Extending and Embedding Python

發Ⓕ *3.13.2*

Guido van Rossum and the Python development team

Contents

這份⊞明文件描述如何在 C 或 C++ 中編寫模組，⊞使用新模組來擴充 Python 直譯器功能。那些模組不僅可以定義新的函式，也可以定義新的物件型⊞及其方法 (method)。文件⊞容也會描述如何將 Python 直譯器嵌入另一個應用程式中，做⊞一種擴充語言 (extension language) 使用。最後，它會展示如何編譯及連結擴充模組，使那些模組可以動態地（在執行環境）被載入到直譯器中，前提是底層作業系統有支援這個功能。

這份⊞明文件假設你具備 Python 的基礎知識。關於此語言的非正式介紹，請參⊞ tutorial-index。reference-index 給予此語言更⊞正式的定義。library-index 記⊞了賦予此語言廣泛應用範圍的物件型⊞、函式與（⊞建的和以 Python 編寫的）模組。

關於完整的 Python/C API 詳細介紹，請參⊞另外一份 c-api-index。

CHAPTER 1

推薦的第三方工具

這份指南僅涵蓋了此 CPython 版本所提供的、用以建立擴充的基本工具。第三方工具，例如 Cython、cffi、SWIG 和 Numba，提供了更F簡單及更FF雜的多種方法，來F Python 建立 C 和 C ++ 擴充。

> ➦ **也參考**
>
> **Python 封裝使用者指南：二進制擴充**
>
> Python 封裝使用者指南 (Python Packaging User Guide) 不僅涵蓋了數個可以用來簡化二進制擴充建立過程的工具，也會討論F何建立一個擴充模組可能會是你的優先考量。

不使用第三方工具建立擴充

本指南中的這一節將說明，在沒有第三方工具的協助下，如何建立 C 和 C ++ 擴充。它主要是寫給使用那些工具的創作者們，而不是讓你建立自己的 C 擴充的推薦方法。

2.1 以 C 或 C++ 擴充 Python

如果你會撰寫 C 程式語言，那要向 Python 新增內建模組就不困難。這種擴充模組 *(extension modules)* 可以做兩件在 Python 中無法直接完成的事：它們可以實作新的內建物件型別，並且可以呼叫 C 的函式庫函式和系統呼叫。

為了支援擴充，Python API (Application Programmers Interface) 定義了一組函式、巨集和變數，提供對 Python run-time 系統大部分面向的存取。Python API 是透過引入標頭檔 "Python.h" 來被納入到一個 C 原始碼檔案中。

擴充模組的編譯取決於其預期用途以及你的系統設定；詳細資訊將在後面的章節中提供。

> ℹ️ **備註**
>
> C 擴充介面是 CPython 所特有的，擴充模組在其他 Python 實作上無法運作。在許多情況下，可以避免撰寫 C 擴充而保留對其他實作的可移植性。例如，如果你的用例是呼叫 C 函式庫函式或系統呼叫，你應該考慮使用 ctypes 模組或 cffi 函式庫，而不是編寫自定義的 C 程式碼。這些模組讓你可以撰寫 Python 程式碼來與 C 程式碼介接，而且比起撰寫和編譯 C 擴充模組，這些模組在 Python 實作之間更容易移植。

2.1.1 一個簡單范例

讓我們來建立一個叫做 spam（Monty Python 粉絲最愛的食物...）的擴充模組。假設我們要建立一個 Python 介面給 C 函式庫的函式 system()[1] 使用，這個函式接受一個以 null 終止的 (null-terminated) 字元字串做為引數，並回傳一個整數。我們希望這個函式可以在 Python 中被呼叫，如下所示：

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

首先建立一個檔案 spammodule.c。（從過去歷史來看，如果一個模組叫做 spam，包含其實作的 C 檔案就會叫做 spammodule.c；如果模組名稱很長，像是 spammify，模組名稱也可以只是 spammify.c）。

---

[1] An interface for this function already exists in the standard module os --- it was chosen as a simple and straightforward example.

我們檔案的前兩列可以是:

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

這會將 Python API 拉進來(你可以加入註解來說明模組的目的,也可以加入版權聲明)。

> **ℹ 備註**
>
> 由於 Python 可能定義一些影響系統上某些標準標頭檔的預處理器定義,你必須在引入任何標準標頭檔之前引入 `Python.h`。
>
> `#define PY_SSIZE_T_CLEAN` 被用來表示在某些 API 中應該使用 `Py_ssize_t` 而不是 `int`。自 Python 3.13 起,它就不再是必要的了,但我們在此保留它以便向後相容。關於這個巨集的描述請參閱 arg-parsing-string-and-buffers。

除了那些在標準標頭檔中定義的符號以外,所有由 `Python.h` 定義的使用者可見符號 (user-visible symbols) 的前綴都是 `Py` 或 `PY`。為了方便,也因為 Python 直譯器的大量使用,`"Python.h"` 也引入了一些標準的標頭檔: `<stdio.h>`、`<string.h>`、`<errno.h>` 和 `<stdlib.h>`。如果 `<stdlib.h>` 在你的系統上不存在,它會直接宣告 `malloc()`、`free()` 和 `realloc()` 函式。

接下來我們要加入到模組檔案的是 C 函式,當 Python 運算式 `spam.system(string)` 要被求值 (evaluated) 時就會被呼叫(我們很快就會看到它最後是如何被呼叫的):

```c
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);
}
```

可以很直觀地從 Python 的引數串列(例如單一的運算式 `"ls -l"`)直接轉換成傳給 C 函式的引數。C 函式總是有兩個引數,習慣上會命名為 *self* 和 *args*。

對於模組層級的函式,*self* 引數會指向模組物件;而對於方法來說則是指向物件的實例。

args 引數會是一個指向包含引數的 Python 元組物件的指標。元組中的每一項都對應於呼叫的引數串列中的一個引數。引數是 Python 物件 --- 為了在我們的 C 函式中對它們做任何事情,我們必須先將它們轉換成 C 值。Python API 中的 `PyArg_ParseTuple()` 函式能夠檢查引數型別並將他們轉換成 C 值。它使用模板字串來指定所需的引數型別以及儲存轉換值的 C 變數型別。稍後會再詳細說明。

如果所有的引數都有正確的型別,且其元件已儲存在傳入位址的變數中,則 `PyArg_ParseTuple()` 會回傳 true(非零)。如果傳入的是無效引數串列則回傳 false(零)。在後者情況下,它也會產生適當的例外,因此呼叫函式可以立即回傳 `NULL`(就像我們在範例中所看到的)。

2.1.2 插曲: 錯誤與例外

在整個 Python 直譯器中的一個重要慣例是: 當一個函式失敗時,它就應該設定一個例外條件,並回傳一個錯誤值(通常是 `-1` 或一個 `NULL` 指標)。例外資訊會儲存在直譯器執行緒狀態的三個成員中。如果沒有例外,它們就會是 `NULL`。否則,它們是由 `sys.exc_info()` 所回傳的 Python 元組中的 C 等效元組。它們是例外型別、例外實例和回溯物件。了解它們對於理解錯誤是如何傳遞是很重要的。

Python API 定義了許多能夠設定各種類型例外的函式。

最常見的是 `PyErr_SetString()`。它的引數是一個例外物件和一個 C 字串。例外物件通常是預先定義的物件,例如 `PyExc_ZeroDivisionError`。C 字串則指出錯誤的原因,並被轉換成 Python 字串物件且被儲存為例外的「關聯值 (associated value)」。

另一個有用的函式是 `PyErr_SetFromErrno()`，它只接受一個例外引數，並透過檢查全域變數 `errno` 來建立關聯值。最一般的函式是 `PyErr_SetObject()`，它接受兩個物件引數，即例外和它的關聯值。你不需要對傳給任何這些函式的物件呼叫 `Py_INCREF()`。

你可以使用 `PyErr_Occurred()` 來不具破壞性地測試例外是否已被設定。這會回傳當前的例外物件，如果沒有例外發生則回傳 `NULL`。你通常不需要呼叫 `PyErr_Occurred()` 來查看函式呼叫是否發生錯誤，因為你應可從回傳值就得知。

當函式 *f* 呼叫另一個函式 *g* 時檢測到後者失敗，*f* 本身應該回傳一個錯誤值（通常是 `NULL` 或 `-1`）。它不應該呼叫 `PyErr_*` 函式的其中一個，這會已被 *g* 呼叫過。*f* 的呼叫者然後也應該回傳一個錯誤指示給它的呼叫者，同樣不會呼叫 `PyErr_*`，依此類推 --- 最詳細的錯誤原因已經被首先檢測到它的函式回報了。一旦錯誤到達 Python 直譯器的主要迴圈，這會中止目前執行的 Python 程式碼，並嘗試尋找 Python 程式設計者指定的例外處理程式。

（在某些情況下，模組可以透過呼叫另一個 `PyErr_*` 函式來提供更詳細的錯誤訊息，在這種情況下這樣做是沒問題的。然而這一般來說並非必要，而且可能會導致錯誤原因資訊的遺失：大多數的操作都可能因為各種原因而失敗。）

要忽略由函式呼叫失敗所設定的例外，必須明確地呼叫 `PyErr_Clear()` 來清除例外條件。C 程式碼唯一要呼叫 `PyErr_Clear()` 的情況是當它不想將錯誤傳遞給直譯器而想要完全是自己來處理它時（可能是要再嘗試其他東西，或者假裝什麼都沒出錯）。

每個失敗的 `malloc()` 呼叫都必須被轉換成一個例外 --- `malloc()`（或 `realloc()`）的直接呼叫者必須呼叫 `PyErr_NoMemory()` 並回傳一個失敗指示器。所有建立物件的函式（例如 `PyLong_FromLong()`）都已經這樣做了，所以這個注意事項只和那些直接呼叫 `malloc()` 的函式有關。

還要注意的是，有 `PyArg_ParseTuple()` 及同系列函式的這些重要例外，回傳整數狀態的函式通常會回傳一個正值或 0 表示成功、回傳 `-1` 表示失敗，就像 Unix 系統呼叫一樣。

最後，在回傳錯誤指示器時要注意垃圾清理（透過對你已經建立的物件呼叫 `Py_XDECREF()` 或 `Py_DECREF()`）！

你完全可以自行選擇要產生的例外。有一些預先宣告的 C 物件會對應到所有內建的 Python 例外，例如 `PyExc_ZeroDivisionError`，你可以直接使用它們。當然，你應該明智地選擇例外，像是不要使用 `PyExc_TypeError` 來表示檔案無法打開（應該是 `PyExc_OSError`）。如果引數串列有問題，`PyArg_ParseTuple()` 函式通常會引發 `PyExc_TypeError`。如果你有一個引數的值必須在一個特定的範圍內或必須滿足其他條件，則可以使用 `PyExc_ValueError`。

你也可以定義一個你的模組特有的新例外。因此你通常會在檔案開頭宣告一個靜態物件變數：

```
static PyObject *SpamError;
```

並在你的模組初始化函式中使用一個例外物件來初始化它 (`PyInit_spam()`):

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

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

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

    return m;
}
```

請注意，例外物件的 Python 名稱是 `spam.error`。如同 bltin-exceptions 所述，`PyErr_NewException()` 函式可能會建立一個基底類別為 `Exception` 的類別（除非傳入另一個類別來代替 `NULL`）。

請注意，`SpamError` 變數保留了對新建立的例外類⬚的參照；這是故意的！因⬚外部程式碼可能會從模組中移除這個例外，所以需要一個對這個類⬚的參照來確保它不會被⬚⬚而導致 `SpamError` 變成一個迷途指標 (dangling pointer)。如果它變成迷途指標，那⬚生例外的 C 程式碼可能會導致核心轉儲 (core dump) 或其他不預期的 side effect。

我們稍後會討論 `PyMODINIT_FUNC` 作⬚函式回傳型⬚的用法。

可以在你的擴充模組中呼叫 `PyErr_SetString()` 來引發 `spam.error` 例外，如下所示：

```c
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 回到范例

回到我們的範例函式，現在你應該可以理解這個陳述式了：

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

如果在引數串列中檢測到錯誤則會回傳 `NULL`（回傳物件指標之函式的錯誤指示器），其依賴於 `PyArg_ParseTuple()` 設定的例外，否則引數的字串值會已被⬚⬚到區域變數 `command` 中。這是一個指標賦值，你不應該修改它所指向的字串（所以在標準 C 中，`command` 變數應該正確地被宣告⬚ `const char *command`）。

接下來的陳述式會呼叫 Unix 函式 `system()`，⬚將剛才從 `PyArg_ParseTuple()` 得到的字串傳給它：

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

我們的 `spam.system()` 函式必須以 Python 物件的形式來回傳 `sts` 的值。這是透過 `PyLong_FromLong()` 函式來達成。

```c
return PyLong_FromLong(sts);
```

在這種情⬚下它會回傳一個整數物件。(是的，在 Python 中連整數也是堆積 (heap) 上的物件！）

如果你有一個不回傳任何有用引數的 C 函式（一個回傳 `void` 的函式），對應的 Python 函式必須回傳 `None`。你需要以下這個慣例來達成（由 `Py_RETURN_NONE` 巨集實作）：

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

`Py_None` 是特殊 Python 物件 `None` 的 C 名稱。它是一個真正的 Python 物件而不是一個 `NULL` 指標，在大多數的情⬚下它的意思是「錯誤」，如我們所見過的那樣。

2.1.4 模組的方法表和初始化函式

我承諾過要展示 `spam_system()` 是如何從 Python 程式中呼叫的。首先，我們需要在「方法表」中列出它的名稱和位址：

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

請注意第三個項目 (METH_VARARGS)。這是一個告訴直譯器 C 函式之呼叫方式的旗標。通常應該是 METH_VARARGS 或 METH_VARARGS | METH_KEYWORDS；0 表示是使用 PyArg_ParseTuple() 的一個過時變體。

當只使用 METH_VARARGS 時，函式應預期 Python 層級的參數是以元組形式傳入且能⊞接受以 PyArg_ParseTuple() 進行剖析；有關此函式的更多資訊將在下面提供。

如果要將關鍵字引數傳給函式，可以在第三個欄位設定 METH_KEYWORDS 位元。在這種情⊞下，C 函式應該要能接受第三個 PyObject * 參數，這個參數將會是關鍵字的字典。可使用 PyArg_ParseTupleAndKeywords() 來剖析這種函式的引數。

方法表必須在模組定義結構中被參照：

```
static struct PyModuleDef spammodule = {
    PyModuleDef_HEAD_INIT,
    "spam",  /* 模組名稱 */
    spam_doc, /* 模組文件，可能⊞ NULL */
    -1,       /* 模組的個⊞直譯器狀態的大小，
                 如果模組將狀態保存在全域變數中則⊞ -1 */
    SpamMethods
};
```

反過來⊞，這個結構必須在模組的初始化函式中被傳給直譯器。初始化函式必須被命名⊞ PyInit_name()，其中 *name* 是模組的名稱，且應該是模組檔案中唯一定義的非「⊞態 (static)」項目：

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

請注意，PyMODINIT_FUNC 宣告函式的回傳型⊞⊞ *PyObject **'、宣告平台所需的任何特殊連結宣告、⊞針對 C++ 宣告函式⊞ extern "C"。

