Extending and Embedding Python

发布 *3.11.6*

Guido van Rossum and the Python development team

十月 27, 2023

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 。

CHAPTER 1

推荐的第三方工具

This guide only covers the basic tools for creating extensions provided as part of this version of CPython. Third party tools like Cython, cffi, SWIG and Numba offer both simpler and more sophisticated approaches to creating C and C++ extensions for Python.

参见:

Python Packaging User Guide: Binary Extensions “Python Packaging User Guide ”不仅涵盖了几个简化二进制扩展创建的可用工具，还讨论了为什么首先创建扩展模块的各种原因。

CHAPTER 2

不使用第三方工具创建扩展

本指南的这一部分包括在没有第三方工具帮助的情况下创建 C 和 C ++ 扩展。它主要用于这些工具的创建者，而不是建议你创建自己的 C 扩展的方法。

2.1 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。

备注： C 扩展接口特指 CPython，扩展模块无法在其他 Python 实现上工作。在大多数情况下，应该避免写 C 扩展，来保持可移植性。举个例子，如果你的用例调用了 C 库或系统调用，你应该考虑使用 `ctypes` 模块或 cffi 库，而不是自己写 C 代码。这些模块允许你写 Python 代码来接口 C 代码，而且可移植性更好。不知为何编译失败了。

2.1.1 一个简单的例子

让我们创建一个扩展模块 `spam` (Monty Python 粉丝最喜欢的食物...) 并且想要创建对应 C 库函数 `system()`[1] 的 Python 接口。这个函数接受一个以 null 结尾的字符串参数并返回一个整数。我们希望可以在 Python 中以如下方式调用此函数:

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

[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。

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。）

文件中开始的两行是：

```
#define PY_SSIZE_T_CLEAN
#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）。

备注： 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。

推荐总是在 Python.h 前定义 PY_SSIZE_T_CLEAN 。查看提取扩展函数的参数 来了解这个宏的更多内容。

所有在 Python.h 中定义的用户可见的符号都具有 Py 或 PY 前缀，已在标准头文件中定义的那些除外。考虑到便利性，也由于其在 Python 解释器中被广泛使用，"Python.h" 还包含了一些标准头文件: <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，它还会直接声明函数 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 PyLong_FromLong(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* 。

对模块级函数，*self* 参数指向模块对象；对于方法则指向对象实例。

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() 在所有参数都有正确类型且组成部分按顺序放在传递进来的地址里时，返回真 (非零)。其在传入无效参数时返回假 (零)。在后续例子里，还会抛出特定异常，使得调用的函数可以理解返回 NULL (也就是例子里所见)。

2.1.2 关于错误和异常

整个 Python 解释器系统有一个如下所述的重要惯例：当一个函数运行失败时，它应当设置一个异常条件并返回一个错误值（通常为 -1 或 NULL 指针）。异常信息保存在解释器线程状态的三个成员中。如果没有异常则它们的值为 NULL。在其他情况下它们是 sys.exc_info() 所返回的 Python 元组的成员的 C 对应物。它们分别是异常类型、异常实例和回溯对象。理解它们对于理解错误是如何被传递的非常重要。

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。

你可以通过 PyErr_Occurred() 在不造成破坏的情况下检测是否设置了异常。这将返回当前异常对象，或者如果未发生异常则返回 NULL。你通常不需要调用 PyErr_Occurred() 来查看函数调用中是否发生了错误，因为你应该能从返回值中看出来。

当一个函数 *f* 调用另一个函数 *g* 时检测到后者出错了，*f* 应当自己返回一个错误值 (通常为 NULL 或 -1)。它不应调用某个 PyErr_* 函数 --- 这类函数已经被 *g* 调用过了。*f* 的调用者随后也应当返回一个错误来提示 它的调用者，同样 不应调用 PyErr_*，依此类推 --- 错误的最详细原因已经由首先检测到它的函数报告了。一旦这个错误到达 Python 解释器的主循环，它会中止当前执行的 Python 代码并尝试找出由 Python 程序员所指定的异常处理句柄。

（在某些情况下模块确实能够通过调用其它 PyErr_* 函数来给出更为详细的错误消息，并且在这些情况下是可以这样做的。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)。

每次失败的 malloc() 调用必须转换为一个异常。malloc() (或 realloc()) 的直接调用者必须调用 PyErr_NoMemory() 来返回错误来提示。所有对象创建函数 (例如 PyLong_FromLong()) 已经这么做了，所以这个提示仅用于直接调用 malloc() 的情况。

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 Python 内置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 来表示文件无法打开（可能应该用 PyExc_OSError 比较好）。如果参数列表有问题，PyArg_ParseTuple() 函数通常会引发 PyExc_TypeError。如果你希望一个参数的值必须在特定范围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;
```

并在模块的初始化函数 (PyInit_spam()) 中附带异常对象对其进行初始化:

```
PyMODINIT_FUNC
PyInit_spam(void)
{
```

(下页继续)

(续上页)

```
    PyObject *m;

    m = PyModule_Create(&spammodule);
    if (m == NULL)
        return NULL;

    SpamError = PyErr_NewException("spam.error", NULL, NULL);
    Py_XINCREF(SpamError);
    if (PyModule_AddObject(m, "error", SpamError) < 0) {
        Py_XDECREF(SpamError);
        Py_CLEAR(SpamError);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

请注意该异常对象的 Python 名称为 `spam.error`。`PyErr_NewException()` 函数可以创建基类为 `Exception` 的类 (除非传入了另一个类而不是 `NULL`)，如 bltin-exceptions 中所描述的。

请注意 `SpamError` 变量保留了一个对新创建的异常类的引用；这是有意为之的！由于异常可能会被外部代码从模块中删除，因此需要拥有一个对该类的引用以确保它不会被丢弃，从而导致 `SpamError` 成为一个悬空指针。如果异常类成为悬空指针，则引发该异常的 C 代码可能会导致核心转储或其他预期之外的附带影响。

本样例稍后将讨论 `PyMODINIT_FUNC` 作为函数返回类型的用法。

可在扩展模块中调用 `PyErr_SetString()` 来引发 `spam.error` 异常，如下所示:

```
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);
}
```

2.1.3 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;
```

如果在参数列表中检测到错误，它将返回 `NULL` (该值是返回对象指针的函数的错误提示)，这取决于 `PyArg_ParseTuple()` 设置的异常。在其他情况下参数的字符串值会被拷贝到局部变量 `command`。这是

一个指针赋值并且你不应该修改它所指向的字符串 (因此在标准 C 中，变量 command 应当被正确地声明为 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);
```

我们的 spam.system() 函数必须以 Python 对象的形式返回 sts 的值。这是通过使用函数 PyLong_FromLong() 完成的。

```
return PyLong_FromLong(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)。

如果你有一个不返回有用参数的 C 函数（即返回 void 的函数），则对应的 Python 函数必须返回 None。你必须使用这种写法（它是通过 Py_RETURN_NONE 宏来实现的）

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

Py_None 是特殊 Python 对象 None 所对应的 C 名称。它是一个真正的 Python 对象而不是 NULL 指针，如我们所见，后者在大多数上下文中都意味着“错误”。

2.1.4 模块方法表和初始化函数

我承诺过要向大家展示如何从 Python 程序中调用 spam_system()。首先，我们需要在“方法表”中列出它的名称和地址:

```
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。

如果应当将关键字参数传给该函数则可以在第三个字段中设置 METH_KEYWORDS 比特位。在此情况下，C 函数应当接受第三个 PyObject * 形参，它将为一个由关键字组成的字典。使用 PyArg_ParseTupleAndKeywords() 来将参数解析为函数。

这个方法表必须被模块定义结构所引用。

```
static struct PyModuleDef spammodule = {
    PyModuleDef_HEAD_INIT,
    "spam",   /* name of module */
    spam_doc, /* module documentation, may be NULL */
    -1,       /* size of per-interpreter state of the module,
                 or -1 if the module keeps state in global variables. */
    SpamMethods
};
```

这个结构体必须在模块的初始化函数中传递给解释器。初始化函数必须命名为 PyInit_name()，其中 *name* 是模块的名称，并且应该是模块文件中定义的唯一非 static 条目:

```
PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModule_Create(&spammodule);
}
```

请注意 PyMODINIT_FUNC 将函数声明为 PyObject * 返回类型，声明了平台所要求的任何特殊链接声明，并针对于 = C++ 将函数声明为 extern "C"。

当 Python 程序首次导入 spam 模块时，PyInit_spam() 将被调用。（有关嵌入 Python 的注释参见下文。）它将调用 PyModule_Create()，该函数会返回一个模块对象，并基于在模块定义中找到的表（一个 PyMethodDef 结构体的数组）将内置函数对象插入到新创建的模块中。PyModule_Create() 返回一个指向它所创建的模块对象的指针。它可能会会程度严重的特定错误而中止，或者在模块无法成功初始化时返回 NULL。初始化函数必须将模块对象返回给其调用者，以便将其插入到 sys.modules 中。

当嵌入 Python 时，除非 PyImport_Inittab 表中有条目，否则不会自动调用 PyInit_spam() 函数。要将模块添加到初始化表中，请使用 PyImport_AppendInittab()，可选择随后导入该模块:

```
int
main(int argc, char *argv[])
{
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]\n");
        exit(1);
    }

    /* Add a built-in module, before Py_Initialize */
    if (PyImport_AppendInittab("spam", PyInit_spam) == -1) {
        fprintf(stderr, "Error: could not extend in-built modules table\n");
        exit(1);
    }

    /* Pass argv[0] to the Python interpreter */
    Py_SetProgramName(program);

    /* Initialize the Python interpreter.  Required.
       If this step fails, it will be a fatal error. */
    Py_Initialize();

    /* Optionally import the module; alternatively,
       import can be deferred until the embedded script
       imports it. */
    PyObject *pmodule = PyImport_ImportModule("spam");
    if (!pmodule) {
        PyErr_Print();
        fprintf(stderr, "Error: could not import module 'spam'\n");
    }

    ...

    PyMem_RawFree(program);
    return 0;
}
```

备注： 要从 `sys.modules` 删除实体或导入已编译模块到一个进程里的多个解释器 (或使用 `fork()` 而没用 `exec()`) 会在一些扩展模块上产生错误。扩展模块作者可以在初始化内部数据结构时给出警告。

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。

备注： 不像我们的 `spam` 例子，`xxmodule` 使用了 *多阶段初始化* (Python3.5 开始引入)，`PyInit_spam` 会返回一个 PyModuleDef 结构体，然后创建的模块放到导入机制。细节参考 **PEP 489** 的多阶段初始化。

2.1.5 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（*构建 C/C++ 扩展* 章节），以及在 Windows 上编译需要的额外信息（*在 Windows 上构建 C 和 C++ 扩展* 章节）。

如果你不使用动态加载，或者想要让模块永久性的作为 Python 解释器的一部分，就必须修改配置设置，并重新构建解释器。幸运的是在 Unix 上很简单，只需要把你的文件 (`spammodule.c` 为例) 放在解压缩源码发行包的 `Modules/` 目录下，添加一行到 `Modules/Setup.local` 来描述你的文件：

```
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。)

如果你的模块需要额外的链接，这些内容可以列出在配置文件里，举个实例：

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

2.1.6 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 函数很简单，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)) {
```

(下页继续)

(续上页)

```
        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;
}
```

此函数必须使用 METH_VARARGS 旗标注册到解释器；这将在模块方法表和初始化函数 一节中详细描述。PyArg_ParseTuple() 函数及其参数的文档见提取扩展函数的参数 一节。

Py_XINCREF() 和 Py_XDECREF() 这两个宏可增加/减少一个对象的引用计数，并且当存在 NULL 指针时仍可保证安全 (但请注意在这个上下文中 *temp* 将不为 NULL)。更多相关信息请参考引用计数 章节。

随后，当要调用此函数时，你将调用 C 函数 PyObject_CallObject()。该函数有两个参数，它们都属于指针，指向任意 Python 对象：即 Python 函数，及其参数列表。参数列表必须总是一个元组对象，其长度即参数的个数量。要不带参数地调用 Python 函数，则传入 NULL 或一个空元组；要带一个参数调用它，则传入一个单元组。Py_BuildValue() 会在其格式字符串包含一对圆括号内的零个或多个格式代码时返回一个元组。例如:

```
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()。

PyObject_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。

但是在你这么做之前，很重要的一点是检查返回值不是 NULL。如果是的话，Python 函数会终止并引发异常。如果调用 PyObject_CallObject() 的 C 代码是在 Python 中发起调用的，它应当立即返回一个错误来告知其 Python 调用者，以便解释器能打印栈回溯信息，或者让调用方 Python 代码能处理该异常。如果这无法做到或不合本意，则应当通过调用 PyErr_Clear() 来清除异常。例如:

```
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 PyObject_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);
```

2.1.7 提取扩展函数的参数

函数 `PyArg_ParseTuple()` 的声明如下:

```
int PyArg_ParseTuple(PyObject *arg, const 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:

```
#define PY_SSIZE_T_CLEAN  /* Make "s#" use Py_ssize_t rather than int. */
#include <Python.h>
```

```
int ok;
int i, j;
long k, l;
const char *s;
Py_ssize_t 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) */
}
```

2.1.8 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,
                                const char *format, char *kwlist[], ...);
```

arg 与 *format* 形参与 `PyArg_ParseTuple()` 函数所定义的一致。*kwdict* 形参是作为第三个参数从 Python 运行时接收的关键字字典。*kwlist* 形参是以 `NULL` 结尾的字符串列表，它被用来标识形参；名称从左至右与来自 *format* 的类型信息相匹配。如果执行成功，`PyArg_ParseTupleAndKeywords()` 会返回真值，否则返回假值并引发一个适当的异常。

备注： 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):

```
#define PY_SSIZE_T_CLEAN  /* Make "s#" use Py_ssize_t rather than int. */
#include <Python.h>

static PyObject *
keywdarg_parrot(PyObject *self, PyObject *args, PyObject *keywds)
{
    int voltage;
    const char *state = "a stiff";
    const char *action = "voom";
    const 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_RETURN_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.
     */
    {"parrot", (PyCFunction)(void(*)(void))keywdarg_parrot, METH_VARARGS | METH_
↪KEYWORDS,
     "Print a lovely skit to standard output."},
    {NULL, NULL, 0, NULL}   /* sentinel */
};

static struct PyModuleDef keywdargmodule = {
    PyModuleDef_HEAD_INIT,
    "keywdarg",
    NULL,
    -1,
    keywdarg_methods
};

PyMODINIT_FUNC
PyInit_keywdarg(void)
{
    return PyModule_Create(&keywdargmodule);
}
```

