Python —instead, some parts of the application occasionally call the Python interpreter to run some Python code.

So if you are embedding Python, you are providing your own main program. One of the things this main program has to do is initialize the Python interpreter. At the very least, you have to call the function `Py_Initialize()`. There are optional calls to pass command line arguments to Python. Then later you can call the interpreter from any part of the application.

There are several different ways to call the interpreter: you can pass a string containing Python statements to `PyRun_SimpleString()`, or you can pass a stdio file pointer and a file name (for identification in error messages only) to `PyRun_SimpleFile()`. You can also call the lower-level operations described in the previous chapters to construct and use Python objects.

A simple demo of embedding Python can be found in the directory `Demo/embed/` of the source distribution.

参见:

c-api-index The details of Python's C interface are given in this manual. A great deal of necessary information can be found here.

5.1 Very High Level Embedding

The simplest form of embedding Python is the use of the very high level interface. This interface is intended to execute a Python script without needing to interact with the application directly. This can for example be used to perform some operation on a file.

```
#include <Python.h>

int
main(int argc, char *argv[])
{
  Py_SetProgramName(argv[0]);  /* optional but recommended */
  Py_Initialize();
  PyRun_SimpleString("from time import time,ctime\n"
                     "print 'Today is',ctime(time())\n");
  Py_Finalize();
  return 0;
}
```

The `Py_SetProgramName()` function should be called before `Py_Initialize()` to inform the interpreter about paths to Python run-time libraries. Next, the Python interpreter is initialized with `Py_Initialize()`, followed by the execution of a hard-coded Python script that prints the date and time. Afterwards, the `Py_Finalize()` call shuts the interpreter down, followed by the end of the program. In a real program, you may want to get the Python script from another source, perhaps a text-editor routine, a file, or a database. Getting the Python code from a file can better be done by using the `PyRun_SimpleFile()` function, which saves you the trouble of allocating memory space and loading the file contents.

5.2 Beyond Very High Level Embedding: An overview

The high level interface gives you the ability to execute arbitrary pieces of Python code from your application, but exchanging data values is quite cumbersome to say the least. If you want that, you should use lower level calls. At the cost of having to write more C code, you can achieve almost anything.

It should be noted that extending Python and embedding Python is quite the same activity, despite the different intent. Most topics discussed in the previous chapters are still valid. To show this, consider what the extension code from Python to C really does:

1. 转换 Python 的数据值到 C,
2. Perform a function call to a C routine using the converted values, and
3. Convert the data values from the call from C to Python.

When embedding Python, the interface code does:

1. 转换 C 的数据值到 Python,
2. Perform a function call to a Python interface routine using the converted values, and
3. Convert the data values from the call from Python to C.

As you can see, the data conversion steps are simply swapped to accommodate the different direction of the cross-language transfer. The only difference is the routine that you call between both data conversions. When extending, you call a C routine, when embedding, you call a Python routine.

This chapter will not discuss how to convert data from Python to C and vice versa. Also, proper use of references and dealing with errors is assumed to be understood. Since these aspects do not differ from extending the interpreter, you can refer to earlier chapters for the required information.

5.3 纯嵌入

The first program aims to execute a function in a Python script. Like in the section about the very high level interface, the Python interpreter does not directly interact with the application (but that will change in the next section).

The code to run a function defined in a Python script is:

```
#include <Python.h>

int
main(int argc, char *argv[])
{
    PyObject *pName, *pModule, *pFunc;
    PyObject *pArgs, *pValue;
    int i;

    if (argc < 3) {
        fprintf(stderr,"Usage: call pythonfile funcname [args]\n");
        return 1;
    }

    Py_Initialize();
    pName = PyString_FromString(argv[1]);
    /* Error checking of pName left out */

    pModule = PyImport_Import(pName);
    Py_DECREF(pName);

    if (pModule != NULL) {
        pFunc = PyObject_GetAttrString(pModule, argv[2]);
        /* pFunc is a new reference */

        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(argc - 3);
            for (i = 0; i < argc - 3; ++i) {
                pValue = PyInt_FromLong(atoi(argv[i + 3]));
                if (!pValue) {
                    Py_DECREF(pArgs);
                    Py_DECREF(pModule);
                    fprintf(stderr, "Cannot convert argument\n");
                    return 1;
                }
                /* pValue reference stolen here: */
                PyTuple_SetItem(pArgs, i, pValue);
            }
            pValue = PyObject_CallObject(pFunc, pArgs);
            Py_DECREF(pArgs);
            if (pValue != NULL) {
                printf("Result of call: %ld\n", PyInt_AsLong(pValue));
                Py_DECREF(pValue);
            }
            else {
                Py_DECREF(pFunc);
                Py_DECREF(pModule);
                PyErr_Print();
                fprintf(stderr,"Call failed\n");
                return 1;
```

(下页继续)

(续上页)

```
            }
        }
        else {
            if (PyErr_Occurred())
                PyErr_Print();
            fprintf(stderr, "Cannot find function \"%s\"\n", argv[2]);
        }
        Py_XDECREF(pFunc);
        Py_DECREF(pModule);
    }
    else {
        PyErr_Print();
        fprintf(stderr, "Failed to load \"%s\"\n", argv[1]);
        return 1;
    }
    Py_Finalize();
    return 0;
}
```