當 Python 程式第一次引入模組 spam 時，PyInit_spam() 會被呼叫。（有關嵌入 Python 的⊞解請參見下文。）它會呼叫回傳一個模組物件的 PyModule_Create()，⊞根據在模組定義中所找到的表（一個 PyMethodDef 結構的陣列）將⊞建的函式物件插入到新建立的模組中。PyModule_Create() 會回傳一個指向它建立之模組物件的指標。對於某些錯誤情⊞，它可能會以嚴重錯誤的形式來中止；如果模組無法令人滿意地被初始化，它也會回傳 NULL。初始化函式必須把模組物件回傳給它的呼叫者，這樣它才會被插入到 sys.modules 中。

嵌入 Python 時，除非在 PyImport_Inittab 表中有相關條目，否則不會自動呼叫 PyInit_spam() 函式。要將模組加入初始化表，請使用 PyImport_AppendInittab()⊞在隨後選擇性地將該模組引入：

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

int
main(int argc, char *argv[])
{
    PyStatus status;
    PyConfig config;
    PyConfig_InitPythonConfig(&config);
```

<div align="right">(繼續下一頁)</div>

```c
    /* 在 Py_Initialize 之前加入⊡建模組 */
    if (PyImport_AppendInittab("spam", PyInit_spam) == -1) {
        fprintf(stderr, "Error: could not extend in-built modules table\n");
        exit(1);
    }

    /* 將 argv[0] 傳給 Python 直譯器 */
    status = PyConfig_SetBytesString(&config, &config.program_name, argv[0]);
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    /* 初始化 Python 直譯器。這會是必要的。
       如果此步驟失敗就會導致嚴重錯誤。 */
    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);

    /* 可選擇引入模組; 或者
       可以延遲引入，直至嵌入式⊡本
       將其引入。 */
    PyObject *pmodule = PyImport_ImportModule("spam");
    if (!pmodule) {
        PyErr_Print();
        fprintf(stderr, "Error: could not import module 'spam'\n");
    }

    // ... 在此使用 Python C API ...

    return 0;

 exception:
    PyConfig_Clear(&config);
    Py_ExitStatusException(status);
}
```

> **ⓘ 備⊡**
>
> 從 sys.modules 中移除項目，或在一個行程中將已編譯模組引入到多個直譯器中（或在⊡有 exec() 介入的情⊡下使用 fork()）可能會對某些擴充模組造成問題。擴充模組作者在初始化⊡部資料結構時應特⊡小心。

Python 原始碼發行版本中包含了一個更實質的範例模組 Modules/xxmodule.c。這個檔案可以當作模板使用，也可以簡單地當作範例來⊡讀。

> **ⓘ 備⊡**
>
> 不像我們的 spam 範例，xxmodule 使用了多階段初始化 *(multi-phase initialization)*（Python 3.5 新增），其中的 PyModuleDef 結構會從 PyInit_spam 回傳，而模組的建立則交由引入機制來完成。關於多階段初始化的詳細資訊請參⊡ **PEP 489**。

2.1.5 Compilation and Linkage

There are two more things to do before you can use your new extension: compiling and linking it with the Python system. If you use dynamic loading, the details may depend on the style of dynamic loading your system uses; see the chapters about building extension modules (chapter 建立 *C* 與 *C++* 擴充套件) and additional information that pertains only to building on Windows (chapter 建置 *Windows* 上的 *C* 和 *C++* 擴充) for more information about this.

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

```
spam spammodule.o
```

and rebuild the interpreter by running **make** in the toplevel directory. You can also run **make** in the `Modules/` subdirectory, but then you must first rebuild `Makefile` there by running '**make** Makefile'. (This is necessary each time you change the `Setup` file.)

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

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

2.1.6 Calling Python Functions from C

So far we have concentrated on making C functions callable from Python. The reverse is also useful: calling Python functions from C. This is especially the case for libraries that support so-called "callback" functions. If a C interface makes use of callbacks, the equivalent Python often needs to provide a callback mechanism to the Python programmer; the implementation will require calling the Python callback functions from a C callback. Other uses are also imaginable.

Fortunately, the Python interpreter is easily called recursively, and there is a standard interface to call a Python function. (I won't dwell on how to call the Python parser with a particular string as input --- if you're interested, have a look at the implementation of the `-c` command line option in `Modules/main.c` from the Python source code.)

Calling a Python function is easy. First, the Python program must somehow pass you the Python function object. You should provide a function (or some other interface) to do this. When this function is called, save a pointer to the Python function object (be careful to `Py_INCREF()` it!) in a global variable --- or wherever you see fit. For example, the following function might be part of a module definition:

```c
static PyObject *my_callback = NULL;

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

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

(繼續下一頁)

```
    return result;
}
```

This function must be registered with the interpreter using the `METH_VARARGS` flag; this is described in section 模組的方法表和初始化函式. The `PyArg_ParseTuple()` function and its arguments are documented in section *Extracting Parameters in Extension Functions*.

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

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

```c
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()` returns a Python object pointer: this is the return value of the Python function. `PyObject_CallObject()` is "reference-count-neutral" with respect to its arguments. In the example a new tuple was created to serve as the argument list, which is `Py_DECREF()`-ed immediately after the `PyObject_CallObject()` call.

The return value of `PyObject_CallObject()` is "new": either it is a brand new object, or it is an existing object whose reference count has been incremented. So, unless you want to save it in a global variable, you should somehow `Py_DECREF()` the result, even (especially!) if you are not interested in its value.

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

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

Depending on the desired interface to the Python callback function, you may also have to provide an argument list to `PyObject_CallObject()`. In some cases the argument list is also provided by the Python program, through the same interface that specified the callback function. It can then be saved and used in the same manner as the function object. In other cases, you may have to construct a new tuple to pass as the argument list. The simplest way to do this is to call `Py_BuildValue()`. For example, if you want to pass an integral event code, you might use the following code:

```c
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);
```

Note the placement of `Py_DECREF(arglist)` immediately after the call, before the error check! Also note that strictly speaking this code is not complete: `Py_BuildValue()` may run out of memory, and this should be checked.

You may also call a function with keyword arguments by using `PyObject_Call()`, which supports arguments and keyword arguments. As in the above example, we use `Py_BuildValue()` to construct the dictionary.

```
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 Extracting Parameters in Extension Functions

The `PyArg_ParseTuple()` function is declared as follows:

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

The *arg* argument must be a tuple object containing an argument list passed from Python to a C function. The *format* argument must be a format string, whose syntax is explained in arg-parsing in the Python/C API Reference Manual. The remaining arguments must be addresses of variables whose type is determined by the format string.

Note that while `PyArg_ParseTuple()` checks that the Python arguments have the required types, it cannot check the validity of the addresses of C variables passed to the call: if you make mistakes there, your code will probably crash or at least overwrite random bits in memory. So be careful!

Note that any Python object references which are provided to the caller are *borrowed* references; do not decrement their reference count!

一些呼叫範例：

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

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

ok = PyArg_ParseTuple(args, ""); /* F有引數 */
    /* Python 呼叫: 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 Keyword Parameters for Extension Functions

The `PyArg_ParseTupleAndKeywords()` function is declared as follows:

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

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

> ℹ️ 備F
>
> Nested tuples cannot be parsed when using keyword arguments! Keyword parameters passed in which are not present in the *kwlist* will cause `TypeError` to be raised.

Here is an example module which uses keywords, based on an example by Geoff Philbrick (philbrick@hks.com):

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

static PyObject *
keywdarg_parrot(PyObject *self, PyObject *args, PyObject *keywds)
{
```

(繼續下一頁)

```c
    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 Building Arbitrary Values

This function is the counterpart to `PyArg_ParseTuple()`. It is declared as follows:

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

It recognizes a set of format units similar to the ones recognized by `PyArg_ParseTuple()`, but the arguments (which are input to the function, not output) must not be pointers, just values. It returns a new Python object, suitable for returning from a C function called from Python.

One difference with `PyArg_ParseTuple()`: while the latter requires its first argument to be a tuple (since Python argument lists are always represented as tuples internally), `Py_BuildValue()` does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns `None`; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

Examples (to the left the call, to the right the resulting Python value):

```
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 Reference Counts

In languages like C or C++, the programmer is responsible for dynamic allocation and deallocation of memory on the heap. In C, this is done using the functions `malloc()` and `free()`. In C++, the operators `new` and `delete` are used with essentially the same meaning and we'll restrict the following discussion to the C case.

Every block of memory allocated with `malloc()` should eventually be returned to the pool of available memory by exactly one call to `free()`. It is important to call `free()` at the right time. If a block's address is forgotten but `free()` is not called for it, the memory it occupies cannot be reused until the program terminates. This is called a *memory leak*. On the other hand, if a program calls `free()` for a block and then continues to use the block, it creates a conflict with reuse of the block through another `malloc()` call. This is called *using freed memory*. It has the same bad consequences as referencing uninitialized data --- core dumps, wrong results, mysterious crashes.

Common causes of memory leaks are unusual paths through the code. For instance, a function may allocate a block of memory, do some calculation, and then free the block again. Now a change in the requirements for the function may add a test to the calculation that detects an error condition and can return prematurely from the function. It's easy to forget to free the allocated memory block when taking this premature exit, especially when it is added later to the code. Such leaks, once introduced, often go undetected for a long time: the error exit is taken only in a small fraction of all calls, and most modern machines have plenty of virtual memory, so the leak only becomes apparent in a long-running process that uses the leaking function frequently. Therefore, it's important to prevent leaks from happening by having a coding convention or strategy that minimizes this kind of errors.

Since Python makes heavy use of `malloc()` and `free()`, it needs a strategy to avoid memory leaks as well as the use of freed memory. The chosen method is called *reference counting*. The principle is simple: every object contains a counter, which is incremented when a reference to the object is stored somewhere, and which is decremented when a reference to it is deleted. When the counter reaches zero, the last reference to the object has been deleted and the object is freed.

An alternative strategy is called *automatic garbage collection*. (Sometimes, reference counting is also referred to as a garbage collection strategy, hence my use of "automatic" to distinguish the two.) The big advantage of automatic garbage collection is that the user doesn't need to call `free()` explicitly. (Another claimed advantage is an improvement in speed or memory usage --- this is no hard fact however.) The disadvantage is that for C, there is no truly portable automatic garbage collector, while reference counting can be implemented portably (as long as the functions `malloc()` and `free()` are available --- which the C Standard guarantees). Maybe some day a sufficiently portable automatic garbage collector will be available for C. Until then, we'll have to live with reference counts.

While Python uses the traditional reference counting implementation, it also offers a cycle detector that works to detect reference cycles. This allows applications to not worry about creating direct or indirect circular references; these are the weakness of garbage collection implemented using only reference counting. Reference cycles consist of objects which contain (possibly indirect) references to themselves, so that each object in the cycle has a reference count which is non-zero. Typical reference counting implementations are not able to reclaim the memory belonging to any objects in a reference cycle, or referenced from the objects in the cycle, even though there are no further references to the cycle itself.

The cycle detector is able to detect garbage cycles and can reclaim them. The `gc` module exposes a way to run the detector (the `collect()` function), as well as configuration interfaces and the ability to disable the detector at runtime.

Reference Counting in Python

There are two macros, `Py_INCREF(x)` and `Py_DECREF(x)`, which handle the incrementing and decrementing of the reference count. `Py_DECREF()` also frees the object when the count reaches zero. For flexibility, it doesn't call `free()` directly --- rather, it makes a call through a function pointer in the object's *type object*. For this purpose (and others), every object also contains a pointer to its type object.

The big question now remains: when to use `Py_INCREF(x)` and `Py_DECREF(x)`? Let's first introduce some terms. Nobody "owns" an object; however, you can *own a reference* to an object. An object's reference count is now defined as the number of owned references to it. The owner of a reference is responsible for calling `Py_DECREF()` when the reference is no longer needed. Ownership of a reference can be transferred. There are three ways to dispose of an owned reference: pass it on, store it, or call `Py_DECREF()`. Forgetting to dispose of an owned reference creates a memory leak.

It is also possible to *borrow*[2] a reference to an object. The borrower of a reference should not call `Py_DECREF()`. The borrower must not hold on to the object longer than the owner from which it was borrowed. Using a borrowed reference after the owner has disposed of it risks using freed memory and should be avoided completely[3].

The advantage of borrowing over owning a reference is that you don't need to take care of disposing of the reference on all possible paths through the code --- in other words, with a borrowed reference you don't run the risk of leaking when a premature exit is taken. The disadvantage of borrowing over owning is that there are some subtle situations where in seemingly correct code a borrowed reference can be used after the owner from which it was borrowed has in fact disposed of it.

A borrowed reference can be changed into an owned reference by calling `Py_INCREF()`. This does not affect the status of the owner from which the reference was borrowed --- it creates a new owned reference, and gives full owner responsibilities (the new owner must dispose of the reference properly, as well as the previous owner).

Ownership Rules

Whenever an object reference is passed into or out of a function, it is part of the function's interface specification whether ownership is transferred with the reference or not.

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

Many functions that extract objects from other objects also transfer ownership with the reference, for instance `PyObject_GetAttrString()`. The picture is less clear, here, however, since a few common routines are exceptions: `PyTuple_GetItem()`, `PyList_GetItem()`, `PyDict_GetItem()`, and `PyDict_GetItemString()` all return references that you borrow from the tuple, list or dictionary.

The function `PyImport_AddModule()` also returns a borrowed reference, even though it may actually create the object it returns: this is possible because an owned reference to the object is stored in `sys.modules`.

When you pass an object reference into another function, in general, the function borrows the reference from you --- if it needs to store it, it will use `Py_INCREF()` to become an independent owner. There are exactly two important exceptions to this rule: `PyTuple_SetItem()` and `PyList_SetItem()`. These functions take over ownership of the item passed to them --- even if they fail! (Note that `PyDict_SetItem()` and friends don't take over ownership --- they are "normal.")

When a C function is called from Python, it borrows references to its arguments from the caller. The caller owns a reference to the object, so the borrowed reference's lifetime is guaranteed until the function returns. Only when such a borrowed reference must be stored or passed on, it must be turned into an owned reference by calling `Py_INCREF()`.

---

[2] The metaphor of "borrowing" a reference is not completely correct: the owner still has a copy of the reference.

[3] Checking that the reference count is at least 1 **does not work** --- the reference count itself could be in freed memory and may thus be reused for another object!

The object reference returned from a C function that is called from Python must be an owned reference --- ownership is transferred from the function to its caller.

Thin Ice

There are a few situations where seemingly harmless use of a borrowed reference can lead to problems. These all have to do with implicit invocations of the interpreter, which can cause the owner of a reference to dispose of it.

The first and most important case to know about is using `Py_DECREF()` on an unrelated object while borrowing a reference to a list item. For instance:

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

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

This function first borrows a reference to `list[0]`, then replaces `list[1]` with the value `0`, and finally prints the borrowed reference. Looks harmless, right? But it's not!

Let's follow the control flow into `PyList_SetItem()`. The list owns references to all its items, so when item 1 is replaced, it has to dispose of the original item 1. Now let's suppose the original item 1 was an instance of a user-defined class, and let's further suppose that the class defined a `__del__()` method. If this class instance has a reference count of 1, disposing of it will call its `__del__()` method.

Since it is written in Python, the `__del__()` method can execute arbitrary Python code. Could it perhaps do something to invalidate the reference to `item` in `bug()`? You bet! Assuming that the list passed into `bug()` is accessible to the `__del__()` method, it could execute a statement to the effect of `del list[0]`, and assuming this was the last reference to that object, it would free the memory associated with it, thereby invalidating `item`.

The solution, once you know the source of the problem, is easy: temporarily increment the reference count. The correct version of the function reads:

```
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);
}
```

This is a true story. An older version of Python contained variants of this bug and someone spent a considerable amount of time in a C debugger to figure out why his `__del__()` methods would fail...

The second case of problems with a borrowed reference is a variant involving threads. Normally, multiple threads in the Python interpreter can't get in each other's way, because there is a global lock protecting Python's entire object space. However, it is possible to temporarily release this lock using the macro `Py_BEGIN_ALLOW_THREADS`, and to re-acquire it using `Py_END_ALLOW_THREADS`. This is common around blocking I/O calls, to let other threads use the processor while waiting for the I/O to complete. Obviously, the following function has the same problem as the previous one:

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);
    Py_BEGIN_ALLOW_THREADS
    ...some blocking I/O call...
```

```
    Py_END_ALLOW_THREADS
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

NULL 指標

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

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

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

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

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

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

2.1.11 Writing Extensions in C++

It is possible to write extension modules in C++. Some restrictions apply. If the main program (the Python interpreter) is compiled and linked by the C compiler, global or static objects with constructors cannot be used. This is not a problem if the main program is linked by the C++ compiler. Functions that will be called by the Python interpreter (in particular, module initialization functions) have to be declared using extern "C". It is unnecessary to enclose the Python header files in extern "C" {...} --- they use this form already if the symbol __cplusplus is defined (all recent C++ compilers define this symbol).

2.1.12 Providing a C API for an Extension Module

Many extension modules just provide new functions and types to be used from Python, but sometimes the code in an extension module can be useful for other extension modules. For example, an extension module could implement a type "collection" which works like lists without order. Just like the standard Python list type has a C API which permits extension modules to create and manipulate lists, this new collection type should have a set of C functions for direct manipulation from other extension modules.

At first sight this seems easy: just write the functions (without declaring them static, of course), provide an appropriate header file, and document the C API. And in fact this would work if all extension modules were always linked statically with the Python interpreter. When modules are used as shared libraries, however, the symbols defined in one module may not be visible to another module. The details of visibility depend on the operating system; some systems use one global namespace for the Python interpreter and all extension modules (Windows, for example), whereas others require an explicit list of imported symbols at module link time (AIX is one example), or offer a choice of different strategies (most Unices). And even if symbols are globally visible, the module whose functions one wishes to call might not have been loaded yet!

Portability therefore requires not to make any assumptions about symbol visibility. This means that all symbols in extension modules should be declared static, except for the module's initialization function, in order to avoid name clashes with other extension modules (as discussed in section 模組的方法表和初始化函式). And it means that symbols that *should* be accessible from other extension modules must be exported in a different way.

Python provides a special mechanism to pass C-level information (pointers) from one extension module to another one: Capsules. A Capsule is a Python data type which stores a pointer (void*). Capsules can only be created and

---

[4] These guarantees don't hold when you use the "old" style calling convention --- this is still found in much existing code.

accessed via their C API, but they can be passed around like any other Python object. In particular, they can be assigned to a name in an extension module's namespace. Other extension modules can then import this module, retrieve the value of this name, and then retrieve the pointer from the Capsule.

There are many ways in which Capsules can be used to export the C API of an extension module. Each function could get its own Capsule, or all C API pointers could be stored in an array whose address is published in a Capsule. And the various tasks of storing and retrieving the pointers can be distributed in different ways between the module providing the code and the client modules.

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

In particular, Capsules used to expose C APIs should be given a name following this convention:

```
modulename.attributename
```

The convenience function `PyCapsule_Import()` makes it easy to load a C API provided via a Capsule, but only if the Capsule's name matches this convention. This behavior gives C API users a high degree of certainty that the Capsule they load contains the correct C API.

The following example demonstrates an approach that puts most of the burden on the writer of the exporting module, which is appropriate for commonly used library modules. It stores all C API pointers (just one in the example!) in an array of `void` pointers which becomes the value of a Capsule. The header file corresponding to the module provides a macro that takes care of importing the module and retrieving its C API pointers; client modules only have to call this macro before accessing the C API.

The exporting module is a modification of the `spam` module from section 一個簡單範例. The function `spam.system()` does not call the C library function `system()` directly, but a function `PySpam_System()`, which would of course do something more complicated in reality (such as adding "spam" to every command). This function `PySpam_System()` is also exported to other extension modules.

The function `PySpam_System()` is a plain C function, declared `static` like everything else:

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

The function `spam_system()` is modified in a trivial way:

```c
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);
}
```

In the beginning of the module, right after the line

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

two more lines must be added:

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

The `#define` is used to tell the header file that it is being included in the exporting module, not a client module. Finally, the module's initialization function must take care of initializing the C API pointer array:

```c
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_Add(m, "_C_API", c_api_object) < 0) {
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

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

The bulk of the work is in the header file `spammodule.h`, which looks like this:

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

/* Header file for spammodule */

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

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


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

static PySpam_System_RETURN PySpam_System PySpam_System_PROTO;

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

static void **PySpam_API;

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

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

#endif


#ifdef __cplusplus
}
#endif


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

All that a client module must do in order to have access to the function `PySpam_System()` is to call the function (or rather macro) `import_spam()` in its initialization function:

```
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;
}
```

The main disadvantage of this approach is that the file `spammodule.h` is rather complicated. However, the basic structure is the same for each function that is exported, so it has to be learned only once.

Finally it should be mentioned that Capsules offer additional functionality, which is especially useful for memory allocation and deallocation of the pointer stored in a Capsule. The details are described in the Python/C API Reference Manual in the section capsules and in the implementation of Capsules (files `Include/pycapsule.h` and `Objects/pycapsule.c` in the Python source code distribution).

[F]解

2.2 Defining Extension Types: Tutorial

Python allows the writer of a C extension module to define new types that can be manipulated from Python code, much like the built-in `str` and `list` types. The code for all extension types follows a pattern, but there are some details that you need to understand before you can get started. This document is a gentle introduction to the topic.

2.2.1 The Basics

The *CPython* runtime sees all Python objects as variables of type `PyObject*`, which serves as a "base type" for all Python objects. The `PyObject` structure itself only contains the object's *reference count* and a pointer to the object's "type object". This is where the action is; the type object determines which (C) functions get called by the interpreter when, for instance, an attribute gets looked [F] up on an object, a method called, or it is multiplied by another object. These C functions are called "type methods".

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

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

> 🛈 備⿰
>
> What we're showing here is the traditional way of defining *static* extension types. It should be adequate for most uses. The C API also allows defining heap-allocated extension types using the PyType_FromSpec() function, which isn't covered in this tutorial.

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

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

static PyTypeObject CustomType = {
    .ob_base = 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 = {
    .m_base = 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;

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

    return m;
}
```

Now that's quite a bit to take in at once, but hopefully bits will seem familiar from the previous chapter. This file defines three things:

1. What a Custom **object** contains: this is the CustomObject struct, which is allocated once for each Custom instance.

2. How the Custom **type** behaves: this is the CustomType struct, which defines a set of flags and function pointers that the interpreter inspects when specific operations are requested.

3. How to initialize the `custom` module: this is the `PyInit_custom` function and the associated `custommodule` struct.

The first bit is:

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

This is what a Custom object will contain. `PyObject_HEAD` is mandatory at the start of each object struct and defines a field called `ob_base` of type `PyObject`, containing a pointer to a type object and a reference count (these can be accessed using the macros `Py_TYPE` and `Py_REFCNT` respectively). The reason for the macro is to abstract away the layout and to enable additional fields in debug builds.

> ⓘ **備註**
>
> There is no semicolon above after the `PyObject_HEAD` macro. Be wary of adding one by accident: some compilers will complain.

Of course, objects generally store additional data besides the standard `PyObject_HEAD` boilerplate; for example, here is the definition for standard Python floats:

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

The second bit is the definition of the type object.

```c
static PyTypeObject CustomType = {
    .ob_base = 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,
};
```

> ⓘ **備註**
>
> We recommend using C99-style designated initializers as above, to avoid listing all the `PyTypeObject` fields that you don't care about and also to avoid caring about the fields' declaration order.

The actual definition of `PyTypeObject` in `object.h` has many more fields than the definition above. The remaining fields will be filled with zeros by the C compiler, and it's common practice to not specify them explicitly unless you need them.

We're going to pick it apart, one field at a time:

```c
.ob_base = PyVarObject_HEAD_INIT(NULL, 0)
```

This line is mandatory boilerplate to initialize the `ob_base` field mentioned above.

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

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

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

Note that the name is a dotted name that includes both the module name and the name of the type within the module. The module in this case is `custom` and the type is `Custom`, so we set the type name to `custom.Custom`. Using the real dotted import path is important to make your type compatible with the `pydoc` and `pickle` modules.

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

This is so that Python knows how much memory to allocate when creating new `Custom` instances. `tp_itemsize` is only used for variable-sized objects and should otherwise be zero.

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

We set the class flags to `Py_TPFLAGS_DEFAULT`.

```
.tp_flags = Py_TPFLAGS_DEFAULT,
```

All types should include this constant in their flags. It enables all of the members defined until at least Python 3.3. If you need further members, you will need to OR the corresponding flags.

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

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

To enable object creation, we have to provide a `tp_new` handler. This is the equivalent of the Python method `__new__()`, but has to be specified explicitly. In this case, we can just use the default implementation provided by the API function `PyType_GenericNew()`.

```
.tp_new = PyType_GenericNew,
```

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

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

This initializes the `Custom` type, filling in a number of members to the appropriate default values, including `ob_type` that we initially set to `NULL`.

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

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

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

That's it! All that remains is to build it; put the above code in a file called `custom.c`,

```
[build-system]
requires = ["setuptools"]
build-backend = "setuptools.build_meta"

[project]
name = "custom"
version = "1"
```

in a file called `pyproject.toml`, and

```python
from setuptools import Extension, setup
setup(ext_modules=[Extension("custom", ["custom.c"])])
```

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

```
$ python -m pip install .
```

in a shell should produce a file `custom.so` in a subdirectory and install it; now fire up Python --- you should be able to `import custom` and play around with `Custom` objects.

That wasn't so hard, was it?

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

2.2.2 Adding data and methods to the Basic example

Let's extend the basic example to add some data and methods. Let's also make the type usable as a base class. We'll create a new module, `custom2` that adds these capabilities:

```c
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <stddef.h> /* for offsetof() */

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

<div align="right">(繼續下一頁)</div>

```
            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;

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

    if (first) {
        Py_XSETREF(self->first, Py_NewRef(first));
    }
    if (last) {
        Py_XSETREF(self->last, Py_NewRef(last));
    }
    return 0;
}

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

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 = {
    .ob_base = 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 = {
    .m_base =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;

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

    return m;
}
```

This version of the module has a number of changes.

The `Custom` type now has three data attributes in its C struct, *first*, *last*, and *number*. The *first* and *last* variables are Python strings containing first and last names. The *number* attribute is a C integer.

The object structure is updated accordingly:

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

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

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

which is assigned to the `tp_dealloc` member:

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

This method first clears the reference counts of the two Python attributes. `Py_XDECREF()` correctly handles the case where its argument is `NULL` (which might happen here if `tp_new` failed midway). It then calls the `tp_free` member of the object's type (computed by `Py_TYPE(self)`) to free the object's memory. Note that the object's type might not be `CustomType`, because the object may be an instance of a subclass.

> **ⓘ 備F**
>
> The explicit cast to `destructor` above is needed because we defined `Custom_dealloc` to take a `CustomObject *` argument, but the `tp_dealloc` function pointer expects to receive a `PyObject *` argument. Otherwise, the compiler will emit a warning. This is object-oriented polymorphism, in C!

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

```
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;
}
```

and install it in the `tp_new` member:

```
.tp_new = Custom_new,
```

The `tp_new` handler is responsible for creating (as opposed to initializing) objects of the type. It is exposed in Python as the `__new__()` method. It is not required to define a `tp_new` member, and indeed many extension types will simply reuse `PyType_GenericNew()` as done in the first version of the `Custom` type above. In this case, we use the `tp_new` handler to initialize the `first` and `last` attributes to non-`NULL` default values.

`tp_new` is passed the type being instantiated (not necessarily `CustomType`, if a subclass is instantiated) and any arguments passed when the type was called, and is expected to return the instance created. `tp_new` handlers always accept positional and keyword arguments, but they often ignore the arguments, leaving the argument handling to initializer (a.k.a. `tp_init` in C or `__init__` in Python) methods.

> **ⓘ 備F**
>
> `tp_new` shouldn't call `tp_init` explicitly, as the interpreter will do it itself.

The `tp_new` implementation calls the `tp_alloc` slot to allocate memory:

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

Since memory allocation may fail, we must check the `tp_alloc` result against `NULL` before proceeding.

> ℹ️ **備註**
>
> We didn't fill the `tp_alloc` slot ourselves. Rather `PyType_Ready()` fills it for us by inheriting it from our base class, which is `object` by default. Most types use the default allocation strategy.

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

We also define an initialization function which accepts arguments to provide initial values for our instance:

```c
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;
}
```

by filling the `tp_init` slot.

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

The `tp_init` slot is exposed in Python as the `__init__()` method. It is used to initialize an object after it's created. Initializers always accept positional and keyword arguments, and they should return either `0` on success or `-1` on error.

Unlike the `tp_new` handler, there is no guarantee that `tp_init` is called at all (for example, the `pickle` module by default doesn't call `__init__()` on unpickled instances). It can also be called multiple times. Anyone can call the `__init__()` method on our objects. For this reason, we have to be extra careful when assigning the new attribute values. We might be tempted, for example to assign the `first` member like this:

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

But this would be risky. Our type doesn't restrict the type of the `first` member, so it could be any kind of object. It could have a destructor that causes code to be executed that tries to access the `first` member; or that destructor could release the *Global interpreter Lock* and let arbitrary code run in other threads that accesses and modifies our object.

To be paranoid and protect ourselves against this possibility, we almost always reassign members before decrementing their reference counts. When don't we have to do this?

- when we absolutely know that the reference count is greater than 1;

- when we know that deallocation of the object[1] will neither release the *GIL* nor cause any calls back into our type's code;

- when decrementing a reference count in a `tp_dealloc` handler on a type which doesn't support cyclic garbage collection[2].

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

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

and put the definitions in the `tp_members` slot:

```
.tp_members = Custom_members,
```

Each member definition has a member name, type, offset, access flags and documentation string. See the *Generic Attribute Management* section below for details.

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

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

```
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;
    }
```

---

[1] This is true when we know that the object is a basic type, like a string or a float.
[2] We relied on this in the `tp_dealloc` handler in this example, because our type doesn't support garbage collection.

```
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}
```

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

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

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

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

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

(note that we used the `METH_NOARGS` flag to indicate that the method is expecting no arguments other than *self*)

and assign it to the `tp_methods` slot:

```c
.tp_methods = Custom_methods,
```

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

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

We rename `PyInit_custom()` to `PyInit_custom2()`, update the module name in the `PyModuleDef` struct, and update the full class name in the `PyTypeObject` struct.