2.1.9 构造任意值

这个函数与 PyArg_ParseTuple() 很相似，声明如下：

```
PyObject *Py_BuildValue(const 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("y", "hello")              b'hello'
Py_BuildValue("ss", "hello", "world")    ('hello', 'world')
Py_BuildValue("s#", "hello", 4)          'hell'
Py_BuildValue("y#", "hello", 4)          b'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))
```

2.1.10 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。

循环检测器能够检测垃圾回收循环并能回收它们。gc 模块提供了一种运行该检测器的方式 (collect() 函数)，以及多个配置接口和在运行时禁用该检测器的功能。

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]。

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。这会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)。

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。

大多数函数返回一个对象的引用，并传递引用拥有关系。通常，所有创建对象的函数，例如 PyLong_FromLong() 和 Py_BuildValue()，会传递拥有关系给接收者。即便是对象不是真正新的，你仍然可以获得对象的新引用。一个实例是 PyLong_FromLong() 维护了一个流行值的缓存，并可以返回已缓存项目的新引用。

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()。

[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。

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。

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, PyLong_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。

让我们跟随控制流进入 `PyList_SetItem()`。列表拥有对其所有条目的引用，因此当条目 1 被替换时，它必须丢弃原始条目 1。现在我们假设原始条目 1 是一个用户定义类的实例，并进一步假设该类定义了一个 `__del__()` 方法。如果该类实例的引用计数为 1，那么丢弃它时将调用其 `__del__()` 方法。

由于它是用 Python 编写的，因此 `__del__()` 方法可以执行任意 Python 代码。它是否可以使 `bug()` 中对 `item` 的引用失效呢？当然可以！假定传入 `bug()` 的列表可以被 `__del__()` 方法访问，它就可以执行一条语句实现 `del list[0]` 的效果，假定这是对该对象的最后一次引用，它就会释放与之相关联的内存，从而使 `item` 失效。

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, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0);
    Py_DECREF(item);
}
```

这是一个真实的故事。一个较旧版本的 Python 曾经包含此问题的变化形式，有人在 C 语言调试器中花费了大量时间，才弄明白为什么他的 `__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
```

(下页继续)

(续上页)

```
    ...some blocking I/O call...
    Py_END_ALLOW_THREADS
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

NULL 指针

通常，接受对象引用作为参数的函数不希望你传给它们 NULL 指针，并且当你这样做时将会转储核心（或在以后导致核心转储）。返回对象引用的函数通常只在要指明发生了异常时才返回 NULL。不检测 NULL 参数的原因在于这些函数经常要将它们所接收的对象传给其他函数 --- 如果每个函数都检测 NULL，将会导致大量的冗余检测而使代码运行得更缓慢。

更好的做法是仅在“源头”上检测 NULL，即在接收到一个可能为 NULL 的指针，例如来自 malloc() 或是一个可能引发异常的函数的时候。

Py_INCREF() 和 Py_DECREF() 等宏不会检测 NULL 指针 --- 但是，它们的变种 Py_XINCREF() 和 Py_XDECREF() 则会检测。

用于检测特定对象类型的宏 (Pytype_Check()) 不会检测 NULL 指针 --- 同样地，有大量代码会连续调用这些宏来测试一个对象是否为几种不同预期类型之一，这将会生成冗余的测试。不存在带有 NULL 检测的变体。

C 函数调用机制会保证传给 C 函数的参数列表 (本示例中为 args) 绝不会为 NULL --- 实际上它会保证其总是为一个元组[4]。

任何时候将 NULL 指针“泄露”给 Python 用户都会是个严重的错误。

2.1.11 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)。

2.1.12 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用来从一个扩展模块向另一个扩展模块传递 C 层级的信息 (指针): Capsule。一个 Capsule 就是一个存储了指针 (void*) 的 Python 数据类型。Capsule 只能通过其 C API 来创建和访问，但

[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。

它们可以像任何其他 Python 对象一样被传递。特别地，它们可以被赋值给扩展模块命名空间中的一个名称。其他扩展模块将可以导入这个模块，获取该名称对应的值，然后从 Capsule 中获取指针。

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。

无论你选择哪个方法，为你的 Capsule 指定适当的名称都很重要。函数 `PyCapsule_New()` 接受一个 name 形参 (`const char*`)；允许你传入一个 `NULL` 作为名称，但我们强烈推荐你指定名称。正确地命名的 Capsule 提供了一定的运行时类型安全性；没有可行的方式能区别两个未命名的 Capsule。

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。

下面的例子演示了一种将大部分负担交给导出模块编写者的处理方式，这对于常用的库模块来说是合适的。它会将所有 C API 指针（在这个例子里只有一个!）储存到一个 `void` 指针数组，它将成为一个 Capsule 的值。与模块对应的头文件提供了一个宏用来管理导入模块和获取其 C API 指针；客户端模块只需要在访问 C API 之前调用这个宏即可。

导出模块是对一个简单的例子 部分的 `spam` 模块的修改。函数 `spam.system()` 并不直接调用 C 库函数 `system()`，而是调用一个函数 `PySpam_System()`，这个函数在现实中当然会做一些更复杂的事情（比如在每条命令中添加 “sapm”）。该函数 `PySpam_System()` 也会导出给其他扩展模块。

函数 `PySpam_System()` 是一个纯 C 函数，像其他函数一样声明为 `static`:

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

函数 `spam_system()` 已按如下方式修改:

```
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 PyLong_FromLong(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
PyInit_spam(void)
{
    PyObject *m;
    static void *PySpam_API[PySpam_API_pointers];
    PyObject *c_api_object;

    m = PyModule_Create(&spammodule);
    if (m == NULL)
        return NULL;

    /* 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 (PyModule_AddObject(m, "_C_API", c_api_object) < 0) {
        Py_XDECREF(c_api_object);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

请注意 PySpam_API 被声明为 static；否则指针数组会在 PyInit_spam() 终结时消失！

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 \
```

(下页继续)

(续上页)

```
 (*(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) */
```

客户端模块要能够访问函数 `PySpam_System()` 所必须做的事情就是在其初始化函数中调用函数 (实际上是宏) `import_spam()`:

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

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

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

2.2 自定义扩展类型：教程

Python 允许编写 C 扩展模块定义可以从 Python 代码中操纵的新类型，这很像内置的 `str` 和 `list` 类型。所有扩展类型的代码都遵循一个模式，但是在您开始之前，您需要了解一些细节。这份文件是对这个主题介绍。

2.2.1 基础

CPython 运行时会将所有 Python 对象都视为 `PyObject*` 类型的变量，这是所有 Python 对象的“基础类型”。`PyObject` 结构体本身只包含对象的*reference count* 和指向对象的“类型对象”的指针。这是动作所针对的目标。类型对象决定解释器要调用哪些 (C) 函数，例如，在对象上查找一个属性，调用一个方法，或者与另一个对象相乘等。这些 C 函数被称为“类型方法”。

所以，如果你想要定义新的扩展类型，需要创建新的类型对象。

这种事情只能通过例子来解释，下面是一个最小但完整的模块，它在 C 扩展模块 `custom` 中定义了一个名为 `Custom` 的新类型：

备注： 这里展示的方法是定义 *static* 扩展类型的传统方法。可以适合大部分用途。C API 也可以定义在堆上分配的扩展类型，使用 `PyType_FromSpec()` 函数，但不在本入门里讨论。

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

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

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = PyType_GenericNew,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
```

(下页继续)

(续上页)

```
    }

    return m;
}
```

这部分很容易理解，这是为了跟上一章能对接上。这个文件定义了三件事：

1. 一个 Custom **对象**包含的东西：这是 CustomObject 结构体，它会为每个 Custom 实例分配一次。
2. Custom **类型**的行为：这是 CustomType 结构体，它定义了一组旗标和函数指针供解释器在收到特定操作请求时进行检查。
3. 如何初始化 custom 模块：这是 PyInit_custom 函数及其对应的 custommodule 结构体。

结构的第一块是

```
typedef struct {
    PyObject_HEAD
} CustomObject;
```

这就是一个自定义对象将会包含的内容。PyObject_HEAD 是强制要求放在每个对象结构体之前并定义一个名为 ob_base 的 PyObject 类型的字段，其中包含一个指向类型对象和引用计数的指针（这两者可以分别使用宏 Py_TYPE 和 Py_REFCNT 来区分）。使用宏的理由是将布局抽象出来并在 调试编译版中中启用附加字段。

备注： 注意在宏 PyObject_HEAD 后没有分号。意外添加分号会导致编译器提示出错。

当然，对象除了在 PyObject_HEAD 存储数据外，还有额外数据；例如，如下定义了标准的 Python 浮点数:

```
typedef struct {
    PyObject_HEAD
    double ob_fval;
} PyFloatObject;
```

第二个位是类型对象的定义:

```
static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = PyType_GenericNew,
};
```

备注： 推荐使用如上 C99 风格的初始化，以避免列出所有的 PyTypeObject 字段，其中很多是你不需要关心的，这样也可以避免关注字段的定义顺序。

在 object.h 中实际定义的 PyTypeObject 具有比如上定义更多的 字段。剩余的字段会由 C 编译器用零来填充，通常的做法是不显式地指定它们，除非你确实需要它们。

我们先挑选一部分，每次一个字段:

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

这一行是强制的样板，用以初始化如上提到的 ob_base 字段:

```
.tp_name = "custom.Custom",
```

我们的类型的名称。这将出现在我们的对象的默认文本表示形式和某些错误消息中，例如:

```
>>> "" + custom.Custom()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "custom.Custom") to str
```

请注意此名称是一个带点号名称，它同时包括模块名称和模块中的类型名称。本例中的模块是 custom 而类型是 Custom，因此我们将类型名称设为 custom.Custom。使用真正的带点号的导入路径对于使你的类型与 pydoc 和 pickle 模块保持兼容是很重要的。

```
.tp_basicsize = sizeof(CustomObject),
.tp_itemsize = 0,
```

这样能让 Python 知道当创建新的 Custom 实例时需要分配多少内存。tp_itemsize 仅用于可变大小的对象而在其他情况下都应为零。

备注: 如果你希望你的类型可在 Python 中被子类化，并且你的类型和它的基类型具有相同的 tp_basicsize，那么你可能会遇到多重继承问题。你的类型的 Python 中的子类必须在其 __bases__ 中将你的类型列表最前面，否则在调用你的类型的 __new__() 方法时将会出错。你可以通过确保你的类型具有比其基类型最大的 tp_basicsize 值来避免这个问题。在大多数时候，这都是可以的，因为要么你的基类型是 object，要么你将为你的基类型添加数据成员，从而增加其大小。

我们将类旗标设为 Py_TPFLAGS_DEFAULT。

```
.tp_flags = Py_TPFLAGS_DEFAULT,
```

所有类型都应当在它们的旗标中包括此常量。该常量将启用至少在 Python 3.3 之前定义的全部成员。如果你需要更多的成员，你将需要对相应的旗标进行 OR 运算。

我们为 tp_doc 类型提供一个文档字符串.

```
.tp_doc = PyDoc_STR("Custom objects"),
```

要启用对象创建，我们必须提供一个 tp_new 处理句柄。这等价于 Python 方法 __new__()，但必须显式地指定。在这种情况下，我们可以使用 API 函数 PyType_GenericNew() 所提供的默认实现。

```
.tp_new = PyType_GenericNew,
```

除了 PyInit_custom() 中的某些代码以外，文件中的其他内容应该都很容易理解:

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

这将初始化 Custom 类型，为一些成员填充适当的默认值，包括我们在初始时设为 NULL 的 ob_type。

```
Py_INCREF(&CustomType);
if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
    Py_DECREF(&CustomType);
```

(下页继续)

(续上页)

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

这将把类型添加到模块字典中。这样我们就能通过调用 Custom 类来创建 Custom 实例：

```
>>> import custom
>>> mycustom = custom.Custom()
```

好了！接下来要做的就是编译它；将上述代码放到名为 custom.c 的文件中然后:

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

在名为 setup.py 的文件中；然后输入

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

到 shell 中应当会在一个子目录中产生文件 custom.so；进入该目录并运行 Python --- 你应当能够执行 import custom 并尝试使用 Custom 对象。

这并不难，对吗?

当然，当前的自定义类型非常无趣。它没有数据，也不做任何事情。它甚至不能被子类化。

备注： While this documentation showcases the standard distutils module for building C extensions, it is recommended in real-world use cases to use the newer and better-maintained setuptools library. Documentation on how to do this is out of scope for this document and can be found in the Python Packaging User's Guide.