This code loads a Python script using `argv[1]`, and calls the function named in `argv[2]`. Its integer arguments are the other values of the `argv` array. If you compile and link this program (let' s call the finished executable **call**), and use it to execute a Python script, such as:

```
def multiply(a,b):
    print "Will compute", a, "times", b
    c = 0
    for i in range(0, a):
        c = c + b
    return c
```

然后结果应该是:

```
$ call multiply multiply 3 2
Will compute 3 times 2
Result of call: 6
```

Although the program is quite large for its functionality, most of the code is for data conversion between Python and C, and for error reporting. The interesting part with respect to embedding Python starts with

```
Py_Initialize();
pName = PyString_FromString(argv[1]);
/* Error checking of pName left out */
pModule = PyImport_Import(pName);
```

After initializing the interpreter, the script is loaded using `PyImport_Import()`. This routine needs a Python string as its argument, which is constructed using the `PyString_FromString()` data conversion routine.

```
pFunc = PyObject_GetAttrString(pModule, argv[2]);
/* pFunc is a new reference */

if (pFunc && PyCallable_Check(pFunc)) {
    ...
}
Py_XDECREF(pFunc);
```

Once the script is loaded, the name we' re looking for is retrieved using `PyObject_GetAttrString()`. If the name exists, and the object returned is callable, you can safely assume that it is a function. The program then proceeds by constructing a tuple of arguments as normal. The call to the Python function is then made with:

```
pValue = PyObject_CallObject(pFunc, pArgs);
```

Upon return of the function, `pValue` is either *NULL* or it contains a reference to the return value of the function. Be sure to release the reference after examining the value.

5.4 Extending Embedded Python

Until now, the embedded Python interpreter had no access to functionality from the application itself. The Python API allows this by extending the embedded interpreter. That is, the embedded interpreter gets extended with routines provided by the application. While it sounds complex, it is not so bad. Simply forget for a while that the application starts the Python interpreter. Instead, consider the application to be a set of subroutines, and write some glue code that gives Python access to those routines, just like you would write a normal Python extension. For example:

```
static int numargs=0;

/* Return the number of arguments of the application command line */
static PyObject*
emb_numargs(PyObject *self, PyObject *args)
{
    if(!PyArg_ParseTuple(args, ":numargs"))
        return NULL;
    return Py_BuildValue("i", numargs);
}

static PyMethodDef EmbMethods[] = {
    {"numargs", emb_numargs, METH_VARARGS,
     "Return the number of arguments received by the process."},
    {NULL, NULL, 0, NULL}
};
```

Insert the above code just above the `main()` function. Also, insert the following two statements directly after `Py_Initialize()`:

```
numargs = argc;
Py_InitModule("emb", EmbMethods);
```

These two lines initialize the `numargs` variable, and make the `emb.numargs()` function accessible to the embedded Python interpreter. With these extensions, the Python script can do things like

```
import emb
print "Number of arguments", emb.numargs()
```

In a real application, the methods will expose an API of the application to Python.

5.5 在 C++ 中嵌入 Python

It is also possible to embed Python in a C++ program; precisely how this is done will depend on the details of the C++ system used; in general you will need to write the main program in C++, and use the C++ compiler to compile and link your program. There is no need to recompile Python itself using C++.

5.6 在类 Unix 系统中编译和链接

It is not necessarily trivial to find the right flags to pass to your compiler (and linker) in order to embed the Python interpreter into your application, particularly because Python needs to load library modules implemented as C dynamic extensions (`.so` files) linked against it.

To find out the required compiler and linker flags, you can execute the `pythonX.Y-config` script which is generated as part of the installation process (a `python-config` script may also be available). This script has several options, of which the following will be directly useful to you:

- `pythonX.Y-config --cflags` will give you the recommended flags when compiling:

```
$ /opt/bin/python2.7-config --cflags
-I/opt/include/python2.7 -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -
↪Wstrict-prototypes
```

- `pythonX.Y-config --ldflags` will give you the recommended flags when linking:

```
$ /opt/bin/python2.7-config --ldflags
-L/opt/lib/python2.7/config -lpthread -ldl -lutil -lm -lpython2.7 -Xlinker -
↪export-dynamic
```

注解： To avoid confusion between several Python installations (and especially between the system Python and your own compiled Python), it is recommended that you use the absolute path to `pythonX.Y-config`, as in the above example.

If this procedure doesn’t work for you (it is not guaranteed to work for all Unix-like platforms; however, we welcome bug reports) you will have to read your system’s documentation about dynamic linking and/or examine Python’s `Makefile` (use `sysconfig.get_makefile_filename()` to find its location) and compilation options. In this case, the `sysconfig` module is a useful tool to programmatically extract the configuration values that you will want to combine together. For example:

```
>>> import sysconfig
>>> sysconfig.get_config_var('LIBS')
'-lpthread -ldl  -lutil'
>>> sysconfig.get_config_var('LINKFORSHARED')
'-Xlinker -export-dynamic'
```

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-reference。

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 dict-views.

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 typeiter。

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:

C.3.10 test_epoll

The test_epoll contains the following notice:

(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。

索引

非字母