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

```python
from setuptools import Extension, setup
setup(ext_modules=[
    Extension("custom", ["custom.c"]),
    Extension("custom2", ["custom2.c"]),
])
```

and then we re-install so that we can `import custom2`:

```
$ python -m pip install .
```

2.2.3 Providing finer control over data attributes

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

```c
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <stddef.h> /* for offsetof() */
```

```c
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;
        }
        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;

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

    if (first) {
        Py_SETREF(self->first, Py_NewRef(first));
    }
    if (last) {
        Py_SETREF(self->last, Py_NewRef(last));
    }
    return 0;
}

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

```c
};

static PyObject *
Custom_getfirst(CustomObject *self, void *closure)
{
    return Py_NewRef(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_SETREF(self->first, Py_NewRef(value));
    return 0;
}

static PyObject *
Custom_getlast(CustomObject *self, void *closure)
{
    return Py_NewRef(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_SETREF(self->last, Py_NewRef(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[] = {
```

```c
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL}  /* Sentinel */
};

static PyTypeObject CustomType = {
    .ob_base = 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 = {
    .m_base = 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;

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

    return m;
}
```

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

```c
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;
    }
    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;
}
```

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

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

We create an array of `PyGetSetDef` structures:

```
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 */
};
```

and register it in the `tp_getset` slot:

```
.tp_getset = Custom_getsetters,
```

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

We also remove the member definitions for these attributes:

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

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

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

---

[3] We now know that the first and last members are strings, so perhaps we could be less careful about decrementing their reference counts, however, we accept instances of string subclasses. Even though deallocating normal strings won't call back into our objects, we can't guarantee that deallocating an instance of a string subclass won't call back into our objects.

```c
    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;
}
```

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

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

2.2.4 Supporting cyclic garbage collection

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

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

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

In the second version of the `Custom` example, we allowed any kind of object to be stored in the `first` or `last` attributes[4]. Besides, in the second and third versions, we allowed subclassing `Custom`, and subclasses may add arbitrary attributes. For any of those two reasons, `Custom` objects can participate in cycles:

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

To allow a `Custom` instance participating in a reference cycle to be properly detected and collected by the cyclic GC, our `Custom` type needs to fill two additional slots and to enable a flag that enables these slots:

```c
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <stddef.h> /* for offsetof() */
```

---

[4] Also, even with our attributes restricted to strings instances, the user could pass arbitrary `str` subclasses and therefore still create reference cycles.

```c
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;
}

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

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi", kwlist,
                                     &first, &last,
                                     &self->number))
```

```
        return -1;

    if (first) {
        Py_SETREF(self->first, Py_NewRef(first));
    }
    if (last) {
        Py_SETREF(self->last, Py_NewRef(last));
    }
    return 0;
}

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

static PyObject *
Custom_getfirst(CustomObject *self, void *closure)
{
    return Py_NewRef(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_XSETREF(self->first, Py_NewRef(value));
    return 0;
}

static PyObject *
Custom_getlast(CustomObject *self, void *closure)
{
    return Py_NewRef(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_XSETREF(self->last, Py_NewRef(value));
    return 0;
}
```

```c
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 = {
    .ob_base = 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,
    .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 = {
    .m_base = 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;

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

```
    return m;
}
```

First, the traversal method lets the cyclic GC know about subobjects that could participate in cycles:

```
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;
}
```

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

Python provides a `Py_VISIT()` macro that automates calling visit functions. With `Py_VISIT()`, we can minimize the amount of boilerplate in `Custom_traverse`:

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

> ℹ 備註
>
> The `tp_traverse` implementation must name its arguments exactly *visit* and *arg* in order to use `Py_VISIT()`.

Second, we need to provide a method for clearing any subobjects that can participate in cycles:

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

Notice the use of the `Py_CLEAR()` macro. It is the recommended and safe way to clear data attributes of arbitrary types while decrementing their reference counts. If you were to call `Py_XDECREF()` instead on the attribute before setting it to `NULL`, there is a possibility that the attribute's destructor would call back into code that reads the attribute again (*especially* if there is a reference cycle).

> ℹ 備註

> You could emulate `Py_CLEAR()` by writing:
>
> ```
> PyObject *tmp;
> tmp = self->first;
> self->first = NULL;
> Py_XDECREF(tmp);
> ```
>
> Nevertheless, it is much easier and less error-prone to always use `Py_CLEAR()` when deleting an attribute. Don't try to micro-optimize at the expense of robustness!

The deallocator `Custom_dealloc` may call arbitrary code when clearing attributes. It means the circular GC can be triggered inside the function. Since the GC assumes reference count is not zero, we need to untrack the object from the GC by calling `PyObject_GC_UnTrack()` before clearing members. Here is our reimplemented deallocator using `PyObject_GC_UnTrack()` and `Custom_clear`:

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

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

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

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

2.2.5 Subclassing other types

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

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

```
>>> import 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)
{
```

(繼續下一頁)

```c
    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)
    .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;

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

    return m;
}
```

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

```c
typedef struct {
```

```
    PyListObject list;
    int state;
} SubListObject;
```

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

When a Python object is a `SubList` instance, its `PyObject *` pointer can be safely cast to both `PyListObject *` and `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;
    return 0;
}
```

We see above how to call through to the `__init__()` method of the base type.

This pattern is important when writing a type with custom `tp_new` and `tp_dealloc` members. The `tp_new` handler should not actually create the memory for the object with its `tp_alloc`, but let the base class handle it by calling its own `tp_new`.

The `PyTypeObject` struct supports a `tp_base` specifying the type's concrete base class. Due to cross-platform compiler issues, you can't fill that field directly with a reference to `PyList_Type`; it should be done later in the module initialization function:

```
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;

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

    return m;
}
```

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

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

註解

2.3 Defining Extension Types: Assorted Topics

This section aims to give a quick fly-by on the various type methods you can implement and what they do.

Here is the definition of `PyTypeObject`, with some fields only used in debug builds omitted:

```c
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;
    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;
    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;

    /* bitset of which type-watchers care about this type */
    unsigned char tp_watched;
} PyTypeObject;
```

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

As you probably expect by now, we're going to go over this and give more information about the various handlers. We won't go in the order they are defined in the structure, because there is a lot of historical baggage that impacts the ordering of the fields. It's often easiest to find an example that includes the fields you need and then change the values to suit your new type.

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

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

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

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

```
const char *tp_doc;
```

Here you can put a string (or its address) that you want returned when the Python script references obj.__doc__ to retrieve the doc string.

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

2.3.1 Finalization and De-allocation

```
destructor tp_dealloc;
```

This function is called when the reference count of the instance of your type is reduced to zero and the Python interpreter wants to reclaim it. If your type has memory to free or other clean-up to perform, you can put it here. The object itself needs to be freed here as well. Here is an example of this function:

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

If your type supports garbage collection, the destructor should call `PyObject_GC_UnTrack()` before clearing any member fields:

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

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

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

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

        /* 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);
}
```

> ℹ️ **備⿰Ｆ**
>
> There are limitations to what you can safely do in a deallocator function. First, if your type supports garbage collection (using `tp_traverse` and/or `tp_clear`), some of the object's members can have been cleared or finalized by the time `tp_dealloc` is called. Second, in `tp_dealloc`, your object is in an unstable state: its reference count is equal to zero. Any call to a non-trivial object or API (as in the example above) might end up calling `tp_dealloc` again, causing a double free and a crash.

Starting with Python 3.4, it is recommended not to put any complex finalization code in `tp_dealloc`, and instead use the new `tp_finalize` type method.

> **➔ 也參考**
>
> **PEP 442** explains the new finalization scheme.

2.3.2 Object Presentation

In Python, there are two ways to generate a textual representation of an object: the `repr()` function, and the `str()` function. (The `print()` function just calls `str()`.) These handlers are both optional.

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

The `tp_repr` handler should return a string object containing a representation of the instance for which it is called. Here is a simple example:

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

If no `tp_repr` handler is specified, the interpreter will supply a representation that uses the type's `tp_name` and a uniquely identifying value for the object.

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

以下是個簡單的範例：

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

2.3.3 Attribute Management

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

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

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

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

Generic Attribute Management

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

1. The name of the attributes must be known when `PyType_Ready()` is called.

2. No special processing is needed to record that an attribute was looked up or set, nor do actions need to be taken based on the value.

Note that this list does not place any restrictions on the values of the attributes, when the values are computed, or how relevant data is stored.

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

The tables are declared as three fields of the type object:

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

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

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

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

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

```
typedef struct PyMemberDef {
    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 a type code like `Py_T_INT` or `Py_T_DOUBLE`; 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: you can set it to `Py_READONLY` to prevent Python code from setting it.

An interesting advantage of using the `tp_members` table to build descriptors that are used at runtime is that any attribute defined this way can have an associated doc string simply by providing the text in the table. An application

can use the introspection API to retrieve the descriptor from the class object, and get the doc string using its `__doc__` attribute.

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

Type-specific Attribute Management

For simplicity, only the `char*` version will be demonstrated here; the type of the name parameter is the only difference between the `char*` and `PyObject*` flavors of the interface. This example effectively does the same thing as the generic example above, but does not use the generic support added in Python 2.2. It explains how the handler functions are called, so that if you do need to extend their functionality, you'll understand what needs to be done.

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

舉例來說:

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

    PyErr_Format(PyExc_AttributeError,
                "'%.100s' object has no attribute '%.400s'",
                Py_TYPE(obj)->tp_name, name);
    return NULL;
}
```

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

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

2.3.4 Object Comparison

```
richcmpfunc tp_richcompare;
```

The `tp_richcompare` handler is called when comparisons are needed. It is analogous to the rich comparison methods, like `__lt__()`, and also called by `PyObject_RichCompare()` and `PyObject_RichCompareBool()`.

This function is called with two Python objects and the operator as arguments, where the operator is one of `Py_EQ`, `Py_NE`, `Py_LE`, `Py_GE`, `Py_LT` or `Py_GT`. It should compare the two objects with respect to the specified operator and return `Py_True` or `Py_False` if the comparison is successful, `Py_NotImplemented` to indicate that comparison is not implemented and the other object's comparison method should be tried, or `NULL` if an exception was set.

Here is a sample implementation, for a datatype that is considered equal if the size of an internal pointer is equal:

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

(繼續下一頁)

```
    /* 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 Abstract Protocol Support

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

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

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

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

```
hashfunc tp_hash;
```

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

```
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` is a signed integer type with a platform-varying width. Returning `-1` from `tp_hash` indicates an error, which is why you should be careful to avoid returning it when hash computation is successful, as seen above.

```
ternaryfunc tp_call;
```

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

This function takes three arguments:

1. *self* is the instance of the data type which is the subject of the call. If the call is `obj1('hello')`, then *self* is `obj1`.

2. *args* is a tuple containing the arguments to the call. You can use `PyArg_ParseTuple()` to extract the arguments.

3. *kwds* is a dictionary of keyword arguments that were passed. If this is non-`NULL` and you support keyword arguments, use `PyArg_ParseTupleAndKeywords()` to extract the arguments. If you do not want to support keyword arguments and this is non-`NULL`, raise a `TypeError` with a message saying that keyword arguments are not supported.

Here is a toy `tp_call` implementation:

```c
static PyObject *
newdatatype_call(newdatatypeobject *obj, 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;
}
```

```c
/* ⬚代器 */
getiterfunc tp_iter;
iternextfunc tp_iternext;
```

These functions provide support for the iterator protocol. Both handlers take exactly one parameter, the instance for which they are being called, and return a new reference. In the case of an error, they should set an exception and return `NULL`. `tp_iter` corresponds to the Python `__iter__()` method, while `tp_iternext` corresponds to the Python `__next__()` method.

Any *iterable* object must implement the `tp_iter` handler, which must return an *iterator* object. Here the same guidelines apply as for Python classes:

- For collections (such as lists and tuples) which can support multiple independent iterators, a new iterator should be created and returned by each call to `tp_iter`.

- Objects which can only be iterated over once (usually due to side effects of iteration, such as file objects) can implement `tp_iter` by returning a new reference to themselves -- and should also therefore implement the `tp_iternext` handler.

Any *iterator* object should implement both `tp_iter` and `tp_iternext`. An iterator's `tp_iter` handler should return a new reference to the iterator. Its `tp_iternext` handler should return a new reference to the next object in the iteration, if there is one. If the iteration has reached the end, `tp_iternext` may return `NULL` without setting an exception, or it may set `StopIteration` *in addition* to returning `NULL`; avoiding the exception can yield slightly better performance. If an actual error occurs, `tp_iternext` should always set an exception and return `NULL`.

2.3.6 Weak Reference Support

One of the goals of Python's weak reference implementation is to allow any type to participate in the weak reference mechanism without incurring the overhead on performance-critical objects (such as numbers).

> **↪ 也參考**
>
> Documentation for the `weakref` module.

For an object to be weakly referenceable, the extension type must set the `Py_TPFLAGS_MANAGED_WEAKREF` bit of the `tp_flags` field. The legacy `tp_weaklistoffset` field should be left as zero.

Concretely, here is how the statically declared type object would look:

```c
static PyTypeObject TrivialType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    /* ... other members omitted for brevity ... */
    .tp_flags = Py_TPFLAGS_MANAGED_WEAKREF | ...,
};
```

The only further addition is that `tp_dealloc` needs to clear any weak references (by calling `PyObject_ClearWeakRefs()`):

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

2.3.7 More Suggestions

In order to learn how to implement any specific method for your new data type, get the *CPython* source code. Go to the `Objects` directory, then search the C source files for `tp_` plus the function you want (for example, `tp_richcompare`). You will find examples of the function you want to implement.