2.2.2 向基本示例添加数据和方法

让我们通过添加一些数据和方法来扩展这个基本示例。让我们再使该类型可以作为基类使用。我们将创建一个新模块 custom2 来添加这些功能：

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

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

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

(下页继续)

(续上页)

```
static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    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 Custom_members[] = {
    {"first", T_OBJECT_EX, offsetof(CustomObject, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(CustomObject, last), 0,
     "last name"},
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL}  /* Sentinel */
```

(下页继续)

(续上页)

```
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

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

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom2.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_new = Custom_new,
    .tp_init = (initproc) Custom_init,
    .tp_dealloc = (destructor) Custom_dealloc,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom2",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom2(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
```

(下页继续)

(续上页)

```
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

该模块的新版本包含多处修改。

我们已经添加呢一个外部导入:

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

这包括提供被我们用来处理属性的声明，正如我们稍后所描述的。

现在 Custom 类型的 C 结构体中有三个数据属性，*first*、*last* 和 *number*。其中 *first* 和 *last* 变量是包含名字和姓氏的 Python 字符串。*number* 属性是一个 C 整数。

对象的结构将被相应地更新:

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

因为现在我们有数据需要管理，我们必须更加小心地处理对象的分配和释放。至少，我们需要有一个释放方法:

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

它会被赋值给 tp_dealloc 成员:

```
.tp_dealloc = (destructor) Custom_dealloc,
```

此方法会先清空两个 Python 属性的引用计数。Py_XDECREF() 可以正确处理参数为 NULL 的情况（这可能在 tp_new 中途失败时发生）。随后它将调用对象类型的 tp_free 成员（通过 Py_TYPE(self) 计算得到）来释放对象的内存。请注意对象类型可以不是 CustomType，因为对象可能是一个子类的实例。

备注: 上面需要强制转换 destructor 是因为我们定义了 Custom_dealloc 接受一个 CustomObject * 参数，但 tp_dealloc 函数指针预期接受一个 PyObject * 参数。如果不这样做，编译器将发出警告。这是 C 语言中面向对象的多态性!

我们希望确保头一个和末一个名称被初始化为空字符串，因此我们提供了一个 tp_new 实现:

```
static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}
```

并在 tp_new 成员中安装它:

```
.tp_new = Custom_new,
```

tp_new 处理句柄负责创建（而不是初始化）该类型的对象。它在 Python 中被暴露为 __new__() 方法。它不需要定义 tp_new 成员，实际上许多扩展类型会简单地重用 PyType_GenericNew()，就像上面 Custom 类型的第一个版本所做的那样。在此情况下，我们使用 tp_new 处理句柄来将 first 和 last 属性初始化为非 NULL 的默认值。

tp_new 将接受被实例化的类型（不要求为 CustomType，如果被实例化的是一个子类）以及在该类型被调用时传入的任何参数，并预期返回所创建的实例。tp_new 处理句柄总是接受位置和关键字参数，但它们总是会忽略这些参数，而将参数处理留给初始化（即 C 中的 tp_init 或 Python 中的 __init__ 函数）方法来执行。

备注： tp_new 不应显式地调用 tp_init，因为解释器会自行调用它。

tp_new 实现会调用 tp_alloc 槽位来分配内存:

```
self = (CustomObject *) type->tp_alloc(type, 0);
```

由于内存分配可能会失败，我们必须在继续执行之前检查 tp_alloc 结果确认其不为 NULL。

备注： 我们没有自行填充 tp_alloc 槽位。而是由 PyType_Ready() 通过从我们的基类继承来替我们填充它，其中默认为 object。大部分类型都是使用默认的分配策略。

备注： 如果您要创建一个协作式 tp_new (它会调用基类型的 tp_new 或 __new__())，那么你 不能在运行时尝试使用方法解析顺序来确定要调用的方法。必须总是静态地确定你要调用的类型，并直接调用它的 tp_new，或是通过 type->tp_base->tp_new。如果你不这样做，你的类型的同样继承自其它由 Python 定义的类的 Python 子类可能无法正常工作。(具体地说，你可能无法创建这样的子类的实例而是会引发 TypeError。)

我们还定义了一个接受参数来为我们的实例提供初始值的初始化函数:

```
static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    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;
}
```

通过填充 tp_init 槽位。

```
.tp_init = (initproc) Custom_init,
```

tp_init 槽位在 Python 中暴露为 __init__() 方法。它被用来在创建对象后对其进行初始化。初始化器总是接受位置和关键字参数，它们应当在成功时返回 0 而在出错时返回 -1。

不同于 tp_new 处理句柄，tp_init 不保证一定会被调用 (例如，在默认情况下 pickle 模块不会在未解封的实例上调用 __init__())。它还可能被多次调用。任何人都可以在我们的对象上调用 __init__() 方法。因此，我们在为属性赋新值时必须格外小心。例如像这样给 first 成员赋值:

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

但是这可能会有危险。我们的类型没有限制 first 成员的类型，因此它可以是任何种类的对象。它可以带有一个会执行尝试访问 first 成员的代码的析构器；或者该析构器可能会释放全局解释器锁 并让任意代码在其他线程中运行来访问和修改我们的对象。

为了保持谨慎并使我们避免这种可能性，我们几乎总是要在减少成员的引用计数之前给它们重新赋值。什么时候我们可以不必再这样做?

- 当我们明确知道引用计数大于 1 的时候;
- 当我们知道对象的释放[1] 既不会释放*GIL* 也不会导致任何对我们的类型的代码的回调的时候;
- 当减少一个 tp_dealloc 处理句柄内不支持循环垃圾回收的类型的引用计数的时候[2].

我们可能会想将我们的实例变量暴露为属性。有几种方式可以做到这一点。最简单的方式是定义成员的定义:

[1] 当我们知道该对象属于基本类型，如字符串或浮点数时情况就是如此。
[2] 在本示例中我们需要 tp_dealloc 处理句柄中的这一机制，因为我们的类型不支持垃圾回收。

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

并将定义放置到 tp_members 槽位中:

```
.tp_members = Custom_members,
```

每个成员的定义都有成员名称、类型、偏移量、访问旗标和文档字符串。请参阅下面的泛型属性管理 小节来了解详情。section below for details.

此方式的缺点之一是它没有提供限制可被赋值给 Python 属性的对象类型的办法。我们预期 first 和 last 的名称为字符串，但它们可以被赋值为任意 Python 对象。此外，这些属性还可以被删除，并将 C 指针设为 NULL。即使我们可以保证这些成员被初始化为非 NULL 值，如果这些属性被删除这些成员仍可被设为 NULL。

我们定义了一个单独的方法，Custom.name()，它将对象名称输出为 first 和 last 的拼接。

```
static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}
```

该方法以的实现形式是一个接受 Custom (或:class:*!Custom* 的子类) 实例作为第一个参数的 C 函数。方法总是接受一个实例作为第一个参数。方法往往也接受位置和关键字参数，但在本例中我们未接受任何参数也不需要接受位置参数元组或关键字参数字典。该方法等价于以下 Python 方法:

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

请注意我们必须检查 first 和 last 成员是否可能为 NULL。这是因为它们可以被删除，在此情况下它们会被设为 NULL。更好的做法是防止删除这些属性并将属性的值限制为字符串。我们将在下一节了解如何做到这一点。

现在我们已经定义好了方法，我们需要创建一个方法定义数组:

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

（请注意我们使用了 METH_NOARGS 旗标来指明该方法不准备接受除 *self* 以外的任何参数）

并将其赋给 tp_methods 槽位:

```
.tp_methods = Custom_methods,
```

最后，我们将使我们的类型可被用作派生子类的基类。我们精心地编写我们的方法以便它们不会随意假定被创建或使用的对象类型，所以我们需要做的就是将 Py_TPFLAGS_BASETYPE 添加到我们的类旗标定义中:

```
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
```

我们将 PyInit_custom() 重命名为 PyInit_custom2()，更新 PyModuleDef 结构体中的模块名称，并更新 PyTypeObject 结构体中的完整类名。

最后，我们更新我们的 setup.py 文件来生成新的模块。

```
from distutils.core import setup, Extension
setup(name="custom", version="1.0",
      ext_modules=[
         Extension("custom", ["custom.c"]),
         Extension("custom2", ["custom2.c"]),
         ])
```

2.2.3 提供对于数据属性的更精细控制

在本节中，我们将对 Custom 示例中 first 和 last 属性的设置进行更精细的控制。在我们上一版本的模块中，实例变量 first 和 last 可以被设为非字符串值甚至被删除。我们希望确保这些属性始终包含字符串。

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

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

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

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
```

(下页继续)

(续上页)

```
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi", 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 Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL}  /* Sentinel */
};

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

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

(下页继续)

(续上页)

```
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a string");
        return -1;
    }
    tmp = self->first;
    Py_INCREF(value);
    self->first = value;
    Py_DECREF(tmp);
    return 0;
}

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

static int
Custom_setlast(CustomObject *self, PyObject *value, void *closure)
{
    PyObject *tmp;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The last attribute value must be a string");
        return -1;
    }
    tmp = self->last;
    Py_INCREF(value);
    self->last = value;
    Py_DECREF(tmp);
    return 0;
}

static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL}  /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
```

(下页继续)

(续上页)

```
     "Return the name, combining the first and last name"
    },
    {NULL}  /* Sentinel */
};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom3.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_new = Custom_new,
    .tp_init = (initproc) Custom_init,
    .tp_dealloc = (destructor) Custom_dealloc,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
    .tp_getset = Custom_getsetters,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom3",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom3(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

为了更好地控制 `first` 和 `last` 属性，我们将使用自定义的读取器和设置器函数。以下就是用于读取和设置 `first` 属性的函数:

```
static PyObject *
Custom_getfirst(CustomObject *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
```

(下页继续)

(续上页)

```
}

static int
Custom_setfirst(CustomObject *self, PyObject *value, void *closure)
{
    PyObject *tmp;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a string");
        return -1;
    }
    tmp = self->first;
    Py_INCREF(value);
    self->first = value;
    Py_DECREF(tmp);
    return 0;
}
```

读取器函数接受一个 Custom 对象和一个“闭包”，后者是一个空指针。在本例中，该闭包将被忽略。(闭包支持将定义数据传递给读取器和设置器的高级用法。例如，这可以被用来允许一组获取器和设置器函数根据闭包中的数据来决定要读取或设置的属性)。

设置器函数接受传入 Custom 对象、新值和闭包。新值可能为 NULL，在这种情况下属性将被删除。在我们的设置器中，如果属性被删除或者如果其新值不是字符串则会引发一个错误。

我们创建一个 PyGetSetDef 结构体的数组:

```
static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL}  /* Sentinel */
};
```

并在 tp_getset 槽位中注册它:

```
.tp_getset = Custom_getsetters,
```

在 PyGetSetDef 结构体中的最后一项是上面提到的“闭包”。在本例中，我们没有使用闭包，因此我们只传入 NULL。

我们还移除了这些属性的成员定义:

```
static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL}  /* Sentinel */
};
```

我们还需要将 tp_init 处理句柄更新为只允许传入字符串[3]:

[3] 现在我们知道 first 和 last 成员都是字符串，因此也许我们可以对减少它们的引用计数不必太过小心，但是，我们还接受字符串子类的实例。即使释放普通字符串不会对我们的对象执行回调，我们也不能保证释放一个字符串子类的实例不会对我们的对象执行回调。

```
static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi", 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;
}
```

通过这些更改，我们能够确保 `first` 和 `last` 成员一定不为 `NULL` 以便我们能在几乎所有情况下移除 `NULL` 值检查。这意味着大部分 `Py_XDECREF()` 调用都可以被转换为 `Py_DECREF()` 调用。我们不能更改这些调用的唯一场合是在 `tp_dealloc` 实现中，那里这些成员的初始化有可能在 `tp_new` 中失败。

我们还重命名了模块初始化函数和初始化函数中的模块名称，就像我们之前所做的一样，我们还向 `setup.py` 文件添加了一个额外的定义。

2.2.4 支持循环垃圾回收

Python 具有一个可以标识不再需要的对象的*循环垃圾回收器 (GC)* 即使它们的引用计数并不为零。这种情况会在对象被循环引用时发生。例如，设想:

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

在这个例子中，我们创建了一个包含其自身的列表。当我们删除它的时候，它将仍然具有一个来自其本身的引用。它的引用计数并未降为零。幸运的是，Python 的循环垃圾回收器将最终发现该列表是无用的垃圾并释放它。

在 `Custom` 示例的第二个版本中，我们允许任意类型的对象存储到 `first` 或 `last` 属性中[4]。此外，在第二和第三个版本中，我们还允许子类化 `Custom`，并且子类可以添加任意属性。出于这两个原因中的任何一个，`Custom` 对象都可以加入循环:

```
>>> import custom3
>>> class Derived(custom3.Custom): pass
...
>>> n = Derived()
>>> n.some_attribute = n
```

[4] 而且，即使是将我们的属性限制为字符串实例，用户还是可以传入任意 `str` 子类因而仍能造成引用循环。

要允许一个加入引用循环的 Custom 实例能被循环 GC 正确检测和收集，我们的 Custom 类型需要填充两个额外的槽位并增加一个旗标来启用这些槽位：

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

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

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

static int
Custom_clear(CustomObject *self)
{
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}

static void
Custom_dealloc(CustomObject *self)
{
    PyObject_GC_UnTrack(self);
    Custom_clear(self);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}
```

(下页继续)

(续上页)

```
static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi", 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 Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL}  /* Sentinel */
};

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

static int
Custom_setfirst(CustomObject *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a string");
        return -1;
    }
    Py_INCREF(value);
    Py_CLEAR(self->first);
    self->first = value;
    return 0;
```

(下页继续)

(续上页)

```
}

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

static int
Custom_setlast(CustomObject *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The last attribute value must be a string");
        return -1;
    }
    Py_INCREF(value);
    Py_CLEAR(self->last);
    self->last = value;
    return 0;
}

static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL}  /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

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

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom4.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
    .tp_new = Custom_new,
```

(下页继续)

(续上页)

```
    .tp_init = (initproc) Custom_init,
    .tp_dealloc = (destructor) Custom_dealloc,
    .tp_traverse = (traverseproc) Custom_traverse,
    .tp_clear = (inquiry) Custom_clear,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
    .tp_getset = Custom_getsetters,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom4",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom4(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

首先，遍历方法让循环 GC 知道能够参加循环的子对象:

```
static int
Custom_traverse(CustomObject *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;
}
```

对于每个可以加入循环的子对象，我们都需要调用 `visit()` 函数，它会被传递给遍历方法。`visit()` 函数

接受该子对象和传递给遍历方法的额外参数 *arg* 作为其参数。它返回一个在其为非零值时必须被返回的整数值。

Python 提供了一个可自动调用 visit 函数的 `Py_VISIT()` 宏。使用 `Py_VISIT()`，我们可以最小化 `Custom_traverse` 中的准备工作量:

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

备注: `tp_traverse` 实现必须将其参数准确命名为 *visit* 和 *arg* 以便使用 `Py_VISIT()`。

第二，我们需要提供一个方法用来清除任何可以参加循环的子对象:

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

请注意 `Py_CLEAR()` 宏的使用。它是清除任意类型的数据属性并减少其引用计数的推荐的且安全的方式。如果你要选择在将属性设为 `NULL` 之间在属性上调用 `Py_XDECREF()`，则属性的析构器有可能会回调再次读取该属性的代码 (特别是如果存在引用循环的话)。

备注: 你可以通过以下写法来模拟 `Py_CLEAR()`:

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

无论如何，在删除属性时始终使用 Nevertheless, it is much easier and less error-prone to always use `Py_CLEAR()` 都是更简单且更不易出错的。请不要尝试以健壮性为代价的微小优化!