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

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

> **↪ 也參考**
>
> **Download CPython source releases.**
>   https://www.python.org/downloads/source/
>
> **The CPython project on GitHub, where the CPython source code is developed.**
>   https://github.com/python/cpython

2.4 建立 C 與 C++ 擴充套件

一個 CPython 的 C 擴充套件是一個共用函式庫（例如在 Linux 上的 `.so` 檔案，在 Windows 上的 `.pyd`），會匯出一個初始化函式。

要能⊡被引入，共用函式庫必須在 `PYTHONPATH` 上可用，⊡且必須以模組名稱命名，⊡且必須有適當的副檔名。使用 setuptools 時，正確的檔名會自動⊡生。

初始化函式具有簽名：

PyObject *`PyInit_modulename`(void)

它回傳一個完全初始化的模組，或一個 `PyModuleDef` 實例。詳細資訊請參見 initializing-modules。

對於僅包含 ASCII 名稱的模組，函式必須以 `PyInit_<modulename>` 命名，其中 `<modulename>` 要替⊡⊡模組的名稱。當使用 multi-phase-initialization 時，允許非 ASCII 模組名稱。在這種情⊡下，初始化函式名稱是 `PyInitU_<modulename>`，其中 `<modulename>` 使用 Python 的 *punycode* 編碼，⊡將連字符號替⊡⊡底⊡。在 Python 中：

```python
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
```

可以透過定義多個初始化函式，來從單一共用函式庫中匯出多個模組。然而要引入它們需要使用符號連結或自訂引入器，因⊡預設只會找到對應於檔名的函式。詳細資訊請參見 **PEP 489** 中的 *"Multiple modules in one library"* 部分。

2.4.1 用 setuptools 建置 C 與 C++ 擴充套件

Python 3.12 與之後的版本不再帶有 distutils。請在 https://setuptools.readthedocs.io/en/latest/setuptools.html 上參⊡ `setuptools` 文件，以了解如何使用 setuptools 建置和發⊡ C/C++ 擴充套件。

2.5 建置 Windows 上的 C 和 C++ 擴充

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

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

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

2.5.1 A Cookbook Approach

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

2.5.2 Differences Between Unix and Windows

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

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

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

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

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

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

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

2.5.3 Using DLLs in Practice

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

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

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

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

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

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

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

CHAPTER 3

在更大的應用程式中嵌入 CPython 運行環境 (runtime)

有時候，相較於建立一個擴充，使其在 Python 直譯器中可作⊞主應用程式運行，還不如將 CPython 運行環境嵌入至一個更大的應用程式中更可取。本節將涵蓋一些要成功完成此任務所涉及的細節。

3.1 在其它 App ⊞嵌入 Python

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

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

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

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

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

3.1.1 Very High Level Embedding

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

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

int
main(int argc, char *argv[])
{
    PyStatus status;
    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* optional but recommended */
    status = PyConfig_SetBytesString(&config, &config.program_name, argv[0]);
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);

    PyRun_SimpleString("from time import time,ctime\n"
                       "print('Today is', ctime(time()))\n");
    if (Py_FinalizeEx() < 0) {
        exit(120);
    }
    return 0;

 exception:
    PyConfig_Clear(&config);
    Py_ExitStatusException(status);
}
```

> ℹ️ **備⿕**
>
> #define PY_SSIZE_T_CLEAN was used to indicate that Py_ssize_t should be used in some APIs instead of int. It is not necessary since Python 3.13, but we keep it here for backward compatibility. See arg-parsing-string-and-buffers for a description of this macro.

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

3.1.2 Beyond Very High Level Embedding: An overview

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

It should be noted that extending Python and embedding Python is quite the same activity, despite the different intent.

Most topics discussed in the previous chapters are still valid. To show this, consider what the extension code from Python to C really does:

1. Convert data values from Python to C,

2. Perform a function call to a C routine using the converted values, and

3. Convert the data values from the call from C to Python.

When embedding Python, the interface code does:

1. Convert data values from C to Python,

2. Perform a function call to a Python interface routine using the converted values, and

3. Convert the data values from the call from Python to C.

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

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

3.1.3 Pure Embedding

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

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

```c
#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);
```

```c
                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;
        }
    }
    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;
}
```

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

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

then the result should be:

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

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

```
Py_Initialize();
pName = PyUnicode_DecodeFSDefault(argv[1]);
```

```
/* Error checking of pName left out */
pModule = PyImport_Import(pName);
```

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

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

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

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

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

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

3.1.4 Extending Embedded Python

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

```
static int numargs=0;

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

Insert the above code just above the `main()` function. Also, insert the following two statements before the call to `Py_Initialize()`:

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

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

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

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

3.1.5 Embedding Python in C++

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

3.1.6 Compiling and Linking under Unix-like systems

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

To find out the required compiler and linker flags, you can execute the `pythonX.Y-config` script which is generated as part of the installation process (a `python3-config` script may also be available). This script has several options, of which the following will be directly useful to you:

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

  ```
  $ /opt/bin/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` will give you the recommended flags when linking:

  ```
  $ /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
  ```

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

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

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

術語表

>>>
　　互動式 shell 的預設 Python 提示字元。常見於能在直譯器中以互動方式被執行的程式碼範例。

...
　　可以表示：

- 在一個被縮排的程式碼區塊、在一對匹配的左右定界符（delimiter，例如括號、方括號、花括號或三引號）回部，或是在指定一個裝飾器 (decorator) 之後，要輸入程式碼時，*互動式* shell 顯示的預設 Python 提示字元。
- 回建常數 `Ellipsis`。

abstract base class（抽象基底類回）
　　抽象基底類回（又稱回 ABC）提供了一種定義介面的方法，作回*duck-typing*（鴨子型回）的補充。其他類似的技術，像是 `hasattr()`，則顯得笨拙或是帶有細微的錯誤（例如使用魔術方法 (magic method)）。ABC 回用回擬的 subclass（子類回），它們回不繼承自另一個 class（類回），但仍可被 `isinstance()` 及 `issubclass()` 辨識；請參回 abc 模組的回明文件。Python 有許多回建的 ABC，用於資料結構（在 `collections.abc` 模組）、數字（在 `numbers` 模組）、串流（在 `io` 模組）及 import 尋檢器和載入器（在 `importlib.abc` 模組）。你可以使用 abc 模組建立自己的 ABC。

annotation（回釋）
　　一個與變數、class 屬性、函式的參數或回傳值相關聯的標回。照慣例，它被用來作回*type hint*（型回提示）。

　　在執行環境 (runtime)，區域變數的回釋無法被存取，但全域變數、class 屬性和函式的回解，會分回被儲存在模組、class 和函式的 `__annotations__` 特殊屬性中。

　　請參回*variable annotation*、*function annotation*、**PEP 484** 和 **PEP 526**，這些章節皆有此功能的回明。關於回釋的最佳實踐方法也請參回 annotations-howto。

argument（引數）
　　呼叫函式時被傳遞給*function*（或*method*）的值。引數有兩種：

- 關鍵字引數 *(keyword argument)*：在函式呼叫中，以識回字（identifier，例如 `name=`）開頭的引數，或是以 `**` 後面 dictionary（字典）回的值被傳遞的引數。例如，3 和 5 都是以下 `complex()` 呼叫中的關鍵字引數：

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- 位置引數 (*positional argument*)：不是關鍵字引數的引數。位置引數可在一個引數列表的起始處出現，和（或）作為 `*` 之後的 *iterable*（可迭代物件）中的元素被傳遞。例如，3 和 5 都是以下呼叫中的位置引數：

```
complex(3, 5)
complex(*(3, 5))
```

引數會被指定給函式主體中的附名區域變數。關於支配這個指定過程的規則，請參見 calls 章節。在語法上，任何運算式都可以被用來表示一個引數；其評估值會被指定給區域變數。

另請參見術語表的 *parameter*（參數）條目、常見問題中的引數和參數之間的差異，以及 **PEP 362**。

asynchronous context manager（非同步情境管理器）
　　一個可以控制 `async with` 陳述式中所見環境的物件，而它是透過定義 `__aenter__()` 和 `__aexit__()` method（方法）來控制的。由 **PEP 492** 引入。

asynchronous generator（非同步產生器）
　　一個會回傳 *asynchronous generator iterator*（非同步產生器產代器）的函式。它看起來像一個以 `async def` 定義的協程函式 (coroutine function)，但不同的是它包含了 `yield` 運算式，能生成一系列可用於 `async for` 迴圈的值。

　　這個術語通常用來表示一個非同步產生器函式，但在某些情境中，也可能是表示非同步產生器產代器 (*asynchronous generator iterator*)。萬一想表達的意思不夠清楚，那就使用完整的術語，以避免歧義。

　　一個非同步產生器函式可能包含 `await` 運算式，以及 `async for` 和 `async with` 陳述式。

asynchronous generator iterator（非同步產生器產代器）
　　一個由 *asynchronous generator*（非同步產生器）函式所建立的物件。

　　這是一個 *asynchronous iterator*（非同步產代器），當它以 `__anext__()` method 被呼叫時，會回傳一個可等待物件 (awaitable object)，該物件將執行非同步產生器的函式主體，直到遇到下一個 `yield` 運算式。

　　每個 `yield` 會暫停處理程序，並記住執行狀態（包括區域變數及擱置中的 try 陳述式）。當非同步產生器產代器以另一個被 `__anext__()` 回傳的可等待物件有效地回復時，它會從停止的地方繼續執行。請參見 **PEP 492** 和 **PEP 525**。

asynchronous iterable（非同步可迭代物件）
　　一個物件，它可以在 `async for` 陳述式中被使用。必須從它的 `__aiter__()` method 回傳一個 *asynchronous iterator*（非同步產代器）。由 **PEP 492** 引入。

asynchronous iterator（非同步產代器）
　　一個實作 `__aiter__()` 和 `__anext__()` method 的物件。`__anext__()` 必須回傳一個 *awaitable*（可等待物件）。`async for` 會解析非同步產代器的 `__anext__()` method 所回傳的可等待物件，直到它引發 `StopAsyncIteration` 例外。由 **PEP 492** 引入。

attribute（屬性）
　　一個與某物件相關聯的值，該值大多能透過使用點分隔運算式 (dotted expression) 的名稱被參照。例如，如果物件 *o* 有一個屬性 *a*，則該屬性能以 *o.a* 被參照。

　　如果一個物件允許，給予該物件一個名稱不是由 identifiers 所定義之識別符 (identifier) 的屬性是有可能的，例如使用 `setattr()`。像這樣的屬性將無法使用點分隔運算式來存取，而是需要使用 `getattr()` 來取得它。

awaitable（可等待物件）
　　一個可以在 `await` 運算式中被使用的物件。它可以是一個 *coroutine*(協程)，或是一個有 `__await__()` method 的物件。另請參見 **PEP 492**。

BDFL
　　Benevolent Dictator For Life（終身仁慈獨裁者），又名 Guido van Rossum，Python 的創造者。

binary file（二進位檔案）
　　一個能夠讀取和寫入 *bytes-like objects*（類位元組串物件）的 *file object*（檔案物件）。二進位檔案的例子有：以二進位模式（`'rb'`、`'wb'` 或 `'rb+'`）開啟的檔案、`sys.stdin.buffer`、`sys.stdout.buffer`，以及 `io.BytesIO` 和 `gzip.GzipFile` 實例。

另請參⬚*text file*（文字檔案），它是一個能⬚讀取和寫入 `str` 物件的檔案物件。

borrowed reference（借用參照）

在 Python 的 C API 中，借用參照是一個對物件的參照，其中使用該物件的程式碼⬚不擁有這個參照。如果該物件被銷⬚，它會成⬚一個迷途指標 (dangling pointer)。例如，一次垃圾回收 (garbage collection) 可以移除對物件的最後一個*strong reference*（⬚參照），而將該物件銷⬚。

對*borrowed reference* 呼叫 `Py_INCREF()` 以將它原地 (in-place) 轉⬚⬚*strong reference* 是被建議的做法，除非該物件不能在最後一次使用借用參照之前被銷⬚。`Py_NewRef()` 函式可用於建立一個新的*strong reference*。

bytes-like object（類位元組串物件）

一個支援 bufferobjects 且能⬚匯出 C-*contiguous* 緩衝區的物件。這包括所有的 `bytes`、`bytearray` 和 `array.array` 物件，以及許多常見的 `memoryview` 物件。類位元組串物件可用於處理二進位資料的各種運算；這些運算包括壓縮、儲存至二進位檔案和透過 socket（插座）發送。

有些運算需要二進位資料是可變的。⬚明文件通常會將這些物件稱⬚「可讀寫的類位元組串物件」。可變緩衝區的物件包括 `bytearray`，以及 `bytearray` 的 `memoryview`。其他的運算需要讓二進位資料被儲存在不可變物件（「唯讀的類位元組串物件」）中；這些物件包括 `bytes`，以及 `bytes` 物件的 `memoryview`。

bytecode（位元組碼）

Python 的原始碼會被編譯成位元組碼，它是 Python 程式在 CPython 直譯器中的⬚部表示法。該位元組碼也會被暫存在 `.pyc` 檔案中，以便第二次執行同一個檔案時能⬚更快速（可以不用從原始碼重新編譯⬚位元組碼）。這種「中間語言 (intermediate language)」據⬚是運行在一個*virtual machine*（⬚擬機器）上，該⬚擬機器會執行與每個位元組碼對應的機器碼 (machine code)。要注意的是，位元組碼理論上是無法在不同的 Python ⬚擬機器之間運作的，也不能在不同版本的 Python 之間保持穩定。

位元組碼的指令列表可以在 dis 模組的⬚明文件中找到。

callable（可呼叫物件）

一個 callable 是可以被呼叫的物件，呼叫時可能以下列形式帶有一組引數（請見*argument*）：

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

一個*function* 與其延伸的*method* 都是 callable。一個有實作 `__call__()` 方法的 class 之實例也是個 callable。

callback（回呼）

作⬚引數被傳遞的一個副程式 (subroutine) 函式，會在未來的某個時間點被執行。

class（類⬚）

一個用於建立使用者定義物件的模板。Class 的定義通常會包含 method 的定義，這些 method 可以在 class 的實例上進行操作。

class variable（類⬚變數）

一個在 class 中被定義，且應該只能在 class 層次（意即不是在 class 的實例中）被修改的變數。

closure variable（閉包變數）

從外部作用域中定義且從巢狀作用域參照的自由變數，不是於 runtime 從全域或⬚建命名空間解析。可以使用 `nonlocal` 關鍵字明確定義以允許寫入存取，或者如果僅需讀取變數則隱式定義即可。

例如在下面程式碼中的 `inner` 函式中，`x` 和 `print` 都是自由變數，但只有 `x` 是閉包變數：

```python
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

由於 `codeobject.co_freevars` 屬性（⬚管名稱如此，但它僅包含閉包變數的名稱，而不是列出所有參照的自由變數），當預期含義是特指閉包變數時，有時候甚至也會使用更通用的自由變數一詞。

complex number（複數）

一個我們熟悉的實數系統的擴充，在此所有數字都會被表示為一個實部和一個虛部之和。虛數就是虛數單位（-1 的平方根）的實數倍，此單位通常在數學中被寫為 i，在工程學中被寫為 j。Python 內建了對複數的支援，它是用後者的記法來表示複數；虛部會帶著一個後綴的 j 被編寫，例如 3+1j。若要將 math 模組內的工具等效地用於複數，請使用 cmath 模組。複數的使用是一個相當進階的數學功能。如果你沒有察覺到對它們的需求，那你幾乎能確定你可以安全地忽略它們。

context（情境）

This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a *context manager* via a `with` statement.

- The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.

- 一個 `contextvars.Context` 物件。另請參閱 *current context*。

context management protocol（情境管理協定）

由 `with` 陳述式所呼叫的 `__enter__()` 和 `__exit__()` 方法。另請參閱 **PEP 343**。

context manager（情境管理器）

An object which implements the *context management protocol* and controls the environment seen in a `with` statement. See **PEP 343**.

context variable（情境變數）

A variable whose value depends on which context is the *current context*. Values are accessed via `contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

contiguous（連續的）

如果一個緩衝區是 *C-contiguous* 或是 *Fortran contiguous*，則它會確切地被視為是連續的。零維 (zero-dimensional) 的緩衝區都是 C 及 Fortran contiguous。在一維 (one-dimensional) 陣列中，各項目必須在記憶體中彼此相鄰地排列，而其索引順序是從零開始遞增。在多維的 (multidimensional) C-contiguous 陣列中，按記憶體位址的順序訪問各個項目時，最後一個索引的變化最快。然而，在 Fortran contiguous 陣列中，第一個索引的變化最快。

coroutine（協程）

協程是副程式 (subroutine) 的一種更為廣義的形式。副程式是在某個時間點被進入並在另一個時間點被退出。協程可以在許多不同的時間點被進入、退出和回復。它們能夠以 `async def` 陳述式被實作。另請參閱 **PEP 492**。

coroutine function（協程函式）

一個回傳 *coroutine*（協程）物件的函式。一個協程函式能以 `async def` 陳述式被定義，且可能會包含 `await`、`async for` 和 `async with` 關鍵字。這些關鍵字由 **PEP 492** 引入。

CPython

Python 程式語言的標準實作 (canonical implementation)，被發布在 python.org 上。「CPython」這個術語在必要時被使用，以區分此實作與其它語言的實作，例如 Jython 或 IronPython。

current context

The *context* (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

decorator（裝飾器）

一個函式，它會回傳另一個函式，通常它會使用 `@wrapper` 語法，被應用為一種函式的變換 (function transformation)。裝飾器的常見範例是 `classmethod()` 和 `staticmethod()`。

裝飾器語法只是語法糖。以下兩個函式定義在語義上是等效的：

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

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

Class 也存在相同的概念，但在那⃞比較不常用。關於裝飾器的更多⃞容，請參⃞函式定義和 class 定義的⃞明文件。

descriptor（描述器）

任何定義了 `__get__()`、`__set__()` 或 `__delete__()` method 的物件。當一個 class 屬性是一個描述器時，它的特殊連結行⃞會在屬性查找時被觸發。通常，使用 *a.b* 來取得、設定或⃞除某個屬性時，會在 *a* 的 class 字典中查找名稱⃞ *b* 的物件，但如果 *b* 是一個描述器，則相對應的描述器 method 會被呼叫。對描述器的理解是深入理解 Python 的關鍵，因⃞它們是許多功能的基礎，這些功能包括函式、method、屬性 (property)、class method、⃞態 method，以及對 super class（父類⃞）的參照。

關於描述器 method 的更多資訊，請參⃞ descriptors 或描述器使用指南。

dictionary（字典）

一個關聯陣列 (associative array)，其中任意的鍵會被對映到值。鍵可以是任何帶有 `__hash__()` 和 `__eq__()` method 的物件。在 Perl 中被稱⃞雜⃞ (hash)。

dictionary comprehension（字典綜合運算）

一種緊密的方法，用來處理一個可⃞代物件中的全部或部分元素，⃞將處理結果以一個字典回傳。`results = {n: n ** 2 for n in range(10)}` 會⃞生一個字典，它包含了鍵 n 對映到值 n ** 2。請參⃞ comprehensions。

dictionary view（字典檢視）

從 `dict.keys()`、`dict.values()` 及 `dict.items()` 回傳的物件被稱⃞字典檢視。它們提供了字典中項目的動態檢視，這表示當字典有變動時，該檢視會反映這些變動。若要⃞制將字典檢視轉⃞完整的 list（串列），須使用 `list(dictview)`。請參⃞ dict-views。

docstring（⃞明字串）

一個在 class、函式或模組中，作⃞第一個運算式出現的字串文本。雖然它在套件執行時會被忽略，但它會被編譯器辨識，⃞被放入所屬 class、函式或模組的 `__doc__` 屬性中。由於⃞明字串可以透過⃞省 (introspection) 來⃞覽，因此它是物件的⃞明文件存放的標准位置。

duck-typing（鴨子型⃞）

一種程式設計風格，它不是藉由檢查一個物件的型⃞來確定它是否具有正確的介面；取而代之的是，method 或屬性會單純地被呼叫或使用。（「如果它看起來像一⃞鴨子而且叫起來像一⃞鴨子，那⃞它一定是一⃞鴨子。」）因⃞⃞調介面而非特定型⃞，精心設計的程式碼能讓多形替代 (polymorphic substitution) 來增進它的靈活性。鴨子型⃞要避免使用 `type()` 或 `isinstance()` 進行測試。（但是請注意，鴨子型⃞可以用*抽象基底類⃞ (abstract base class)* 來補充。）然而，它通常會⃞用 `hasattr()` 測試，或是*EAFP* 程式設計風格。

EAFP

Easier to ask for forgiveness than permission.（請求寬恕比請求許可更容易。）這種常見的 Python 編碼風格會先假設有效的鍵或屬性的存在，⃞在該假設被推翻時再捕獲例外。這種乾⃞且快速的風格，其特色是存在許多的 `try` 和 `except` 陳述式。該技術與許多其他語言（例如 C）常見的*LBYL* 風格形成了對比。

expression（運算式）

一段可以被評估⃞求值的語法。⃞句話⃞，一個運算式就是文字、名稱、屬性存取、運算子或函式呼叫等運算式元件的累積，而這些元件都能回傳一個值。與許多其他語言不同的是，⃞非所有的 Python 語言構造都是運算式。另外有一些*statement*（陳述式）不能被用作運算式，例如 `while`。賦值 (assignment) 也是陳述式，而不是運算式。

extension module（擴充模組）

一個以 C 或 C++ 編寫的模組，它使用 Python 的 C API 來與核心及使用者程式碼進行互動。

f-string（f 字串）

以 `'f'` 或 `'F'` ⃞前綴的字串文本通常被稱⃞「f 字串」，它是格式化的字串文本的縮寫。另請參⃞ **PEP 498**。

file object（檔案物件）

一個讓使用者透過檔案導向 (file-oriented) API（如 `read()` 或 `write()` 等 method）來操作底層資源的物件。根據檔案物件被建立的方式，它能⿰協調對真實磁碟檔案或是其他類型的儲存器或通訊裝置（例如標准輸入／輸出、記憶體⿰緩衝區、socket（插座）、管⿰(pipe) 等）的存取。檔案物件也被稱⿰類檔案物件 *(file-like object)* 或串流 *(stream)*。

實際上，有三種檔案物件：原始的二進位檔案、緩衝的二進位檔案和文字檔案。它們的介面在 `io` 模組中被定義。建立檔案物件的標准方法是使用 `open()` 函式。

file-like object（類檔案物件）

file object（檔案物件）的同義字。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）

Python 所使用的一種編碼和錯誤處理函式，用來解碼來自作業系統的位元組，以及將 Unicode 編碼到作業系統。

檔案系統編碼必須保證能成功解碼所有小於 128 的位元組。如果檔案系統編碼無法提供此保證，則 API 函式會引發 `UnicodeError`。

`sys.getfilesystemencoding()` 和 `sys.getfilesystemencodeerrors()` 函式可用於取得檔案系統編碼和錯誤處理函式。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）會在 Python ⿰動時由 `PyConfig_Read()` 函式來配置：請參⿰ `filesystem_encoding`，以及 `PyConfig` 的成員 `filesystem_errors`。

另請參⿰*locale encoding*（區域編碼）。

finder（尋檢器）

一個物件，它會嘗試⿰正在被 import 的模組尋找*loader*（載入器）。

有兩種類型的尋檢器：*元路徑尋檢器 (meta path finder)* 會使用 `sys.meta_path`，而*路徑項目尋檢器 (path entry finder)* 會使用 `sys.path_hooks`。

請參⿰ finders-and-loaders 和 `importlib` 以了解更多細節。

floor division（向下取整除法）

向下無條件舍去到最接近整數的數學除法。向下取整除法的運算子是 `//`。例如，運算式 `11 // 4` 的計算結果⿰ `2`，與 float（浮點數）真除法所回傳的 `2.75` 不同。請注意，`(-11) // 4` 的結果是 `-3`，因⿰是 `-2.75` 被向下無條件舍去。請參⿰ **PEP 238**。

free threading（自由執行緒）

⿰一種執行緒模型，多個執行緒可以在同一直譯器中同時運行 Python 位元組碼。這與全域直譯器鎖形成對比，後者一次只允許一個執行緒執行 Python 位元組碼。請參⿰ **PEP 703**。

free variable（自由變數）

Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

function（函式）

一連串的陳述式，它能⿰向呼叫者回傳一些值。它也可以被傳遞零個或多個引數，這些引數可被使用於函式本體的執行。另請參⿰*parameter*（參數）、*method*（方法），以及 function 章節。

function annotation（函式⿰釋）

函式參數或回傳值的一個*annotation*（⿰釋）。

函式⿰釋通常被使用於型⿰提示：例如，這個函式預期會得到兩個 `int` 引數，⿰會有一個 `int` 回傳值：

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

函式⿰釋的語法在 function 章節有詳細解釋。

請參閱 *variable annotation* 和 **PEP 484**，皆有此功能的描述。關於註釋的最佳實踐方法，另請參閱 annotations-howto。

__future__

future 陳述式：`from __future__ import <feature>`，會指示編譯器使用那些在 Python 未來的發布版本中將成為標準的語法或語義，來編譯當前的模組。而 `__future__` 模組則記錄了 *feature*（功能）可能的值。透過 import 此模組並對其變數求值，你可以看見一個新的功能是何時首次被新增到此語言中，以及它何時將會（或已經）成為預設的功能：

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

garbage collection（垃圾回收）

當記憶體不再被使用時，將其釋放的過程。Python 執行垃圾回收，是透過參照計數 (reference counting)，以及一個能夠檢測和中斷參照循環 (reference cycle) 的循環垃圾回收器 (cyclic garbage collector) 來完成。垃圾回收器可以使用 `gc` 模組對其進行控制。

generator（產生器）

一個會回傳 *generator iterator*（產生器疊代器）的函式。它看起來像一個正常的函式，但不同的是它包含了 `yield` 運算式，能產生一系列的值，這些值可用於 for 迴圈，或是以 `next()` 函式，每次檢索其中的一個值。

這個術語通常用來表示一個產生器函式，但在某些情境中，也可能是表示產生器疊代器。萬一想表達的意思不夠清楚，那就使用完整的術語，以避免歧義。

generator iterator（產生器疊代器）

一個由 *generator*（產生器）函式所建立的物件。

每個 `yield` 會暫停處理程序，並記住執行狀態（包括區域變數及攔置中的 try 陳述式）。當產生器疊代器回復時，它會從停止的地方繼續執行（與那些每次調用時都要重新開始的函式有所不同）。

generator expression（產生器運算式）

一個會回傳疊代器的運算式。它看起來像一個正常的運算式，後面接著一個 `for` 子句，該子句定義了迴圈變數、範圍以及一個選擇性的 `if` 子句。該組合運算式會為外層函式產生多個值：

```
>>> sum(i*i for i in range(10))         # 平方之和 0, 1, 4, ... 81
285
```

generic function（泛型函式）

一個由多個函式組成的函式，該函式會對不同的型別實作相同的運算。呼叫期間應該使用哪種實作，是由調度演算法 (dispatch algorithm) 來決定。

另請參閱 *single dispatch*（單一調度）術語表條目、`functools.singledispatch()` 裝飾器和 **PEP 443**。

generic type（泛型型別）

一個能夠被參數化 (parameterized) 的 *type*（型別）；通常是一個 容器型別，像是 list 和 dict。它被用於型別提示和註釋。

詳情請參閱泛型別名型別、**PEP 483**、**PEP 484**、**PEP 585** 和 `typing` 模組。

GIL

請參閱 *global interpreter lock*（全域直譯器鎖）。

global interpreter lock（全域直譯器鎖）

CPython 直譯器所使用的機制，用以確保每次都只有一個執行緒能執行 Python 的 *bytecode*（位元組碼）。透過使物件模型（包括關鍵的內建型別，如 dict）自動地避免並行存取 (concurrent access) 的危險，此機制可以簡化 CPython 的實作。鎖定整個直譯器，會使直譯器更容易成為多執行緒 (multi-threaded)，但代價是會犧牲掉多處理器的機器能夠提供的一大部分平行性 (parallelism)。

然而，有些擴充模組，無論是標準的或是第三方的，它們被設計成在執行壓縮或雜湊等計算密集 (computationally intensive) 的任務時，可以解除 GIL。另外，在執行 I/O 時，GIL 總是會被解除。

從 Python 3.13 開始可以使用 `--disable-gil` 建置設定來停用 GIL。使用此選項建立 Python 後，必須使用 `-X gil=0` 來執行程式碼，或者設定 `PYTHON_GIL=0` 環境變數後再執行程式碼。此功能可以提高多執行緒應用程式的效能，並使多核心 CPU 的高效使用變得更加容易。有關更多詳細資訊，請參閱 **PEP 703**。

hash-based pyc（雜湊架構的 pyc）

一個位元組碼 (bytecode) 暫存檔，它使用雜湊值而不是對應原始檔案的最後修改時間，來確定其有效性。請參閱 pyc-invalidation。

hashable（可雜湊的）

如果一個物件有一個雜湊值，該值在其生命週期中永不改變（它需要一個 `__hash__()` method），且可與其他物件互相比較（它需要一個 `__eq__()` method），那麼它就是一個可雜湊物件。比較結果為相等的多個可雜湊物件，它們必須擁有相同的雜湊值。

可雜湊性 (hashability) 使一個物件可用作 dictionary（字典）的鍵和 set（集合）的成員，因為這些資料結構都在其內部使用了雜湊值。

大多數的 Python 不可變內建物件都是可雜湊的；可變的容器（例如 list 或 dictionary）並不是；而不可變的容器（例如 tuple（元組）和 frozenset），只有當它們的元素是可雜湊的，它們本身才是可雜湊的。若物件是使用者自定 class 的實例，則這些物件會被預設為可雜湊的。它們在互相比較時都是不相等的（除非它們與自己比較），而它們的雜湊值則是衍生自它們的 `id()`。

IDLE

Python 的 Integrated Development and Learning Environment（整合開發與學習環境）。idle 是一個基本的編輯器和直譯器環境，它和 Python 的標準發行版本一起被提供。

immortal（不滅）

不滅物件 *(Immortal objects)* 是 **PEP 683** 引入的 CPython 實作細節。

如果一個物件是不滅的，它的參照計數永遠不會被修改，因此在直譯器運行時它永遠不會被釋放。例如，`True` 和 `None` 在 CPython 中是不滅的。

immutable（不可變物件）

一個具有固定值的物件。不可變物件包括數字、字串和 tuple（元組）。這類物件是不能被改變的。如果一個不同的值必須被儲存，則必須建立一個新的物件。它們在需要固定雜湊值的地方，扮演重要的角色，例如 dictionary（字典）中的一個鍵。

import path（引入路徑）

一個位置（或路徑項目）的列表，而那些位置就是在 import 模組時，會被 *path based finder*（基於路徑的尋檢器）搜尋模組的位置。在 import 期間，此位置列表通常是來自 `sys.path`，但對於子套件 (subpackage) 而言，它也可能是來自父套件的 `__path__` 屬性。

importing（引入）

一個過程。一個模組中的 Python 程式碼可以透過此過程，被另一個模組中的 Python 程式碼使用。

importer（引入器）

一個能夠尋找及載入模組的物件；它既是 *finder*（尋檢器）也是 *loader*（載入器）物件。

interactive（互動的）

Python 有一個互動式直譯器，這表示你可以在直譯器的提示字元輸入陳述式和運算式，立即執行它們並且看到它們的結果。只要啟動 python，不需要任何引數（可能藉由從你的電腦的主選單選擇它）。這是測試新想法或檢查模塊和包的非常強大的方法（請記住 help(x)）。更多互動式模式相關資訊請見 tut-interac。

interpreted（直譯的）

Python 是一種直譯語言，而不是編譯語言，不過這個區分可能有些模糊，因為有位元組碼 (bytecode) 編譯器的存在。這表示原始檔案可以直接被運行，而不需明確地建立另一個執行檔，然後再執行它。直譯語言通常比編譯語言有更短的開發／除錯週期，不過它們的程式通常也運行得較慢。另請參閱 *interactive*（互動的）。

interpreter shutdown（直譯器關閉）

當 Python 直譯器被要求關閉時，它會進入一個特殊階段，在此它逐漸釋放所有被配置的資源，例如模組和各種關鍵內部結構。它也會多次呼叫垃圾回收器 *(garbage collector)*。這能夠觸發使用者自定的解構函式 (destructor) 或弱引用的回呼 (weakref callback)，並執行其中的程式碼。在關閉階段被

執行的程式碼會遇到各種例外，因為它所依賴的資源可能不再有作用了（常見的例子是函式庫模組或是警告機制）。

直譯器關閉的主要原因，是 `__main__` 模組或正被運行的腳本已經執行完成。

iterable（可迭代物件）

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.