释放器 `Custom_dealloc` 可能会在清除属性时调用任意代码。这意味着循环 GC 可以在函数内部被触发。由于 GC 预期引用计数不为零，我们需要通过调用 `PyObject_GC_UnTrack()` 来让 GC 停止追踪相关的对象。下面是我们使用 `PyObject_GC_UnTrack()` 和 `Custom_clear` 重新实现的释放器:

```
static void
Custom_dealloc(CustomObject *self)
{
    PyObject_GC_UnTrack(self);
    Custom_clear(self);
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```

最后，我们将 `Py_TPFLAGS_HAVE_GC` 旗标添加到类旗标中:

```
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
```

这样就差不多了。如果我们编写了自定义的 tp_alloc 或 tp_free 处理句柄，则我们需要针对循环垃圾回收来修改它。大多数扩展都将使用自动提供的版本。

2.2.5 子类化其他类型

创建派生自现有类型的新类型是有可能的。最容易的做法是从内置类型继承，因为扩展可以方便地使用它所需要的 PyTypeObject。在不同扩展模块之间共享这些 PyTypeObject 结构体则是困难的。

在本例中我们将创建一个继承自内置 list 类型的 SubList 类型。这个新类型将完全兼容常规列表，但将拥有一个额外的 increment() 方法用于递增内部计数器的值：

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

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

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

static PyObject *
SubList_increment(SubListObject *self, PyObject *unused)
{
    self->state++;
    return PyLong_FromLong(self->state);
}

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

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

static PyTypeObject SubListType = {
    PyVarObject_HEAD_INIT(NULL, 0)
```

(下页继续)

(续上页)

```
    .tp_name = "sublist.SubList",
    .tp_doc = PyDoc_STR("SubList objects"),
    .tp_basicsize = sizeof(SubListObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_init = (initproc) SubList_init,
    .tp_methods = SubList_methods,
};

static PyModuleDef sublistmodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "sublist",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_sublist(void)
{
    PyObject *m;
    SubListType.tp_base = &PyList_Type;
    if (PyType_Ready(&SubListType) < 0)
        return NULL;

    m = PyModule_Create(&sublistmodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&SubListType);
    if (PyModule_AddObject(m, "SubList", (PyObject *) &SubListType) < 0) {
        Py_DECREF(&SubListType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

如你所见，此源代码与之前几节中的 `Custom` 示例非常相似。我们将逐一分析它们之间的主要区别。

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

派生类型对象的主要差异在于基类型的对象结构体必须是第一个值。基类型将已经在其结构体的开头包括了 `PyObject_HEAD()`。

当一个 Python 对象是 `SubList` 的实例时，它的 `PyObject *` 指针可以被安全地强制转换为 `PyListObject *` 和 `SubListObject *`:

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

(下页继续)

(续上页)

```
    return 0;
}
```

我们可以在上面看到如何将调用传递到基类型的 `__init__()` 方法。

这个模式在编写具有自定义 `tp_new` 和 `tp_dealloc` 成员的类型时很重要。`tp_new` 处理句柄不应为具有 `tp_alloc` 的对象实际分配内存，而是让基类通过调用自己的 `tp_new` 来处理它。

`PyTypeObject` 支持用 `tp_base` 指定类型的实体基类。由于跨平台编译器问题，你无法使用对 `PyList_Type` 的引用来直接填充该字段；它应当随后在模块初始化函数中完成:

```
PyMODINIT_FUNC
PyInit_sublist(void)
{
    PyObject* m;
    SubListType.tp_base = &PyList_Type;
    if (PyType_Ready(&SubListType) < 0)
        return NULL;

    m = PyModule_Create(&sublistmodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&SubListType);
    if (PyModule_AddObject(m, "SubList", (PyObject *) &SubListType) < 0) {
        Py_DECREF(&SubListType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

在调用 `PyType_Ready()` 之前，类型结构体必须已经填充 `tp_base` 槽位。当我们从现有类型派生时，它不需要将 `tp_alloc` 槽位填充为 `PyType_GenericNew()` -- 来自基类型的分配函数将会被继承。

在那之后，调用 `PyType_Ready()` 并将类型对象添加到模块中的过程与基本的 `Custom` 示例是一样的。

备注

2.3 定义扩展类型：已分类主题

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
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
```

(下页继续)

(续上页)

```
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                    or tp_reserved (Python 3) */
    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 */
    unsigned long tp_flags;

    const 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 */
    Py_ssize_t tp_weaklistoffset;

    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
    struct PyGetSetDef *tp_getset;
    // Strong reference on a heap type, borrowed reference on a static type
    struct _typeobject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    Py_ssize_t tp_dictoffset;
```

(下页继续)

(续上页)

```
    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;
    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6 */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;
} PyTypeObject;
```

这里有 很多方法。但是不要太担心，如果你要定义一个类型，通常只需要实现少量的方法。

正如你猜到的一样，我们正要一步一步详细介绍各种处理程序。因为有大量的历史包袱影响字段的排序，所以我们不会根据它们在结构体里定义的顺序讲解。通常非常容易找到一个包含你需要的字段的例子，然后改变值去适应你新的类型。

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

类型的名字 - 上一章提到过的，会出现在很多地方，几乎全部都是为了诊断目的。尝试选择一个好名字，对于诊断很有帮助。

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

这些字段告诉运行时在创造这个类型的新对象时需要分配多少内存。Python 为了可变长度的结构（想下：字符串，元组）有些内置支持，这是 `tp_itemsize` 字段存在的原由。这部分稍后解释。

```
const 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。

现在我们来看一下基本类型方法 - 大多数扩展类型将实现的方法。

2.3.1 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((PyObject *)obj);
}
```

如果你的类型支持垃圾回收，则析构器应当在清理任何成员字段之前调用 PyObject_GC_UnTrack()：

```
static void
newdatatype_dealloc(newdatatypeobject *obj)
{
    PyObject_GC_UnTrack(obj);
    Py_CLEAR(obj->other_obj);
    ...
    Py_TYPE(obj)->tp_free((PyObject *)obj);
}
```

一个重要的释放器函数实现要求是把所有未决异常放着不动。这很重要是因为释放器会被解释器频繁的调用，当栈异常退出时 (而非正常返回)，不会有任何办法保护释放器看到一个异常尚未被设置。此事释放器的任何行为都会导致额外增加的 Python 代码来检查异常是否被设置。这可能导致解释器的误导性错误。正确的保护方法是，在任何不安全的操作前，保存未决异常，然后在其完成后恢复。者可以通过 PyErr_Fetch() 和 PyErr_Restore() 函数来实现:

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

    if (self->my_callback != NULL) {
        PyObject *err_type, *err_value, *err_traceback;

        /* This saves the current exception state */
        PyErr_Fetch(&err_type, &err_value, &err_traceback);

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

        /* This restores the saved exception state */
        PyErr_Restore(err_type, err_value, err_traceback);

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

备注： 你能在释放器函数中安全执行的操作是有限的。首先，如果你的类型支持垃圾回收 (使用 tp_traverse 和/或 tp_clear)，对象的部分成员可以在调用 tp_dealloc 时被清空或终结。其次，在 tp_dealloc 中，你的对象将处于不稳定状态：它的引用计数等于零。任何对非琐碎对象或 API 的调用 (如上面的示例所做的) 最终都可能会再次调用 tp_dealloc，导致双重释放并发生崩溃。

从 Python 3.4 开始，推荐不要在 tp_dealloc 放复杂的终结代码，而是使用新的 tp_finalize 类型方法。

参见:

PEP 442 解释了新的终结方案。

2.3.2 对象展示

在 Python 中，有两种方式可以生成对象的文本表示: `repr()` 函数和 `str()` 函数。(`print()` 函数会直接调用 `str()`。) 这些处理程序都是可选的。

```
reprfunc tp_repr;
reprfunc tp_str;
```

`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 PyUnicode_FromFormat("Repr-ified_newdatatype{{size:%d}}",
                                obj->obj_UnderlyingDatatypePtr->size);
}
```

如果没有指定 `tp_repr` 处理句柄，解释器将提供一个使用类型的 `tp_name` 的表示形式以及对象的唯一标识值。

`tp_str` 处理句柄对于 `str()` 就如上述的 `tp_repr` 处理句柄对于 `repr()` 一样；也就是说，它会在当 Python 代码在你的对象的某个实例上调用 `str()` 时被调用。它的实现与 `tp_repr` 函数非常相似，但其结果字符串是供人类查看的。如果未指定 `tp_str`，则会使用 `tp_repr` 重句柄来代替。

下面是一个简单的例子:

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

2.3.3 属性管理

对于每个可支持属性操作的对象，相应的类型必须提供用于控制属性获取方式的函数。需要有一个能够检索属性的函数（如果定义了任何属性）还要有另一个函数负责设置属性（如果允许设置属性）。移除属性是一种特殊情况，在此情况下要传给处理句柄的新值为 `NULL`。

Python 支持两对属性处理句柄；一个支持属性操作的类型只需要实现其中一对的函数。两者的差别在于一对接受 `char*` 作为属性名称，而另一对则接受 `PyObject*`。每种类型都可以选择使用对于实现的便利性来说更有意义的那一对。

```
getattrfunc  tp_getattr;        /* char * version */
setattrfunc  tp_setattr;
/* ... */
getattrofunc tp_getattro;       /* PyObject * version */
setattrofunc tp_setattro;
```

如果访问一个对象的属性总是为简单操作（这将在下文进行解释），则有一些泛用实现可被用来提供 `PyObject*` 版本的属性管理函数。从 Python 2.2 开始对于类型专属的属性处理句柄的实际需要几乎已完全消失，尽管还存在着许多尚未理新为使用某种新的可选泛用机制的例子。

泛型属性管理

大多数扩展类型只使用 **简单**属性，那么，是什么让属性变得“简单”呢？只需要满足下面几个条件：

1. 当调用 `PyType_Ready()` 时，必须知道属性的名称。
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。

当 `PyType_Ready()` 被调用时，它会使用由类型对象所引用的三个表来创建要放置到类型对象的字典中的*descriptor*。每个描述器控制对实例对象的一个属性的访问。每个表都是可选的；如果三个表全都为 `NULL`，则该类型的实例将只有从它们的基础类型继承来的属性，并且还应当让 `tp_getattro` 和 `tp_setattro` 字段保持为 `NULL`，以允许由基础类型处理这些属性。

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;
```

如果 `tp_methods` 不为 `NULL`，则它必须指向一个由 `PyMethodDef` 结构体组成的数组。表中的每个条目都是该结构体的一个实例:

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

应当为该类型所提供的每个方法都应定义一个条目；从基类型继承来的方法无需定义条目。还需要在末尾加一个额外的条目；它是一个标记数组结束的哨兵条目。该哨兵条目的 `ml_name` 字段必须为 `NULL`。

第二个表被用来定义要直接映射到实例中的数据的属性。各种原始 C 类型均受到支持，并且访问方式可以为只读或读写。表中的结构体被定义为:

```
typedef struct PyMemberDef {
    const char *name;
    int         type;
    int         offset;
    int         flags;
    const 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.

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

常量	含意
`READONLY`	没有可写的
`PY_AUDIT_READ`	在读取之前发送一个 `object.__getattr__` 审计事件。

在 3.10 版更改: `RESTRICTED`, `READ_RESTRICTED` and `WRITE_RESTRICTED` are deprecated. However, `READ_RESTRICTED` is an alias for `PY_AUDIT_READ`, so fields that specify either `RESTRICTED` or `READ_RESTRICTED` will also raise an audit event.

使用 tp_members 表来构建用于运行时的描述器还有一个有趣的优点是任何以这种方式定义的属性都可以简单地通过在表中提供文本来设置一个相关联的文档字符串。一个应用程序可以使用自省 API 从类对象获取描述器，并使用其 __doc__ 属性来获取文档字符串。

与 tp_methods 表一样，需要有一个值为 NULL 的 ml_name 哨兵条目。

类型专属的属性管理

为了简单起见，这里只演示 char* 版本；name 形参的类型是 char* 和 PyObject* 风格接口之间的唯一区别。这个示例实际上做了与上面的泛用示例相同的事情，但没有使用在 Python 2.2 中增加的泛用支持。它解释了处理句柄函数是如何被调用的，因此如果你确实需要扩展它们的功能，你就会明白有什么是需要做的。

tp_getattr 处理句柄会在对象需要进行属性查找时被调用。它被调用的场合与一个类的 __getattr__() 方法要被调用的场合相同。

例如：

```
static PyObject *
newdatatype_getattr(newdatatypeobject *obj, char *name)
{
    if (strcmp(name, "data") == 0)
    {
        return PyLong_FromLong(obj->data);
    }

    PyErr_Format(PyExc_AttributeError,
                 "'%.50s' object has no attribute '%.400s'",
                 tp->tp_name, name);
    return NULL;
}
```

当调用类实例的 __setattr__() 或 __delattr__() 方法时会调用 tp_setattr 处理句柄。当需要删除一个属性时，第三个形参将为 NULL。下面是一个简单地引发异常的例子；如果这确实是你想要的，则 tp_setattr 处理句柄应当被设为 NULL。

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

2.3.4 对象比较

```
richcmpfunc tp_richcompare;
```

tp_richcompare 处理句柄会在需要进行比较时被调用。它类似于 富比较方法，例如 __lt__()，并会被 PyObject_RichCompare() 和 PyObject_RichCompareBool() 调用。

此函数被调用时将传入两个 Python 对象和运算符作为参数，其中运算符为 Py_EQ, Py_NE, Py_LE, Py_GE, Py_LT 或 Py_GT 之一。它应当使用指定的运算符来比较两个对象并在比较操作成功时返回 Py_True 或 Py_False，如果比较操作未被实现并应尝试其他对象比较方法时则返回 Py_NotImplemented，或者如果设置了异常则返回 NULL。

下面是一个示例实现，该数据类型如果内部指针的大小相等就认为是相等的：

```
static PyObject *
newdatatype_richcmp(PyObject *obj1, PyObject *obj2, int op)
{
    PyObject *result;
    int c, size1, size2;

    /* code to make sure that both arguments are of type
       newdatatype omitted */

    size1 = obj1->obj_UnderlyingDatatypePtr->size;
    size2 = obj2->obj_UnderlyingDatatypePtr->size;

    switch (op) {
    case Py_LT: c = size1 <  size2; break;
    case Py_LE: c = size1 <= size2; break;
    case Py_EQ: c = size1 == size2; break;
    case Py_NE: c = size1 != size2; break;
    case Py_GT: c = size1 >  size2; break;
    case Py_GE: c = size1 >= size2; break;
    }
    result = c ? Py_True : Py_False;
    Py_INCREF(result);
    return result;
 }
```

2.3.5 抽象协议支持

Python 支持多种 '抽象' 协议'；被提供来使用这些接口的专门接口说明请在 abstract 中查看。

这些抽象接口很多都是在 Python 实现开发的早期被定义的。特别地，数字、映射和序列协议从一开始就已经是 Python 的组成部分。其他协议则是后来添加的。对于依赖某些来自类型实现的处理句柄例程的协议来说，较旧的协议被定义为类型对象所引用的处理句柄的可选块。对于较新的协议来说在主类型对象中还有额外的槽位，并带有一个预设旗标位来指明存在该槽位并应当由解释器来检查。(此旗标位并不会指明槽位值非 NULL 的情况，可以设置该旗标来指明一个槽位的存在，但此本位仍可能保持未填充的状态。)

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

如果你希望你的对象的行为类似一个数字、序列或映射对象，那么你就要分别放置一个实现了 C 类型 PyNumberMethods, PySequenceMethods 或 PyMappingMethods, 的结构体的地址。你要负责将适当的值填入这些结构体。你可以在 Python 源代码发布版的 Objects 目录中找到这些对象各自的用法示例。

```
hashfunc tp_hash;
```