可迭代物件可用於 `for` 迴圈和許多其他需要一個序列的地方 (`zip()`、`map()`...)。當一個可迭代物件作為引數被傳遞給內建函式 `iter()` 時，它會為該物件回傳一個迭代器。此迭代器適用於針對一組值進行一遍 (one pass) 運算。使用迭代器時，通常不一定要呼叫 `iter()` 或自行處理迭代器物件。`for` 陳述式會自動地為你處理這些事，它會建立一個暫時性的未命名變數，用於在迴圈期間保有該迭代器。另請參閱*iterator*（迭代器）、*sequence*（序列）和*generator*（產生器）。

iterator（迭代器）

一個表示資料流的物件。重複地呼叫迭代器的 `__next__()` method（或是將它傳遞給內建函式 `next()`）會依序回傳資料流中的各項目。當不再有資料時，則會引發 `StopIteration` 例外。此時，該迭代器物件已被用盡，而任何對其 `__next__()` method 的進一步呼叫，都只會再次引發 `StopIteration`。迭代器必須有一個 `__iter__()` method，它會回傳迭代器物件本身，所以每個迭代器也都是可迭代物件，且可以用於大多數適用其他可迭代物件的場合。一個明顯的例外，是嘗試多遍迭代 (multiple iteration passes) 的程式碼。一個容器物件（像是 `list`）在每次你將它傳遞給 `iter()` 函式或在 `for` 迴圈中使用它時，都會產生一個全新的迭代器。使用迭代器嘗試此事（多遍迭代）時，只會回傳在前一遍迭代中被用過的、同一個已被用盡的迭代器物件，使其看起來就像一個空的容器。

在 typeiter 文中可以找到更多資訊。

CPython 並不是始終如一地都會檢查「迭代器有定義 `__iter__()`」這個規定。另請注意，free-threading（自由執行緒）CPython 不保證迭代器操作的執行緒安全。

key function（鍵函式）

鍵函式或理序函式 (collation function) 是一個可呼叫 (callable) 函式，它會回傳一個用於排序 (sorting) 或定序 (ordering) 的值。例如，`locale.strxfrm()` 被用來產生一個了解區域特定排序慣例的排序鍵。

Python 中的許多工具，都接受以鍵函式來控制元素被定序或分組的方式。它們包括 `min()`、`max()`、`sorted()`、`list.sort()`、`heapq.merge()`、`heapq.nsmallest()`、`heapq.nlargest()` 和 `itertools.groupby()`。

有幾種方法可以建立一個鍵函式。例如，`str.lower()` method 可以作為不分大小寫排序的鍵函式。或者，一個鍵函式也可以從 `lambda` 運算式被建造，例如 `lambda r: (r[0], r[2])`。另外，`operator.attrgetter()`、`operator.itemgetter()` 和 `operator.methodcaller()` 是三個鍵函式式的建構函式 (constructor)。關於如何建立和使用鍵函式的範例，請參閱如何排序。

keyword argument（關鍵字引數）

請參閱*argument*（引數）。

lambda

由單一*expression*（運算式）所組成的一個匿名行內函式 (inline function)，於該函式被呼叫時求值。建立 lambda 函式的語法是 `lambda [parameters]: expression`

LBYL

Look before you leap.（三思而後行。）這種編碼風格會在進行呼叫或查找之前，明確地測試先決條件。這種風格與*EAFP* 方式形成對比，且它的特色是會有許多 if 陳述式的存在。

在一個多執行緒環境中，LBYL 方式有在「三思」和「後行」之間引入了競爭條件 (race condition) 的風險。例如以下程式碼 `if key in mapping: return mapping[key]`，如果另一個執行緒在測試之後但在查找之前，從 *mapping* 中移除了 *key*，則該程式碼就會失效。這個問題可以用鎖 (lock) 或使用 EAFP 編碼方式來解決。

list（串列）

一個 Python 內建的*sequence*（序列）。儘管它的名字是 list，它其實更類似其他語言中的一個陣列

(array) 而較不像一個鏈結串列 (linked list)，因此存取元素的時間複雜度是 $O(1)$。

list comprehension（串列綜合運算）
一種用來處理一個序列中的全部或部分元素，並將處理結果以一個 list 回傳的簡要方法。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 會產生一個字串 list，其中包含 0 到 255 範圍內，所有偶數的十六進位數 (0x..)。`if` 子句是選擇性的。如果省略它，則 `range(256)` 中的所有元素都會被處理。

loader（載入器）
一個能夠載入模組的物件。它必須定義 `exec_module()` 和 `create_module()` 方法以實作 `Loader` 介面。載入器通常是被 *finder*（尋檢器）回傳。更多細節請參考：

- finders-and-loaders

- importlib.abc.Loader

- **PEP 302**

locale encoding（區域編碼）
在 Unix 上，它是 LC_CTYPE 區域設定的編碼。它可以用 `locale.setlocale(locale.LC_CTYPE, new_locale)` 來設定。

在 Windows 上，它是 ANSI 代碼頁（code page，例如 `"cp1252"`）。

在 Android 和 VxWorks 上，Python 使用 `"utf-8"` 作為區域編碼。

`locale.getencoding()` 可以用來取得區域編碼。

也請參考 *filesystem encoding and error handler*。

magic method（魔術方法）
special method（特殊方法）的一個非正式同義詞。

mapping（對映）
一個容器物件，它支援任意鍵的查找，且能實作 abstract base classes（抽象基底類別）中，`collections.abc.Mapping` 或 `collections.abc.MutableMapping` 所指定的 method。範例包括 `dict`、`collections.defaultdict`、`collections.OrderedDict` 和 `collections.Counter`。

meta path finder（元路徑尋檢器）
一種經由搜尋 `sys.meta_path` 而回傳的 *finder*（尋檢器）。元路徑尋檢器與 *路徑項目尋檢器 (path entry finder)* 相關但是不同。

關於元路徑尋檢器實作的 method，請參考 `importlib.abc.MetaPathFinder`。

metaclass（元類別）
一種 class 的 class。Class 定義過程會建立一個 class 名稱、一個 class dictionary（字典），以及一個 base class（基底類別）的列表。Metaclass 負責接受這三個引數，並建立該 class。大多數的物件導向程式語言會提供一個預設的實作。Python 的特色之處在於它能夠建立自訂的 metaclass。大部分的使用者從未需要此工具，但是當需要時，metaclass 可以提供強大且優雅的解決方案。它們已被用於記錄屬性存取、增加執行緒安全性、追蹤物件建立、實作單例模式 (singleton)，以及許多其他的任務。

更多資訊可以在 metaclasses 章節中找到。

method（方法）
一個在 class 本體內被定義的函式。如果 method 作為其 class 實例的一個屬性被呼叫，則它將會得到該實例物件成為它的第一個 *argument*（引數）（此引數通常被稱為 `self`）。請參考 *function*（函式）和 *nested scope*（巢狀作用域）。

method resolution order（方法解析順序）
方法解析順序是在查找某個成員的過程中，base class（基底類別）被搜尋的順序。關於 Python 自 2.3 版直譯器所使用的演算法細節，請參考 python_2.3_mro。

module（模組）
一個擔任 Python 程式碼的組織單位 (organizational unit) 的物件。模組有一個命名空間，它包含任意的 Python 物件。模組是藉由 *importing* 的過程，被載入至 Python。

另請參考 *package*（套件）。

module spec（模組規格）

一個命名空間，它包含用於載入模組的 import 相關資訊。它是 `importlib.machinery.ModuleSpec` 的一個實例。

另請參⬚ module-specs。

MRO

請參⬚*method resolution order*（方法解析順序）。

mutable（可變物件）

可變物件可以改變它們的值，但維持它們的 `id()`。另請參⬚*immutable*（不可變物件）。

named tuple（附名元組）

術語「named tuple（附名元組）」是指從 tuple 繼承的任何型⬚或 class，且它的可索引 (indexable) 元素也可以用附名屬性來存取。這些型⬚或 class 也可以具有其他的特性。

有些⬚建型⬚是 named tuple，包括由 `time.localtime()` 和 `os.stat()` 回傳的值。另一個例子是 `sys.float_info`:

```
>>> sys.float_info[1]            # indexed access
1024
>>> sys.float_info.max_exp       # named field access
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

有些 named tuple 是⬚建型⬚（如上例）。或者，一個 named tuple 也可以從一個正規的 class 定義來建立，只要該 class 是繼承自 `tuple`，且定義了附名欄位 (named field) 即可。這類的 class 可以手工編寫、可以繼承自 `typing.NamedTuple` 來建立，也可以使用工廠函式 (factory function) `collections.namedtuple()` 來建立。後者技術也增加了一些額外的 method，這些 method 可能是在手寫或⬚建的 named tuple 中，無法找到的。

namespace（命名空間）

變數被儲存的地方。命名空間是以 dictionary（字典）被實作。有區域的、全域的及⬚建的命名空間，而在物件中（在 method 中）也有巢狀的命名空間。命名空間藉由防止命名衝突，來支援模組化。例如，函式 `builtins.open` 和 `os.open()` 是透過它們的命名空間來區分彼此。命名空間也藉由明確地區分是哪個模組在實作一個函式，來增進可讀性及可維護性。例如，寫出 `random.seed()` 或 `itertools.islice()` 明確地表示，這些函式分⬚是由 `random` 和 `itertools` 模組在實作。

namespace package（命名空間套件）

A *package* which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a *regular package* because they have no `__init__.py` file.

Namespace packages allow several individually installable packages to have a common parent package. Otherwise, it is recommended to use a *regular package*.

For more information, see **PEP 420** and reference-namespace-package.

另請參⬚*module*（模組）。

nested scope（巢狀作用域）

能⬚參照外層定義 (enclosing definition) 中的變數的能力。舉例來⬚，一個函式如果是在另一個函式中被定義，則它便能⬚參照外層函式中的變數。請注意，在預設情⬚下，巢狀作用域僅適用於參照，而無法用於賦值。區域變數能在最⬚層作用域中讀取及寫入。同樣地，全域變數是在全域命名空間中讀取及寫入。`nonlocal` 容許對外層作用域進行寫入。

new-style class（新式類⬚）

一個舊名，它是指現在所有的 class 物件所使用的 class 風格。在早期的 Python 版本中，只有新式 class 才能使用 Python 較新的、多樣的功能，像是 `__slots__`、描述器 (descriptor)、屬性 (property)、`__getattribute__()`、class method（類⬚方法）和 static method（⬚態方法）。

object（物件）

具有狀態（屬性或值）及被定義的行⬚（method）的任何資料。它也是任何*new-style class*（新式類⬚）的最終 base class（基底類⬚）。

optimized scope（最佳化作用域）
> A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

package（套件）
> 一個 Python 的*module*（模組），它可以包含子模組 (submodule) 或是遞⬚的子套件 (subpackage)。技術上而言，套件就是具有 `__path__` 屬性的一個 Python 模組。
>
> 另請參⬚*regular package*（正規套件）和*namespace package*（命名空間套件）。

parameter（參數）
> 在*function*（函式）或 method 定義中的一個命名實體 (named entity)，它指明該函式能⬚接受的一個*argument*（引數），或在某些情⬚下指示多個引數。共有有五種不同的參數類型：
>
> - *positional-or-keyword*（位置或關鍵字）：指明一個可以按照位置或是作⬚關鍵字引數被傳遞的引數。這是參數的預設類型，例如以下的 *foo* 和 *bar*：
>
> ```
> def func(foo, bar=None): ...
> ```
>
> - *positional-only*（僅限位置）：指明一個只能按照位置被提供的引數。在函式定義的參數列表中包含一個 / 字元，就可以在該字元前面定義僅限位置參數，例如以下的 *posonly1* 和 *posonly2*：
>
> ```
> def func(posonly1, posonly2, /, positional_or_keyword): ...
> ```
>
> - *keyword-only*（僅限關鍵字）：指明一個只能以關鍵字被提供的引數。在函式定義的參數列表中，包含一個任意數量位置參數 (var-positional parameter) 或是單純的 * 字元，就可以在其後方定義僅限關鍵字參數，例如以下的 *kw_only1* 和 *kw_only2*：
>
> ```
> def func(arg, *, kw_only1, kw_only2): ...
> ```
>
> - *var-positional*（任意數量位置）：指明一串能以任意序列被提供的位置引數（在已被其他參數接受的任何位置引數之外）。這類參數是透過在其參數名稱字首加上 * 來定義的，例如以下的 *args*：
>
> ```
> def func(*args, **kwargs): ...
> ```
>
> - *var-keyword*（任意數量關鍵字）：指明可被提供的任意數量關鍵字引數（在已被其他參數接受的任何關鍵字引數之外）。這類參數是透過在其參數名稱字首加上 ** 來定義的，例如上面範例中的 *kwargs*。
>
> 參數可以指明引數是選擇性的或必需的，也可以⬚一些選擇性的引數指定預設值。
>
> 另請參⬚術語表的*argument*（引數）條目、常見問題中的引數和參數之間的差⬚、`inspect.Parameter` class、function 章節，以及 **PEP 362**。

path entry（路徑項目）
> 在*import path*（引入路徑）中的一個位置，而*path based finder*（基於路徑的尋檢器）會參考該位置來尋找要 import 的模組。

path entry finder（路徑項目尋檢器）
> 被 `sys.path_hooks` 中的一個可呼叫物件 (callable)（意即一個*path entry hook*）所回傳的一種*finder*，它知道如何以一個*path entry*定位模組。
>
> 關於路徑項目尋檢器實作的 method，請參⬚ `importlib.abc.PathEntryFinder`。

path entry hook（路徑項目⬚）
> 在 `sys.path_hooks` 列表中的一個可呼叫物件 (callable)，若它知道如何在一個特定的*path entry* 中尋找模組，則會回傳一個*path entry finder*（路徑項目尋檢器）。

path based finder（基於路徑的尋檢器）
> 預設的*元路徑尋檢器 (meta path finder)* 之一，它會在一個*import path* 中搜尋模組。

path-like object（類路徑物件）

一個表示檔案系統路徑的物件。類路徑物件可以是一個表示路徑的 `str` 或 `bytes` 物件，或是一個實作 `os.PathLike` 協定的物件。透過呼叫 `os.fspath()` 函式，一個支援 `os.PathLike` 協定的物件可以被轉⊞⊞ `str` 或 `bytes` 檔案系統路徑；而 `os.fsdecode()` 及 `os.fsencode()` 則分⊞可用於確保 `str` 及 `bytes` 的結果。由 **PEP 519** 引入。

PEP

Python Enhancement Proposal（Python 增⊞提案）。PEP 是一份設計⊞明文件，它能⊞ Python 社群提供資訊，或是描述 Python 的一個新功能或該功能的程序和環境。PEP 應該要提供簡潔的技術規範以及被提案功能的運作原理。

PEP 的存在目的，是要成⊞重大新功能的提案、社群中關於某個問題的意見收集，以及已納入 Python 的設計⊞策的記⊞，這些過程的主要機制。PEP 的作者要負責在社群⊞建立共識⊞記⊞反對意見。

請參⊞ **PEP 1**。

portion（部分）

在單一目⊞中的一組檔案(也可能儲存在一個 zip 檔中)，這些檔案能對一個命名空間套件(namespace package) 有所貢獻，如同 **PEP 420** 中的定義。

positional argument（位置引數）

請參⊞*argument*（引數）。

provisional API（暫行 API）

暫行 API 是指，從標准函式庫的向後相容性 (backwards compatibility) 保證中，故意被排除的 API。雖然此類介面，只要它們被標示⊞暫行的，理論上⊞不會有重大的變更，但如果核心開發人員認⊞有必要，也可能會出現向後不相容的變更（甚至包括移除該介面）。這種變更⊞不會無端地⊞生——只有 API 被納入之前未察覺的嚴重基本缺陷被揭露時，它們才會發生。

即使對於暫行 API，向後不相容的變更也會被視⊞「最後的解⊞方案」——對於任何被發現的問題，仍然會盡可能找出一個向後相容的解⊞方案。

這個過程使得標准函式庫能隨著時間不斷進化，而避免耗費過長的時間去鎖定有問題的設計錯誤。請參⊞ **PEP 411** 了解更多細節。

provisional package（暫行套件）

請參⊞*provisional API*（暫行 API）。

Python 3000

Python 3.x 系列版本的⊞稱（很久以前創造的，當時第 3 版的發布是在⊞遠的未來。）也可以縮寫⊞「Py3k」。

Pythonic（Python 風格的）

一個想法或一段程式碼，它應用了 Python 語言最常見的慣用語，而不是使用其他語言常見的概念來實作程式碼。例如，Python 中常見的一種習慣用法，是使用一個 `for` 陳述式，對一個可⊞代物件的所有元素進行⊞圈。許多其他語言⊞⊞有這種類型的架構，所以不熟悉 Python 的人有時會使用一個數值計數器來代替：