如果你选择提供此函数，则它应当为你的数据类型的实例返回一个哈希数值。下面是一个简单的示例:

```
static Py_hash_t
newdatatype_hash(newdatatypeobject *obj)
{
    Py_hash_t result;
    result = obj->some_size + 32767 * obj->some_number;
    if (result == -1)
        result = -2;
    return result;
}
```

Py_hash_t 是一个在宽度取决于具体平台的有符号整数类型。从 tp_hash 返回 -1 表示发生了错误，这就是为什么你应当注意避免在哈希运算成功时返回它，如上面所演示的。

```
ternaryfunc tp_call;
```

此函数会在“调用”你的数据类型实例时被调用，举例来说，如果 obj1 是你的数据类型的实例而 Python 脚本包含了 obj1('hello')，则将发起调用 tp_call 处理句柄。

此函数接受三个参数:

1. *self* 是作为调用目标的数据类型实例。如果调用是 obj1('hello')，则 *self* 为 obj1。
2. *args* 是包含调用参数的元组。你可以使用 PyArg_ParseTuple() 来提取参数。
3. *kwds* 是由传入的关键字参数组成的字典。如果它不为 NULL 且你支持关键字参数，则可使用 PyArg_ParseTupleAndKeywords() 来提取参数。如果你不想支持关键字参数而它为非 NULL 值，则会引发 TypeError 并附带一个提示不支持关键字参数的消息。

下面是一个演示性的 tp_call 实现:

```
static PyObject *
newdatatype_call(newdatatypeobject *self, PyObject *args, PyObject *kwds)
{
    PyObject *result;
    const char *arg1;
    const char *arg2;
    const char *arg3;

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

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

这些函数提供了对迭代器协议的支持。这两个处理句柄都只接受一个形参，即它们被调用时所使用的实例，并返回一个新的引用。当发生错误时，它们应设置一个异常并返回 NULL。tp_iter 对应于 Python __iter__() 方法，而 tp_iternext 对应于 Python __next__() 方法。

任何*iterable* 对象都必须实现 tp_iter 处理句柄，该处理句柄必须返回一个*iterator* 对象。下面是与 Python 类所应用的同一个指导原则:

- 对于可以支持多个独立迭代器的多项集（如列表和元组），则应当在每次调用 tp_iter 时创建并返回一个新的迭代器。
- 只能被迭代一次的对象（通常是由于迭代操作的附带影响，例如文件对象）可以通过返回一个指向自身的新引用来实现 tp_iter -- 并且为此还应当实现 tp_iternext 处理句柄。

任何*iterator* 对象都应当同时实现 tp_iter 和 tp_iternext。一个迭代器的 tp_iter 处理句柄应当返回一个指向该迭代器的新引用。它的 tp_iternext 处理句柄应当返回一个指向迭代操作的下一个对象的新引用，如果还有下一个对象的话。如果迭代已到达末尾，则 tp_iternext 可以返回 NULL 而不设置异常，或者也可以在返回 NULL 的基础上 额外设置 StopIteration；避免异常可以产生更好的性能。如果发生了实际的错误，则 tp_iternext 应当总是设置一个异常并返回 NULL。

2.3.6 弱引用支持

One of the goals of Python 弱引用实现的目标之一是允许任意类型参与弱引用机制而不会在重视性能的对象（例如数字）上产生额外开销。

参见:

`weakref` 模块的文档。

对于可弱引用的对象，扩展类型必须做两件事:

1. 在 C 对象结构体中包括一个专门用于弱引用机制的 `PyObject*` 字段。该对象的构造器应当让它保持为 `NULL` (当使用默认的 `tp_alloc` 时这将会自动设置)。
2. 将 `tp_weaklistoffset` 类型成员设置为 C 对象结构体中上述字段的偏移量，这样解释器就能知道如何访问和修改该字段。

具体来说，下面是一个微小的对象结构体如何被增强为具有所需的字段:

```
typedef struct {
    PyObject_HEAD
    PyObject *weakreflist;  /* List of weak references */
} TrivialObject;
```

以及在静态声明的类型对象中的相应成员:

```
static PyTypeObject TrivialType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    /* ... other members omitted for brevity ... */
    .tp_weaklistoffset = offsetof(TrivialObject, weakreflist),
};
```

唯一的额外补充是如果字段不为 `NULL` 则 `tp_dealloc` 需要清除任何弱引用 (通过调用 `PyObject_ClearWeakRefs()`)。:

```
static void
Trivial_dealloc(TrivialObject *self)
{
    /* Clear weakrefs first before calling any destructors */
    if (self->weakreflist != NULL)
        PyObject_ClearWeakRefs((PyObject *) self);
    /* ... remainder of destruction code omitted for brevity ... */
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```

2.3.7 更多建议

为了学习如何为你的新数据类型实现任何特定方法，请获取*CPython* 源代码。进入 `Objects` 目录，然后在 C 源文件中搜索 `tp_` 加上你想要的函数 (例如，`tp_richcompare`)。你将找到你想要实现的函数的例子。

当你需要验证一个对象是否为你实现的类型的具体实例时，请使用 `PyObject_TypeCheck()` 函数。它的一个用法示例如下:

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

参见:

下载 CPython 源代码版本。 https://www.python.org/downloads/source/

GitHub 上开发 CPython 源代码的 CPython 项目。 https://github.com/python/cpython

2.4 构建 C/C++ 扩展

一个 CPython 的 C 扩展是一个共享库 (例如一个 Linux 上的 `.so`，或者 Windows 上的 `.pyd`)，其会导出一个*初始化函数*。

为了可导入，共享库必须在 `PYTHONPATH` 中有效，且必须命名遵循模块名字，通过适当的扩展。当使用 distutils 时，会自动生成正确的文件名。

初始化函数的声明如下:

PyObject ***PyInit_modulename**(void)

It returns either a fully initialized module, or a `PyModuleDef` instance. See initializing-modules for details.

对于仅有 ASCII 编码的模块名，函数必须是 `PyInit_<modulename>`，将 `<modulename>` 替换为模块的名字。当使用 multi-phase-initialization 时，允许使用非 ASCII 编码的模块名。此时初始化函数的名字是 `PyInitU_<modulename>`，而 `<modulename>` 需要用 Python 的 *punycode* 编码，连字号需替换为下划线。在 Python 里:

```
def initfunc_name(name):
    try:
        suffix = b'_' + name.encode('ascii')
    except UnicodeEncodeError:
        suffix = b'U_' + name.encode('punycode').replace(b'-', b'_')
    return b'PyInit' + suffix
```

可以在一个动态库里导出多个模块，通过定义多个初始化函数。而导入他们需要符号链接或自定义导入器，因为缺省时只有对应了文件名的函数才会被发现。查看 "*一个库里的多模块*" 章节，在 **PEP 489** 了解更多细节。

2.4.1 使用 distutils 构建 C 和 C++ 扩展

扩展模块可以用 distutils 来构建，这是 Python 自带的。distutils 也支持创建二进制包，用户无需编译器而 distutils 就能安装扩展。

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。

在文件 setup.py 里，所有动作的入口通过 setup 函数。该函数可以接受可变数量个关键字参数，上面的例子只使用了一个子集。特别需要注意的例子指定了构建包的元信息，以及指定了包内容。通常一个包会包括多个模块，就像 Python 的源码模块、文档、子包等。请参数 distutils 的文档，在 distutils-index 来了解更多 distutils 的特性；本章节只解释构建扩展模块的部分。

通常预计算参数给 setup()，想要更好的结构化驱动脚本。有如如上例子函数 setup() 的 ext_modules 参数是一列扩展模块，每个是一个 Extension 类的实例。例子中的实例定义了扩展命名为 demo，从单一源码文件构建 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])
```

例子中函数 setup() 在调用时额外传递了元信息，是推荐发布包构建时的内容。对于这个扩展，其指定了预处理器定义，include 目录，库目录，库。依赖于编译器，distutils 还会用其他方式传递信息给编译器。例如在 Unix 上，结果是如下编译命令

```
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。

2.4.2 发布你的扩展模块

当一个扩展已经成功地被构建时，有三种方式来使用它。

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
```

有些情况下，需要在源码发布包里包含额外的文件；这通过 `MANIFEST.in` 文件实现，查看 manifest 了解细节。

如果源码发行包被成功地构建，维护者还可以创建二进制发行包。取决于具体平台，以下命令中的一个可以用来完成此任务

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

2.5 在 Windows 上构建 C 和 C++ 扩展

这一章简要介绍了如何使用 Microsoft Visual C++ 创建 Python 的 Windows 扩展模块，然后再提供有关其工作机理的详细背景信息。这些说明材料同时适用于 Windows 程序员学习构建 Python 扩展以及 Unix 程序员学习如何生成在 Unix 和 Windows 上均能成功构建的软件。

鼓励模块作者使用 distutils 方式来构建扩展模块，而不使用本节所描述的方式。你仍将需要使用 C 编译器来构建 Python；通常为 Microsoft Visual C++。

备注： 这一章提及了多个包括已编码 Python 版本号的文件名。这些文件名以显示为 `XY` 的版本号来代表；在实践中，`'X'` 将为你所使用的 Python 发布版的主版本号而 `'Y'` 将为次版本号。例如，如果你所使用的是 Python 2.2.1，`XY` 将为 `22`。

2.5.1 菜谱式说明

在 Windows 和 Unix 上构建扩展模块都有两种方式：使用 `distutils` 包来控制构建过程，或者全手动操作。distutils 方式适用于大多数扩展；使用 `distutils` 构建和打包扩展模块的文档见 distutils-index。如果你发现你确实需要手动操作，那么研究一下 winsound 标准库模块的项目文件可能会很有帮助。

2.5.2 Unix 和 Windows 之间的差异

Unix 和 Windows 对于代码的运行时加载使用了完全不同的范式。在你尝试构建可动态加载的模块之前，要先了解你所用系统是如何工作的。

在 Unix 中，一个共享对象 (`.so`) 文件中包含将由程序来使用的代码，也包含在程序中可被找到的函数名称和数据。当文件被合并到程序中时，对在文件代码中这些函数和数据的全部引用都会被改为指向程序中函数和数据在内存中所放置的实际位置。这基本上是一个链接操作。

在 Windows 中，一个动态链接库 (`.dll`) 文件中没有悬挂的引用。而是通过一个查找表执行对函数或数据的访问。因此在运行时 DLL 代码不必在运行时进行修改；相反地，代码已经使用了 DLL 的查找表，并且在运行时查找表会被修改以指向特定的函数和数据。

在 Unix 中，只存在一种库文件 (.a)，它包含来自多个对象文件 (.o) 的代码。在创建共享对象文件 (.so) 的链接阶段，链接器可能会发现它不知道某个标识符是在哪里定义的。链接器将在各个库的对象文件中查找它；如果找到了它，链接器将会包括来自该对象文件的所有代码。

在 Windows 中，存在两种库类型，静态库和导入库 (扩展名都是 .lib)。静态库类似于 Unix 的 .a 文件；它包含在必要时可被包括的代码。导入库基本上仅用于让链接器能确保特定标识符是合法的，并且将在 DLL 被加载时出现于程序中。这样链接器可使用来自导入库的信息构建查找表以便使用未包括在 DLL 中的标识符。当一个应用程序或 DLL 被链接时，可能会生成一个导入库，它将需要被用于应用程序或 DLL 中未来所有依赖于这些符号的 DLL。

假设你正在编译两个动态加载模块 B 和 C，它们应当共享另一个代码块 A。在 Unix 上，你 不应将 A.a 传给链接器作为 B.so 和 C.so；那会导致它被包括两次，这样 B 和 C 将分别拥有它们自己的副本。在 Windows 上，编译 A.dll 将同时编译 A.lib。你 应当将 A.lib 传给链接器用于 B 和 C。A.lib 并不包含代码；它只包含将在运行时被用于访问 A 的代码的信息。

在 Windows 上，使用导入库有点像是使用 import spam；它让你可以访问 spam 中的名称，但并不会创建一个单独副本。在 Unix 上，链接到一个库更像是 from spam import *；它会创建一个单独副本。

2.5.3 DLL 的实际使用

Windows Python 是在 Microsoft Visual C++ 中构建的；使用其他编译器可能会也可能不会工作。本节的其余部分是针对 MSVC++ 的。

当在 Windows 中创建 DLL 时，你必须将 pythonXY.lib 传给链接器。要编译两个 DLL，spam 和 ni (会使用 spam 中找到的 C 函数)，你应当使用以下命令:

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

第一条命令创建了三个文件: spam.obj, spam.dll 和 spam.lib。Spam.dll 不包含任何 Python 函数 (例如 PyArg_ParseTuple())，但它通过 pythonXY.lib 可以知道如何找到所需的 Python 代码。

第二条命令创建了 ni.dll (以及 .obj 和 .lib)，它知道如何从 spam 以及 Python 可执行文件中找到所需的函数。

不是每个标识符都会被导出到查找表。如果你想要任何其他模块（包括 Python）都能看到你的标识符，你必须写上 _declspec(dllexport)，就如在 void _declspec(dllexport) initspam(void) 或 PyObject _declspec(dllexport) *NiGetSpamData(void) 中一样。

Developer Studio 会添加很多你并不真正需要的导入库，命名你的可执行文件大小增加约 100K。要摆脱它们，请使用项目设置对话框中的链接选项卡指定 忽略默认库。将正确的 msvcrtxx.lib 添加到库列表中。

CHAPTER 3

在更大的应用程序中嵌入 CPython 运行时

有时，不是要创建在 Python 解释器中作为主应用程序运行的扩展，而是希望将 CPython 运行时嵌入到更大的应用程序中。本节介绍了成功完成此操作所涉及的一些细节。

3.1 在其它应用程序嵌入 Python

前几章讨论了如何对 Python 进行扩展，也就是如何用 C 函数库扩展 Python 的功能。反过来也是可以的：将 Python 嵌入到 C/C++ 应用程序中丰富其功能。这种嵌入可以让应用程序用 Python 来实现某些功能，而不是用 C 或 C++ 。用途会有很多；比如允许用户用 Python 编写一些脚本，以便定制应用程序满足需求。如果某些功能用 Python 编写起来更为容易，那么开发人员自己也能这么干。

Python 的嵌入类似于扩展，但不完全相同。不同之处在于，扩展 Python 时应用程序的主程序仍然是 Python 解释器，而嵌入 Python 时的主程序可能与 Python 完全无关——而是应用程序的某些部分偶尔会调用 Python 解释器来运行一些 Python 代码。

因此，若要嵌入 Python，就要提供自己的主程序。此主程序要做的事情之一就是初始化 Python 解释器。至少得调用函数 `Py_Initialize()`。还有些可选的调用可向 Python 传递命令行参数。之后即可从应用程序的任何地方调用解释器了。

调用解释器的方式有好几种：可向 `PyRun_SimpleString()` 传入一个包含 Python 语句的字符串，也可向 `PyRun_SimpleFile()` 传入一个 stdio 文件指针和一个文件名（仅在错误信息中起到识别作用）。还可以调用前面介绍过的底层操作来构造并使用 Python 对象。

参见:

c-api-index 本文详细介绍了 Python 的 C 接口。这里有大量必要的信息。

3.1.1 高层次的嵌入

最简单的 Python 嵌入形式就是采用非常高层的接口。该接口的目标是只执行一段 Python 脚本，而无需与应用程序直接交互。比如以下代码可以用来对某个文件进行一些操作。

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

int
main(int argc, char *argv[])
{
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]\n");
        exit(1);
    }
    Py_SetProgramName(program);  /* optional but recommended */
    Py_Initialize();
    PyRun_SimpleString("from time import time,ctime\n"
                       "print('Today is', ctime(time()))\n");
    if (Py_FinalizeEx() < 0) {
        exit(120);
    }
    PyMem_RawFree(program);
    return 0;
}
```

在 `Py_Initialize()` 之前，应该先调用 `Py_SetProgramName()` 函数，以便向解释器告知 Python 运行库的路径。接下来，`Py_Initialize()` 会初始化 Python 解释器，然后执行硬编码的 Python 脚本，打印出日期和时间。之后，调用 `Py_FinalizeEx()` 关闭解释器，程序结束。在真实的程序中，可能需要从其他来源获取 Python 脚本，或许是从文本编辑器例程、文件，或者某个数据库。利用 `PyRun_SimpleFile()` 函数可以更好地从文件中获取 Python 代码，可省去分配内存空间和加载文件内容的麻烦。

3.1.2 突破高层次嵌入的限制：概述

高级接口能从应用程序中执行任何 Python 代码，但至少交换数据可说是相当麻烦的。如若需要交换数据，应使用较低级别的调用。几乎可以实现任何功能，代价是得写更多的 C 代码。

应该注意，尽管意图不同，但扩展 Python 和嵌入 Python 的过程相当类似。前几章中讨论的大多数主题依然有效。为了说明这一点，不妨来看一下从 Python 到 C 的扩展代码到底做了什么：

1. 将 Python 的数据转换为 C 格式，
2. 用转换后的数据执行 C 程序的函数调用，
3. 将调用返回的数据从 C 转换为 Python 格式。

嵌入 Python 时，接口代码会这样做：

1. 将 C 数据转换为 Python 格式，
2. 用转换后的数据执行对 Python 接口的函数调用，
3. 将调用返回的数据从 Python 转换为 C 格式。

可见只是数据转换的步骤交换了一下顺序，以顺应跨语言的传输方向。唯一的区别是在两次数据转换之间调用的函数不同。在执行扩展时，调用一个 C 函数，而执行嵌入时调用的是个 Python 函数。

本文不会讨论如何将数据从 Python 转换到 C 去，反之亦然。另外还假定读者能够正确使用引用并处理错误。由于这些地方与解释器的扩展没有区别，请参考前面的章节以获得所需的信息。

3.1.3 只做嵌入

第一个程序的目标是执行 Python 脚本中的某个函数。就像高层次接口那样，Python 解释器并不会直接与应用程序进行交互（但下一节将改变这一点）。

要运行 Python 脚本中定义的函数，代码如下：

```
#define PY_SSIZE_T_CLEAN
#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 = PyUnicode_DecodeFSDefault(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 = PyLong_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", PyLong_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;
    }
    if (Py_FinalizeEx() < 0) {
        return 120;
    }
    return 0;
}
```

上述代码先利用 `argv[1]` 加载 Python 脚本，再调用 `argv[2]` 指定的函数。函数的整数参数是 `argv` 数组中的其余值。如果编译并链接该程序（此处将最终的可执行程序称作 **call**），并用它执行一个 Python 脚本，例如：

```
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
```

尽管相对其功能而言，该程序体积相当庞大，但大部分代码是用于 Python 和 C 之间的数据转换，以及报告错误。嵌入 Python 的有趣部分从此开始：

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

初始化解释器之后，则用 `PyImport_Import()` 加载脚本。此函数的参数需是个 Python 字符串，一个用 `PyUnicode_FromString()` 数据转换函数构建的字符串。

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

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

脚本一旦加载完毕，就会用 PyObject_GetAttrString() 查找属性名称。如果名称存在，并且返回的是可调用对象，即可安全地视其为函数。然后程序继续执行，照常构建由参数组成的元组。然后用以下方式调用 Python 函数：

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

当函数返回时，pValue 要么为 NULL，要么包含对函数返回值的引用。请确保用完后释放该引用。

3.1.4 对嵌入 Python 功能进行扩展

到目前为止，嵌入的 Python 解释器还不能访问应用程序本身的功能。Python API 通过扩展嵌入解释器实现了这一点。也就是说，用应用程序提供的函数对嵌入的解释器进行扩展。虽然听起来有些复杂，但也没那么糟糕。只要暂时忘记是应用程序启动了 Python 解释器。而把应用程序看作是一堆子程序，然后写一些胶水代码让 Python 访问这些子程序，就像编写普通的 Python 扩展程序一样。例如：

```
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 PyLong_FromLong(numargs);
}

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

static PyModuleDef EmbModule = {
    PyModuleDef_HEAD_INIT, "emb", NULL, -1, EmbMethods,
    NULL, NULL, NULL, NULL
};

static PyObject*
PyInit_emb(void)
{
    return PyModule_Create(&EmbModule);
}
```

在 main() 函数之前插入上述代码。并在调用 Py_Initialize() 之前插入以下两条语句：

```
numargs = argc;
PyImport_AppendInittab("emb", &PyInit_emb);
```

这两行代码初始化了 numargs 变量，并使嵌入式 Python 解释器可以访问 emb.numargs() 函数。通过这些扩展，Python 脚本可以执行以下操作

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

在真实的应用程序中，这种方法将把应用的 API 暴露给 Python 使用。

3.1.5 在 C++ 中嵌入 Python

还可以将 Python 嵌入到 C++ 程序中去；确切地说，实现方式将取决于 C++ 系统的实现细节；一般需用 C++ 编写主程序，并用 C++ 编译器来编译和链接程序。不需要用 C++ 重新编译 Python 本身。

3.1.6 在类 Unix 系统中编译和链接

为了将 Python 解释器嵌入应用程序，找到正确的编译参数传给编译器 (和链接器) 并非易事，特别是因为 Python 加载的库模块是以 C 动态扩展（`.so` 文件）的形式实现的。

为了得到所需的编译器和链接器参数，可执行 `pythonX.Y-config` 脚本，它是在安装 Python 时生成的（也可能存在 `python3-config` 脚本）。该脚本有几个参数，其中以下几个参数会直接有用：

- `pythonX.Y-config --cflags` 将给出建议的编译参数。

```
$ /opt/bin/python3.11-config --cflags
-I/opt/include/python3.11 -I/opt/include/python3.11 -Wsign-compare  -DNDEBUG -g -
↪fwrapv -O3 -Wall
```

- `pythonX.Y-config --ldflags --embed` 将给出在链接时建议的旗标:

```
$ /opt/bin/python3.11-config --ldflags --embed
-L/opt/lib/python3.11/config-3.11-x86_64-linux-gnu -L/opt/lib -lpython3.11 -
↪lpthread -ldl  -lutil -lm
```

备注： 为了避免多个 Python 安装版本引发混乱（特别是在系统安装版本和自己编译版本之间），建议用 `pythonX.Y-config` 指定绝对路径，如上例所述。

如果上述方案不起作用（不能保证对所有 Unix 类平台都生效；欢迎提出 bug 报告），就得阅读系统关于动态链接的文档，并检查 Python 的 `Makefile`（用 `sysconfig.get_makefile_filename()` 找到所在位置）和编译参数。这时 `sysconfig` 模块会是个有用的工具，可用编程方式提取需组合在一起的配置值。比如：

```
>>> 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。

... 具有以下含义：

- 交互式终端中输入特殊代码行时默认的 Python 提示符，包括：缩进的代码块，成对的分隔符之内（圆括号、方括号、花括号或三重引号），或是指定一个装饰器之后。
- `Ellipsis` 内置常量。

2to3 把 Python 2.x 代码转换为 Python 3.x 代码的工具，通过解析源码，遍历解析树，处理绝大多数检测到的不兼容问题。

2to3 包含在标准库中，模块名为 `lib2to3`；提供了独立入口点 `Tools/scripts/2to3`。详见 2to3-reference。

abstract base class -- 抽象基类 抽象基类简称 ABC，是对*duck-typing* 的补充，它提供了一种定义接口的新方式，相比之下其他技巧例如 `hasattr()` 显得过于笨拙或有微妙错误（例如使用 魔术方法）。ABC 引入了虚拟子类，这种类并非继承自其他类，但却仍能被 `isinstance()` 和 `issubclass()` 所认可；详见 `abc` 模块文档。Python 自带许多内置的 ABC 用于实现数据结构（在 `collections.abc` 模块中）、数字（在 `numbers` 模块中）、流（在 `io` 模块中）、导入查找器和加载器（在 `importlib.abc` 模块中）。你可以使用 `abc` 模块来创建自己的 ABC。

annotation -- 标注 关联到某个变量、类属性、函数形参或返回值的标签，被约定作为类型注解 来使用。

局部变量的标注在运行时不可访问，但全局变量、类属性和函数的标注会分别存放模块、类和函数的 `__annotations__` 特殊属性中。

参见*variable annotation*, *function annotation*, **PEP 484** 和 **PEP 526**, 对此功能均有介绍。另请参见 annotations-howto 了解使用标注的最佳实践。

argument -- 参数 在调用函数时传给*function* （或*method* ）的值。参数分为两种：

- *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。

另参见*parameter* 术语表条目，常见问题中 参数与形参的区别以及 **PEP 362**。

asynchronous context manager -- 异步上下文管理器 此种对象通过定义 `__aenter__()` 和 `__aexit__()` 方法来对 `async with` 语句中的环境进行控制。由 **PEP 492** 引入。

asynchronous generator -- 异步生成器 返回值为*asynchronous generator iterator* 的函数。它与使用 `async def` 定义的协程函数很相似，不同之处在于它包含 `yield` 表达式以产生一系列可在 `async for` 循环中使用的值。

此术语通常是指异步生成器函数，但在某些情况下则可能是指 *异步生成器迭代器*。如果需要清楚表达具体含义，请使用全称以避免歧义。

一个异步生成器函数可能包含 `await` 表达式或者 `async for` 以及 `async with` 语句。

asynchronous generator iterator -- 异步生成器迭代器 *asynchronous generator* 函数所创建的对象。

此对象属于*asynchronous iterator*，当使用 `__anext__()` 方法调用时会返回一个可等待对象来执行异步生成器函数的函数体直到下一个 `yield` 表达式。

每个 `yield` 会临时暂停处理，记住当前位置执行状态（包括局部变量和挂起的 try 语句）。当该 *异步生成器迭代器* 通过 `__anext__()` 所返回的其他可等待对象有效恢复时，它会从离开位置继续执行。参见 **PEP 492** 和 **PEP 525**。

asynchronous iterable -- 异步可迭代对象 一个可以在 `async for` 语句中使用的对象。必须通过它的 `__aiter__()` 方法返回一个*asynchronous iterator*。由 **PEP 492** 引入。

asynchronous iterator -- 异步迭代器 一个实现了 `__aiter__()` 和 `__anext__()` 方法的对象。`__anext__()` 必须返回一个*awaitable* 对象。`async for` 会处理异步迭代器的 `__anext__()` 方法所返回的可等待对象直到其引发一个 `StopAsyncIteration` 异常。由 **PEP 492** 引入。

attribute -- 属性 关联到一个对象的值，通常使用点号表达式按名称来引用。举例来说，如果对象 *o* 具有属性 *a* 则可以用 *o.a* 来引用它。

如果对象允许，将未被定义为 identifiers 的非标识名称用作一个对象的属性也是可以的，例如使用 `setattr()`。这样的属性将无法使用点号表达式来访问，而是必须通过 `getattr()` 来获取。

awaitable -- 可等待对象 一个可在 `await` 表达式中使用的对象。可以是*coroutine* 或是具有 `__await__()` 方法的对象。参见 **PEP 492**。

BDFL “终身仁慈独裁者”的英文缩写，即 Guido van Rossum，Python 的创造者。

binary file -- 二进制文件 *file object* 能够读写*字节类对象*。二进制文件的例子包括以二进制模式（`'rb'`, `'wb'` or `'rb+'`）打开的文件、`sys.stdin.buffer`、`sys.stdout.buffer` 以及 `io.BytesIO` 和 `gzip.GzipFile` 的实例。

另请参见*text file* 了解能够读写 `str` 对象的文件对象。

borrowed reference -- 借入引用 在 Python 的 C API 中，借用引用是指一种对象引用，使用该对象的代码并不持有该引用。如果对象被销毁则它就会变成一个悬空指针。例如，垃圾回收器可以移除对象的最后一个*strong reference* 来销毁它。

推荐在*borrowed reference* 上调用 `Py_INCREF()` 以将其原地转换为*strong reference*，除非是当该对象无法在借入引用的最后一次使用之前被销毁。`Py_NewRef()` 函数可以被用来创建一个新的*strong reference*。

bytes-like object -- 字节类对象 支持 bufferobjects 并且能导出 C-*contiguous* 缓冲的对象。这包括所有 `bytes`、`bytearray` 和 `array.array` 对象，以及许多普通 `memoryview` 对象。字节类对象可在多种二进制数据操作中使用；这些操作包括压缩、保存为二进制文件以及通过套接字发送等。

某些操作需要可变的二进制数据。这种对象在文档中常被称为 “可读写字节类对象”。可变缓冲对象的例子包括 `bytearray` 以及 `bytearray` 的 `memoryview`。其他操作要求二进制数据存放于不可变对象 (” 只读字节类对象”)；这种对象的例子包括 `bytes` 以及 `bytes` 对象的 `memoryview`。

bytecode -- 字节码 Python 源代码会被编译为字节码，即 CPython 解释器中表示 Python 程序的内部代码。字节码还会缓存在 `.pyc` 文件中，这样第二次执行同一文件时速度更快（可以免去将源码重新编译为字节码）。这种” 中间语言” 运行在根据字节码执行相应机器码的*virtual machine* 之上。请注意不同 Python 虚拟机上的字节码不一定通用，也不一定能在不同 Python 版本上兼容。

字节码指令列表可以在 dis 模块的文档中查看。

callable -- 可调用对象 可调用对象就是可以执行调用运算的对象，并可能附带一组参数 (参见*argument*)，使用以下语法:

```
callable(argument1, argument2, argumentN)
```

function，还可扩展到*method* 等，就属于可调用对象。实现了 `__call__()` 方法的类的实例也属于可调用对象。

callback -- 回调 一个作为参数被传入以用以在未来的某个时刻被调用的子例程函数。

class 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。

class variable -- 类变量 在类中定义的变量，并且仅限在类的层级上修改 (而不是在类的实例中修改)。

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 An object which controls the environment seen in a `with` statement by defining `__enter__()` and `__exit__()` methods. See **PEP 343**.

context variable -- 上下文变量 一种根据其所属的上下文可以具有不同的值的变量。这类似于在线程局部存储中每个执行线程可以具有不同的变量值。不过，对于上下文变量来说，一个执行线程中可能会有多个上下文，而上下文变量的主要用途是对并发异步任务中变量进行追踪。参见 `contextvars`。

contiguous -- 连续 一个缓冲如果是 *C 连续*或 *Fortran 连续*就会被认为是连续的。零维缓冲是 C 和 Fortran 连续的。在一维数组中，所有条目必须在内存中彼此相邻地排列，采用从零开始的递增索引顺序。在多维 C-连续数组中，当按内存地址排列时用最后一个索引访问条目时速度最快。但是在 Fortran 连续数组中则是用第一个索引最快。

coroutine -- 协程 协程是子例程的更一般形式。子例程可以在某一点进入并在另一点退出。协程则可以在许多不同的点上进入、退出和恢复。它们可通过 `async def` 语句来实现。参见 **PEP 492**。

coroutine function -- 协程函数 返回一个*coroutine* 对象的函数。协程函数可通过 `async def` 语句来定义，并可能包含 `await`、`async for` 和 `async with` 关键字。这些特性是由 **PEP 492** 引入的。

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(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

同样的概念也适用于类，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 任何定义了 `__get__()`, `__set__()` 或 `__delete__()` 方法的对象。当一个类属性为描述器时，它的特殊绑定行为就会在属性查找时被触发。通常情况下，使用 *a.b* 来获取、设置或删除一个属性时会在 *a* 的类字典中查找名称为 *b* 的对象，但如果 *b* 是一个描述器，则会调用对应的描述器方法。理解描述器的概念是更深层次理解 Python 的关键，因为这是许多重要特性的基础，包括函数、方法、属性、类方法、静态方法以及对超类的引用等等。

有关描述器的方法的更多信息，请参阅 descriptors 或 描述器使用指南。

dictionary -- 字典 一个关联数组，其中的任意键都映射到相应的值。键可以是任何具有 `__hash__()` 和 `__eq__()` 方法的对象。在 Perl 语言中称为 hash。

dictionary comprehension -- 字典推导式 处理一个可迭代对象中的所有或部分元素并返回结果字典的一种紧凑写法。`results = {n: n ** 2 for n in range(10)}` 将生成一个由键 `n` 到值 `n ** 2` 的映射构成的字典。参见 comprehensions。

dictionary view -- 字典视图 从 `dict.keys()`, `dict.values()` 和 `dict.items()` 返回的对象被称为字典视图。它们提供了字典条目的一个动态视图，这意味着当字典改变时，视图也会相应改变。要将字典视图强制转换为真正的列表，可使用 `list(dictview)`。参见 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 可以求出某个值的语法单元。换句话说，一个表达式就是表达元素例如字面值、名称、属性访问、运算符或函数调用的汇总，它们最终都会返回一个值。与许多其他语言不同，并非所有语言构件都是表达式。还存在不能被用作表达式的*statement*，例如 `while`。赋值也是属于语句而非表达式。

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。

f-string -- f-字符串 带有 `'f'` 或 `'F'` 前缀的字符串字面值通常被称为“f-字符串”即 格式化字符串字面值的简写。参见 **PEP 498**。

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。

实际上共有三种类别的文件对象: 原始二进制文件, 缓冲二进制文件 以及文本文件。它们的接口定义均在 `io` 模块中。创建文件对象的规范方式是使用 `open()` 函数。

file-like object -- 文件类对象 *file object* 的同义词。

filesystem encoding and error handler -- 文件系统编码格式与错误处理句柄 Python 用来从操作系统解码字节串和向操作系统编码 Unicode 的编码格式与错误处理句柄。

文件系统编码格式必须保证能成功解码长度在 128 以下的所有字节串。如果文件系统编码格式无法提供此保证，则 API 函数可能会引发 `UnicodeError`。

`sys.getfilesystemencoding()` 和 `sys.getfilesystemencodeerrors()` 函数可被用来获取文件系统编码格式与错误处理句柄。

filesystem encoding and error handler 是在 Python 启动时通过 `PyConfig_Read()` 函数来配置的：请参阅 `PyConfig` 的 `filesystem_encoding` 和 `filesystem_errors` 等成员。

另请参见*locale encoding*。

finder -- 查找器 一种会尝试查找被导入模块的*loader* 的对象。

从 Python 3.3 起存在两种类型的查找器: *元路径查找器* 配合 `sys.meta_path` 使用，以及*path entry finders* 配合 `sys.path_hooks` 使用。

更多详情可参见 **PEP 302**, **PEP 420** 和 **PEP 451**。

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。

function annotation -- 函数标注 即针对函数形参或返回值的*annotation* 。

函数标注通常用于*类型提示*：例如以下函数预期接受两个 `int` 参数并预期返回一个 `int` 值:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

函数标注语法的详解见 function 一节。

参见*variable annotation* 和 **PEP 484**，其中描述了此功能。另请参阅 annotations-howto 以了解使用标的最佳实践。

__future__ future 语句, `from __future__ import <feature>` 指示编译器使用将在未来的 Python 发布版中成为标准的语法和语义来编译当前模块。`__future__` 模块文档记录了可能的 *feature* 取值。通过导入此模块并对其变量求值，你可以看到每项新特性在何时被首次加入到该语言中以及它将（或已）在何时成为默认:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection -- 垃圾回收 释放不再被使用的内存空间的过程。Python 是通过引用计数和一个能够检测和打破循环引用的循环垃圾回收器来执行垃圾回收的。可以使用 `gc` 模块来控制垃圾回收器。

generator -- 生成器 返回一个*generator iterator* 的函数。它看起来很像普通函数，不同点在于其包含 `yield` 表达式以便产生一系列值供给 for-循环使用或是通过 `next()` 函数逐一获取。

通常是指生成器函数，但在某些情况下也可能是指 *生成器迭代器*。如果需要清楚表达具体含义，请使用全称以避免歧义。

generator iterator -- 生成器迭代器 *generator* 函数所创建的对象。

每个 `yield` 会临时暂停处理，记住当前位置执行状态（包括局部变量和挂起的 try 语句）。当该 *生成器迭代器* 恢复时，它会从离开位置继续执行（这与每次调用都从新开始的普通函数差别很大）。

generator expression -- 生成器表达式 返回一个迭代器的表达式。它看起来很像普通表达式后面带有定义了一个循环变量、范围的 `for` 子句，以及一个可选的 `if` 子句。以下复合表达式会为外层函数生成一系列值:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

generic function -- 泛型函数 为不同的类型实现相同操作的多个函数所组成的函数。在调用时会由调度算法来确定应该使用哪个实现。

另请参见*single dispatch* 术语表条目、`functools.singledispatch()` 装饰器以及 **PEP 443**。

generic type -- 泛型 可参数化的*type*；通常为 `list` 或 `dict` 这样的 容器类。用于类型提示 和注解。

更多细节参见 泛型别名类型、**PEP 483**、**PEP 484**、**PEP 585** 和 `typing` 模块。

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。

hash-based pyc -- 基于哈希的 pyc 使用对应源文件的哈希值而非最后修改时间来确定其有效性的字节码缓存文件。参见 pyc-invalidation。

hashable -- 可哈希 一个对象的哈希值如果在其生命周期内绝不改变，就被称为 *可哈希*（它需要具有 `__hash__()` 方法），并可以同其他对象进行比较（它需要具有 `__eq__()` 方法）。可哈希对象必须具有相同的哈希值比较结果才会相同。

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。

大多数 Python 中的不可变内置对象都是可哈希的；可变容器（例如列表或字典）都不可哈希；不可变容器（例如元组和 frozenset）仅当它们的元素均为可哈希时才是可哈希的。用户定义类的实例对象默认是可哈希的。它们在比较时一定不相同（除非是与自己比较），它们的哈希值的生成是基于它们的 `id()`。

IDLE Python 的集成开发与学习环境。idle 是 Python 标准发行版附带的基本编辑器和解释器环境。

immutable -- 不可变对象 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。

import path -- 导入路径 由多个位置（或路径条目）组成的列表，会被模块的*path based finder* 用来查找导入目标。在导入时，此位置列表通常来自 `sys.path`，但对次级包来说也可能来自上级包的 `__path__` 属性。

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*。

interpreter shutdown -- 解释器关闭 当被要求关闭时，Python 解释器将进入一个特殊运行阶段并逐步释放所有已分配资源，例如模块和各种关键内部结构等。它还会多次调用垃圾回收器。这会触发用户定义析构器或弱引用回调中的代码执行。在关闭阶段执行的代码可能会遇到各种异常，因为其所依赖的资源已不再有效（常见的例子有库模块或警告机制等）。

解释器需要关闭的主要原因有 `__main__` 模块或所运行的脚本已完成执行。

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`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *sequence* semantics.

可迭代对象被可用于 `for` 循环以及许多其他需要一个序列的地方（`zip()`、`map()` ...）。当一个可迭代对象作为参数传给内置函数 `iter()` 时，它会返回该对象的迭代器。这种迭代器适用于对值集合的一次性遍历。在使用可迭代对象时，你通常不需要调用 `iter()` 或者自己处理迭代器对象。`for` 语句会为你自动处理那些操作，创建一个临时的未命名变量用来在循环期间保存迭代器。参见*iterator*、*sequence* 以及*generator*。

iterator -- 迭代器 用来表示一连串数据流的对象。重复调用迭代器的 `__next__()` 方法（或将其传给内置函数 `next()`）将逐个返回流中的项。当没有数据可用时则将引发 `StopIteration` 异常。到这时迭代器对象中的数据项已耗尽，继续调用其 `__next__()` 方法只会再次引发 `StopIteration` 异常。迭代器必须具有 `__iter__()` 方法用来返回该迭代器对象自身，因此迭代器必定也是可迭代对象，可被用于其他可迭代对象适用的大部分场合。一个显著的例外是那些会多次重复访问迭代项的代码。容器对象（例如 `list`）在你每次向其传入 `iter()` 函数或是在 `for` 循环中使用它时都会产生一个新的迭代器。如果在此情况下你尝试用迭代器则会返回在之前迭代过程中被耗尽的同一迭代器对象，使其看起来就像是一个空容器。

更多信息可查看 typeiter。

CPython 实现细节： CPython 没有统一应用迭代器定义 `__iter__()` 的要求。

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。

Python 中有许多工具都允许用键函数来控制元素的排位或分组方式。其中包括 `min()`，`max()`，`sorted()`，`list.sort()`，`heapq.merge()`，`heapq.nsmallest()`，`heapq.nlargest()` 以及 `itertools.groupby()`。

有多种方式可以创建一个键函数。例如，`str.lower()` 方法可以用作忽略大小写排序的键函数。或者，键函数也可通过 `lambda` 表达式来创建例如 `lambda r: (r[0], r[2])`。此外，`operator.attrgetter()`，`operator.itemgetter()` 和 `operator.methodcaller()` 是键函数的三个构造器。请查看 排序指引来获取创建和使用键函数的示例。

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。

locale encoding -- 语言区域编码格式 在 Unix 上，它是 LC_CTYPE 语言区域的编码格式。它可以通过 `locale.setlocale(locale.LC_CTYPE, new_locale)` 来设置。

在 Windows 上，它是 ANSI 代码页 (如: "cp1252")。

在 Android 和 VxWorks 上，Python 使用 "utf-8" 作为语言区域编码格式。

locale.getencoding() 可被用来获取语言区域编码格式。

另请参阅*filesystem encoding and error handler*。

list 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 处理一个序列中的所有或部分元素并返回结果列表的一种紧凑写法。result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0] 将生成一个 0 到 255 范围内的十六进制偶数对应字符串 (0x..) 的列表。其中 if 子句是可选的，如果省略则 range(256) 中的所有元素都会被处理。

loader -- 加载器 负责加载模块的对象。它必须定义名为 load_module() 的方法。加载器通常由一个*finder*返回。详情参见 **PEP 302**，对于*abstract base class* 可参见 importlib.abc.Loader。

magic method -- 魔术方法 *special method* 的非正式同义词。

mapping -- 映射 一种支持任意键查找并实现了 collections.abc.Mapping 或 collections.abc.MutableMapping 抽象基类所规定方法的容器对象。此类对象的例子包括 dict, collections.defaultdict, collections.OrderedDict 以及 collections.Counter。

meta path finder -- 元路径查找器 sys.meta_path 的搜索所返回的*finder*。元路径查找器与*path entry finders* 存在关联但并不相同。

请查看 importlib.abc.MetaPathFinder 了解元路径查找器所实现的方法。

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 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*。

module spec -- 模块规格 一个命名空间，其中包含用于加载模块的相关导入信息。是 importlib.machinery.ModuleSpec 的实例。

MRO 参见*method resolution order*。

mutable -- 可变对象 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 术语“具名元组”可用于任何继承自元组，并且其中的可索引元素还能使用名称属性来访问的类型或类。这样的类型或类还可能拥有其他特性。

有些内置类型属于具名元组，包括 time.localtime() 和 os.stat() 的返回值。另一个例子是 sys.float_info:

```
>>> sys.float_info[1]                    # indexed access
1024
>>> sys.float_info.max_exp               # named field access
```

(下页继续)

(续上页)

```
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

有些具名元组是内置类型(例如上面的例子)。此外,具名元组还可通过常规类定义从 tuple 继承并定义名称字段的方式来创建。这样的类可以手工编写，或者使用工厂函数 collections.namedtuple() 创建。后一种方式还会添加一些手工编写或内置具名元组所没有的额外方法。

namespace -- 命名空间 命名空间是存放变量的场所。命名空间有局部、全局和内置的，还有对象中的嵌套命名空间（在方法之内）。命名空间通过防止命名冲突来支持模块化。例如，函数 builtins.open 与 os.open() 可通过各自的命名空间来区分。命名空间还通过明确哪个模块实现那个函数来帮助提高可读性和可维护性。例如，random.seed() 或 itertools.islice() 这种写法明确了这些函数是由 random 与 itertools 模块分别实现的。

namespace package -- 命名空间包 **PEP 420** 所引入的一种仅被用作子包的容器的*package*，命名空间包可以没有实体表示物，其描述方式与*regular package* 不同，因为它们没有 __init__.py 文件。

另可参见*module*。

nested scope -- 嵌套作用域 在一个定义范围内引用变量的能力。例如，在另一函数之内定义的函数可以引用前者的变量。请注意嵌套作用域默认只对引用有效而对赋值无效。局部变量的读写都受限于最内层作用域。类似的，全局变量的读写则作用于全局命名空间。通过 nonlocal 关键字可允许写入外层作用域。

new-style class -- 新式类 对于目前已被应于所有类对象的类形式的旧称谓。在早先的 Python 版本中，只有新式类能够使用 Python 新增的更灵活特性，例如 __slots__、描述符、特征属性、__getattribute__()、类方法和静态方法等。

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。

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，包是具有 __path__ 属性的 Python 模块。

另参见*regular package* 和*namespace package*。

parameter -- 形参 *function*（或方法）定义中的命名实体，它指定函数可以接受的一个*argument*（或在某些情况下，多个实参）。有五种形参：

- *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，指定一个只能通过位置传入的参数。仅限位置形参可通过在函数定义的形参列表中它们之后包含一个 / 字符来定义，例如下面的 *posonly1* 和 *posonly2*:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*：仅限关键字，指定一个只能通过关键字传入的参数。仅限关键字形参可通过在函数定义的形参列表中包含单个可变位置形参或者在多个可变位置形参之前放一个 * 来定义，例如下面的 *kw_only1* 和 *kw_only2*:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *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。

另参见*argument* 术语表条目、参数与形参的区别中的常见问题、`inspect.Parameter` 类、function 一节以及 **PEP 362**。

path entry -- 路径入口 *import path* 中的一个单独位置，会被*path based finder* 用来查找要导入的模块。

path entry finder -- 路径入口查找器 任一可调用对象使用 `sys.path_hooks`（即*path entry hook*）返回的*finder*，此种对象能通过*path entry* 来定位模块。

请参看 `importlib.abc.PathEntryFinder` 以了解路径入口查找器所实现的各个方法。

path entry hook -- 路径入口钩子 一种可调用对象，在知道如何查找特定*path entry* 中的模块的情况下能够使用 `sys.path_hook` 列表返回一个*path entry finder*。

path based finder -- 基于路径的查找器 默认的一种元路径查找器，可在一个*import path* 中查找模块。

path-like object -- 路径类对象 代表一个文件系统路径的对象。路径类对象可以是一个表示路径的 `str` 或者 `bytes` 对象，还可以是一个实现了 `os.PathLike` 协议的对象。一个支持 `os.PathLike` 协议的对象可通过调用 `os.fspath()` 函数转换为 `str` 或者 `bytes` 类型的文件系统路径；`os.fsdecode()` 和 `os.fsencode()` 可被分别用来确保获得 `str` 或 `bytes` 类型的结果。此对象是由 **PEP 519** 引入的。

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**。

portion -- 部分 构成一个命名空间包的单个目录内文件集合（也可能存放于一个 zip 文件内），具体定义见 **PEP 420**。

positional argument -- 位置参数 参见*argument*。

provisional API -- 暂定 API 暂定 API 是指被有意排除在标准库的向后兼容性保证之外的应用编程接口。虽然此类接口通常不会再有重大改变，但只要其被标记为暂定，就可能在核心开发者确定有必要的情况下进行向后不兼容的更改（甚至包括移除该接口）。此种更改并不会随意进行 -- 仅在 API 被加入之前未考虑到的严重基础性缺陷被发现时才可能会这样做。

即便是对暂定 API 来说，向后不兼容的更改也会被视为“最后的解决方案”——任何问题被确认时都会尽可能先尝试找到一种向后兼容的解决方案。

这种处理过程允许标准库持续不断地演进，不至于被有问题的长期性设计缺陷所困。详情见 **PEP 411**。

provisional package -- 暂定包 参见*provisional API*。

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)
```

qualified name -- 限定名称 一个以点号分隔的名称，显示从模块的全局作用域到该模块中定义的某个类、函数或方法的“路径”，相关定义见 **PEP 3155**。对于最高层级的函数和类，限定名称与对象名称一致:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

当被用于引用模块时，完整限定名称意为标示该模块的以点号分隔的整个路径，其中包含其所有的父包，例如 `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count -- 引用计数 对某个对象的引用的数量。当一个对象的引用计数降为零时，它将被释放。引用计数对 Python 代码来说通常是不可见的，但这是*CPython* 实现的一个关键元素。程序员可以调用 `sys.getrefcount()` 函数来返回特定对象的引用计数。

regular package -- 常规包 传统型的*package*，例如包含有一个 `__init__.py` 文件的目录。

另参见*namespace package*。

__slots__ 一种写在类内部的声明，通过预先声明实例属性等对象并移除实例字典来节省内存。虽然这种技巧很流行，但想要用好却并不容易，最好是只保留在少数情况下采用，例如极耗内存的应用程序，并且其中包含大量实例。

sequence 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 `bytes`. 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.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`.

set comprehension -- 集合推导式 处理一个可迭代对象中的所有或部分元素并返回结果集合的一种紧凑写法。`results = {c for c in 'abracadabra' if c not in 'abc'}` 将生成字符串集合 `{'r', 'd'}`。参见 comprehensions。

single dispatch -- 单分派 一种*generic function* 分派形式，其实现是基于单个参数的类型来选择的。

slice -- 切片 通常只包含了特定*sequence* 的一部分的对象。切片是通过使用下标标记来创建的，在 `[]` 中给出几个以冒号分隔的数字，例如 `variable_name[1:3:5]`。方括号（下标）标记在内部使用 `slice` 对象。

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`。

strong reference -- 强引用 在 Python 的 C API 中，强引用是指为持有引用的代码所拥有的对象的引用。在创建引用时可通过调用 `Py_INCREF()` 来获取强引用而在删除引用时可通过 `Py_DECREF()` 来释放它。

`Py_NewRef()` 函数可被用于创建一个对象的强引用。通常，必须在退出某个强引用的作用域时在该强引用上调用 `Py_DECREF()` 函数，以避免引用的泄漏。

另请参阅*borrowed reference*。

text encoding -- 文本编码格式 在 Python 中，一个字符串是一串 Unicode 代码点（范围为 `U+0000`--`U+10FFFF`）。为了存储或传输一个字符串，它需要被序列化为一串字节。

将一个字符串序列化为一个字节序列被称为” 编码”，而从字节序列中重新创建字符串被称为” 解码”。

有各种不同的文本序列化 编码器，它们被统称为” 文本编码格式”。

text file -- 文本文件 一种能够读写 `str` 对象的*file object*。通常一个文本文件实际是访问一个面向字节的数据流并自动处理*text encoding*。文本文件的例子包括以文本模式（`'r'` 或 `'w'`）打开的文件、`sys.stdin`、`sys.stdout` 以及 `io.StringIO` 的实例。

另请参看*binary file* 了解能够读写字节类对象 的文件对象。

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。

type alias -- 类型别名 一个类型的同义词，创建方式是把类型赋值给特定的标识符。

类型别名的作用是简化类型注解。例如:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

可以这样提高可读性:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

参见 `typing` 和 **PEP 484**，其中有对此功能的详细描述。

type hint -- 类型注解 *annotation* 为变量、类属性、函数的形参或返回值指定预期的类型。

类型注解属于可选项，Python 不要求提供，但其可对静态类型分析工具起作用，并可协助 IDE 实现代码补全与重构。

全局变量、类属性和函数的类型注解可以使用 `typing.get_type_hints()` 来访问，但局部变量则不可以。

参见 `typing` 和 **PEP 484**，其中有对此功能的详细描述。

universal newlines -- 通用换行 一种解读文本流的方式，将以下所有符号都识别为行结束标志：Unix 的行结束约定 `'\n'`、Windows 的约定 `'\r\n'` 以及旧版 Macintosh 的约定 `'\r'`。参见 **PEP 278** 和 **PEP 3116** 和 `bytes.splitlines()` 了解更多用法说明。

variable annotation -- 变量标注 对变量或类属性的*annotation*。

在标注变量或类属性时，还可选择为其赋值:

```
class C:
    field: 'annotation'
```

变量标注通常被用作类型注解：例如以下变量预期接受 int 类型的值:

```
count: int = 0
```

变量标注语法的详细解释见 annassign 一节。

参见*function annotation*, **PEP 484** 和 **PEP 526**，其中描述了此功能。另请参阅 annotations-howto 以了解使用标注的最佳实践。

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。

另参见 venv。

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

这些文档是用 Sphinx 从 reStructuredText 源生成的，*Sphinx* 是一个专为处理 Python 文档而编写的文档生成器。

本文档及其工具链之开发，皆在于志愿者之努力，亦恰如 Python 本身。如果您想为此作出贡献，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.，原始 Python 文档工具集之创造者，众多文档之作者；
- 用于创建 reStructuredText 和 Docutils 套件的 Docutils 项目；
- Fredrik Lundh 的 Alternative Python Reference 项目，Sphinx 从中得到了许多好的想法。

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 源代码分发的 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 公司；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 公司现在是 Python 软件基金会的赞助成员。

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

备注： 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

Python 软件和文档的使用许可均基于*PSF 许可协议*。

从 Python 3.8.6 开始，文档中的示例、操作指导和其他代码采用的是 PSF 许可协议和*零条款 BSD 许可* 的双重使用许可。

某些包含在 Python 中的软件基于不同的许可。这些许可会与相应许可之下的代码一同列出。有关这些许可的不完整列表请参阅*收录软件的许可与鸣谢*。

C.2.1 用于 PYTHON 3.11.6 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
   3.11.6 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 3.11.6 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.11.6 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 3.11.6 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
   3.11.6.

4. PSF is making Python 3.11.6 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 3.11.6 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.11.6
```

```
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.11.6, 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 3.11.6, 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.
```

(下页继续)

(续上页)

```
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

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
```

(下页继续)

(续上页)

```
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

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.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.11.6 DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
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 收录软件的许可与鸣谢

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

The `_random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
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.

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 使用了 getaddrinfo() 和 getnameinfo() WIDE 项目的不同源文件中: https://www.wide.ad.jp/

```
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
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 异步套接字服务

The asynchat and asyncore modules contain the following notice:

```
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.3.4 Cookie 管理

http.cookies 模块包含以下声明:

```
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.3.5 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.3.6 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
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML 远程过程调用

xmlrpc.client 模块包含以下声明:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

(下页继续)

(续上页)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

`test.test_epoll` 模块包含以下说明:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
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.3.9 Select kqueue

`select` 模块关于 kqueue 的接口包含以下声明:

```
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
```

(下页继续)

(续上页)

```
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

Python/pyhash.c 文件包含 Marek Majkowski' 对 Dan Bernstein 的 SipHash24 算法的实现。它包含以下声明:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

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.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 和 dtoa

Python/dtoa.c 文件提供了 C 函数 dtoa 和 strtod，用于 C 双精度数值和字符串之间的转换，它派生自由 David M. Gay 编写的同名文件。目前该文件可在 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c 访问。在 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
 * 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.3.12 OpenSSL

如果操作系统提供支持，则 hashlib, posix, ssl, crypt 模块会使用 OpenSSL 库来提高性能。此外，Python 的 Windows 和 macOS 安装程序可能会包括 OpenSSL 库的副本，所以我们也在此包括一份 OpenSSL 许可证的副本。对于 OpenSSL 3.0 版及其后续衍生版本，均使用 Apache 许可证 v2:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
```

(下页继续)

(续上页)

```
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
```

(下页继续)

(续上页)

```
      include a readable copy of the attribution notices contained
      within such NOTICE file, excluding those notices that do not
      pertain to any part of the Derivative Works, in at least one
      of the following places: within a NOTICE text file distributed
      as part of the Derivative Works; within the Source form or
      documentation, if provided along with the Derivative Works; or,
      within a display generated by the Derivative Works, if and
      wherever such third-party notices normally appear. The contents
      of the NOTICE file are for informational purposes only and
      do not modify the License. You may add Your own attribution
      notices within Derivative Works that You distribute, alongside
      or as an addendum to the NOTICE text from the Work, provided
      that such additional attribution notices cannot be construed
      as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.
```

(下页继续)

(续上页)

```
9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

pyexpat 扩展是使用所包括的 expat 源副本来构建的，除非配置了 --with-system-expat:

```
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.3.14 libffi

The _ctypes extension is built using an included copy of the libffi sources unless the build is configured --with-system-libffi:

```
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
```

(下页继续)

(续上页)

```
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 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) 1995-2011 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
```

C.3.16 cfuhash

`tracemalloc` 使用的哈希表的实现基于 cfuhash 项目:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
```

(下页继续)

(续上页)

```
  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * 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.

  * Neither the name of the author 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 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.
```

C.3.17 libmpdec

The `_decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. 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.3.18 W3C C14N 测试套件

test 包中的 C14N 2.0 测试集 (Lib/test/xmltestdata/c14n-20/) 提取自 W3C 网站 https://www.w3.org/TR/xml-c14n2-testcases/ 并根据 3 条款版 BSD 许可证发行:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work 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.
```

C.3.19 audioop

audioop 模块使用 SoX 项目的 g771.c 文件中的代码为基础:

```
Programming the AdLib/Sound Blaster
FM Music Chips
Version 2.0 (24 Feb 1992)
Copyright (c) 1991, 1992 by Jeffrey S. Lee
jlee@smylex.uucp
Warranty and Copyright Policy
This document is provided on an "as-is" basis, and its author makes
no warranty or representation, express or implied, with respect to
its quality performance or fitness for a particular purpose.  In no
event will the author of this document be liable for direct, indirect,
special, incidental, or consequential damages arising out of the use
or inability to use the information contained within.  Use of this
document is at your own risk.
This file may be used and copied freely so long as the applicable
copyright notices are retained, and no modifications are made to the
text of the document.  No money shall be charged for its distribution
beyond reasonable shipping, handling and duplication costs, nor shall
proprietary changes be made to this document so that it cannot be
distributed freely.  This document may not be included in published
```

(下页继续)

(续上页)

```
material or commercial packages without the written consent of its
author.
```

APPENDIX D

版权所有

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

H

I

K

L

M

N

O

P