```python
for i in range(len(food)):
    print(food[i])
```

相較之下，以下方法更簡潔、更具有 Python 風格：

```python
for piece in food:
    print(piece)
```

qualified name（限定名稱）

一個「點分隔名稱」，它顯示從一個模組的全域作用域到該模組中定義的 class、函式或 method 的「路徑」，如 **PEP 3155** 中的定義。對於頂層的函式和 class 而言，限定名稱與其物件名稱相同：

```python
>>> class C:
...     class D:
...         def meth(self):
```

```
...                 pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

當用於引用模組時，完全限定名懲 *(fully qualified name)* 是表示該模組的完整點分隔路徑，包括任何的父套件，例如 email.mime.text:

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

reference count（參照計數）
> 對於一個物件的參照次數。當一個物件的參照計數下降到零時，它會被解除配置 (deallocated)。有些物件是「不滅的 (immortal)」⬚擁有不會被改變的參照計數，也因此永遠不會被解除配置。參照計數通常在 Python 程式碼中看不到，但它⬚是 *CPython* 實作的一個關鍵元素。程式設計師可以呼叫 getrefcount() 函式來回傳一個特定物件的參照計數。

regular package（正規套件）
> 一個傳統的 *package*（套件），例如一個包含 __init__.py 檔案的目⬚。
>
> 另請參⬚ *namespace package*（命名空間套件）。

REPL
> 「read-eval-print ⬚圈 (read–eval–print loop)」的縮寫，是互動式直譯器 shell 的另一個名稱。

__slots__
> 在 class ⬚部的一個宣告，它藉由預先宣告實例屬性的空間，以及消除實例 dictionary（字典），來節省記憶體。雖然該技術很普遍，但它有點難以正確地使用，最好保留給那種在一個記憶體關鍵 (memory-critical) 的應用程式中存在大量實例的罕見情⬚。

sequence（序列）
> 一個 *iterable*（可⬚代物件），它透過 __getitem__() special method（特殊方法），使用整數索引來支援高效率的元素存取，⬚定義了一個 __len__() method 來回傳該序列的長度。一些⬚建序列型⬚包括 list、str、tuple 和 bytes。請注意，雖然 dict 也支援 __getitem__() 和 __len__()，但它被視⬚對映 (mapping) 而不是序列，因⬚其查找方式是使用任意的 *hashable* 鍵，而不是整數。
>
> 抽象基底類⬚ (abstract base class) collections.abc.Sequence 定義了一個更加豐富的介面，⬚不僅止於 __getitem__() 和 __len__()，還增加了 count()、index()、__contains__() 和 __reversed__()。實作此擴充介面的型⬚，可以使用 register() 被明確地⬚⬚。更多關於序列方法的文件，請見常見序列操作。

set comprehension（集合綜合運算）
> 一種緊密的方法，用來處理一個可⬚代物件中的全部或部分元素，⬚將處理結果以一個 set 回傳。results = {c for c in 'abracadabra' if c not in 'abc'} 會⬚生一個字串 set：{'r', 'd'}。請參⬚ comprehensions。

single dispatch（單一調度）
> *generic function*（泛型函式）調度的一種形式，在此，實作的選擇是基於單一引數的型⬚。

slice（切片）
> 一個物件，它通常包含一段 *sequence*（序列）的某一部分。建立一段切片的方法是使用下標符號 (subscript notation) []，若要給出多個數字，則在數字之間使用冒號，例如 variable_name[1:3:5]。在括號（下標）符號的⬚部，會使用 slice 物件。

soft deprecated（軟性⬚用）
> 被軟性⬚用的 API 代表不應再用於新程式碼中，但在現有程式碼中繼續使用它仍會是安全的。API 仍會以文件記⬚⬚會被測試，但不會被繼續改進。

與正常棄用不同，軟性棄用沒有移除 API 的規劃，也不會發出警告。

請參閱 PEP 387: 軟性棄用。

special method（特殊方法）

一種會被 Python 自動呼叫的 method，用於對某種型別執行某種運算，例如加法。這種 method 的名稱會在開頭和結尾有兩個下底線。Special method 在 specialnames 中有詳細說明。

statement（陳述式）

陳述式是一個套組（suite，一個程式碼「區塊」）中的一部分。陳述式可以是一個 *expression*（運算式），或是含有關鍵字（例如 `if`、`while` 或 `for`）的多種結構之一。

static type checker（靜態型別檢查器）

會讀取 Python 程式碼並分析的外部工具，能夠找出錯誤，像是使用了不正確的型別。另請參閱型別提示 *(type hints)* 以及 `typing` 模組。

strong reference（強參照）

在 Python 的 C API 中，強參照是對物件的參照，該物件由持有該參照的程式碼所擁有。建立參照時透過呼叫 `Py_INCREF()` 來獲得強參照、移除參照時透過 `Py_DECREF()` 釋放強參照。

`Py_NewRef()` 函式可用於建立一個對物件的強參照。通常，在退出強參照的作用域之前，必須在該強參照上呼叫 `Py_DECREF()` 函式，以避免洩漏一個參照。

另請參閱 *borrowed reference*（借用參照）。

text encoding（文字編碼）

Python 中的字串是一個 Unicode 碼點 (code point) 的序列（範圍在 `U+0000 -- U+10FFFF` 之間）。若要儲存或傳送一個字串，它必須被序列化為一個位元組序列。

將一個字串序列化為位元組序列，稱為「編碼」，而從位元組序列重新建立該字串則稱為「解碼 (decoding)」。

有多種不同的文字序列化編解碼器 (codecs)，它們被統稱為「文字編碼」。

text file（文字檔案）

一個能夠讀取和寫入 `str` 物件的一個 *file object*（檔案物件）。通常，文字檔案實際上是存取位元組導向的資料流 (byte-oriented datastream) 並會自動處理 *text encoding*（文字編碼）。文字檔案的例子有：以文字模式（`'r'` 或 `'w'`）開啟的檔案、`sys.stdin`、`sys.stdout` 以及 `io.StringIO` 的實例。

另請參閱 *binary file*（二進位檔案），它是一個能夠讀取和寫入類位元組串物件 *(bytes-like object)* 的檔案物件。

triple-quoted string（三引號式字串）

由三個雙引號 (") 或單引號 (') 的作為邊界的一個字串。雖然它們並沒有提供優於單引號字串的任何額外功能，但基於許多原因，它們仍是很有用的。它們讓你可以在字串中包含未跳脫 (unescaped) 的單引號和雙引號，而且它們不需使用連續字元 (continuation character) 就可以跨越多行，這使得它們在編寫說明字串時特別有用。

type（型別）

一個 Python 物件的型別決定了它是什麼類型的物件；每個物件都有一個型別。一個物件的型別可以用它的 `__class__` 屬性來存取，或以 `type(obj)` 來檢索。

type alias（型別別名）

一個型別的同義詞，透過將型別指定給一個識別符 (identifier) 來建立。

型別別名對於簡化型別提示 *(type hint)* 很有用。例如：

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

可以寫成這樣，更具有可讀性：

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

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

請參考 typing 和 **PEP 484**，有此功能的描述。

type hint（型別提示）

一種 *annotation*（註釋），它指定一個變數、一個 class 屬性或一個函式的參數或回傳值的預期型別。

型別提示是選擇性的，而不是被 Python 強制的，但它們對靜態型別檢查器 *(static type checkers)* 很有用，也能協助 IDE 完成程式碼的補全 (completion) 和重構 (refactoring)。

全域變數、class 屬性和函式（不含區域變數）的型別提示，都可以使用 typing.get_type_hints() 來存取。

請參考 typing 和 **PEP 484**，有此功能的描述。

universal newlines（通用換行字元）

一種解譯文字流 (text stream) 的方式，會將以下所有的情況識別為一行的結束：Unix 行尾慣例 '\n'、Windows 慣例 '\r\n' 和舊的 Macintosh 慣例 '\r'。請參考 **PEP 278** 和 **PEP 3116**，以及用於 bytes.splitlines() 的附加用途。

variable annotation（變數註釋）

一個變數或 class 屬性的 *annotation*（註釋）。

註釋變數或 class 屬性時，賦值是選擇性的：

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

變數註釋通常用於型別提示 *(type hint)*：例如，這個變數預期會取得 int（整數）值：

```
count: int = 0
```

變數註釋的語法在 annassign 章節有詳細的解釋。

請參考 *function annotation*（函式註釋）、**PEP 484** 和 **PEP 526**，皆有此功能的描述。關於註釋的最佳實踐方法，另請參考 annotations-howto。

virtual environment（虛擬環境）

一個協作隔離 (cooperatively isolated) 的執行環境，能讓 Python 的使用者和應用程式得以安裝和升級 Python 發行套件，而不會對同一個系統上運行的其他 Python 應用程式的行為產生干擾。

另請參考 venv。

virtual machine（虛擬機器）

一部完全由軟體所定義的電腦 (computer)。Python 的虛擬機器會執行由 *bytecode*（位元組碼）編譯器所發出的位元組碼。

Zen of Python（Python 之禪）

Python 設計原則與哲學的列表，其內容有助於理解和使用此語言。此列表可以透過在互動式提式字元後輸入「import this」來找到它。

關於這份F明文件

Python F明文件是透過使用 Sphinx（一個原F Python 而生的文件F生器、目前是以獨立專案形式來維護）將使用 reStructuredText 撰寫的原始檔轉F而成。

如同 Python 自身，透過自願者的努力下F出文件與封裝後自動化執行工具。若想要回報臭蟲，請見 reporting-bugs 頁面，F含相關資訊。我們永遠歡迎新的自願者加入!

致謝：

- Fred L. Drake, Jr.，原始 Python 文件工具集的創造者以及一大部份F容的作者；

- 創造 reStructuredText 和 Docutils 工具組的 Docutils 專案；

- Fredrik Lundh 先生，Sphinx 從他的 Alternative Python Reference 計劃中獲得許多的好主意。

B.1 Python 文件的貢獻者們

許多人都曾F Python 這門語言、Python 標准函式庫和 Python F明文件貢獻過。Python 所發F的原始碼中含有部份貢獻者的清單，請見 Misc/ACKS 。

正因F Python 社群的撰寫與貢獻才有這份這F棒的F明文件 -- 感謝所有貢獻過的人們!

沿革與授權

C.1 軟體沿革

Python 是由荷蘭數學和計算機科學研究學會（CWI，見 https://www.cwi.nl）的 Guido van Rossum 於 1990 年代早期所創造，目的是作為一種稱為 ABC 語言的後繼者。儘管 Python 包含了許多來自其他人的貢獻，Guido 仍是其主要作者。

1995 年，Guido 在維吉尼亞州雷斯頓的國家創新研究公司（CNRI，見 https://www.cnri.reston.va.us）繼續他在 Python 的工作，並在那裡發布了該軟體的多個版本。

2000 年五月，Guido 和 Python 核心開發團隊轉移到 BeOpen.com 並成立了 BeOpen PythonLabs 團隊。同年十月，PythonLabs 團隊轉移到 Digital Creations，後來成為 Zope Corporation。2001 年，Python 軟體基金會（PSF，見 https://www.python.org/psf/）成立，這是一個專為擁有 Python 相關的智慧財產權而創立的非營利組織。Zope Corporation 過去是 PSF 的一個贊助會員。

所有的 Python 版本都是開源的（有關開源的定義，參見 https://opensource.org）。歷史上，大多數但非全部的 Python 版本，也是 GPL 相容的；以下表格總結各個版本的差異。

發行版本	源自	年份	擁有者	GPL 相容性？(1)
0.9.0 至 1.2	不適用	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	是 (2)
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 以上	2.1.1	2001 至今	PSF	是

> ℹ 備註
>
> (1) GPL 相容並不表示我們是在 GPL 下發行 Python。不像 GPL，所有的 Python 授權都可以讓你發行修改後的版本，但不一定要使你的變更成為開源。GPL 相容的授權使得 Python 可以結合其他

> (2) 根據 Richard Stallman 的說法，1.6.1 不是 GPL 相容的，因為其授權有一個法律選擇條款。然而根據 CNRI 的說法，Stallman 的律師告訴 CNRI 的律師，1.6.1 與 GPL「不相容」。

在 GPL 下發佈的軟體一起使用；但其它的授權則不行。

感謝許多的外部志工，在 Guido 指導下的付出，使得這些版本的發佈成為可能。

C.2 關於存取或以其他方式使用 Python 的合約條款

Python 軟體和說明文件的授權是基於 Python 軟體基金會授權第二版 (Python Software Foundation License Version 2)。

從 Python 3.8.6 開始，說明文件中的範例、程式庫和其他程式碼，是被雙重授權 (dual licensed) 於 PSF 授權第二版以及 *Zero-Clause BSD 授權*。

有些被納入 Python 中的軟體是基於不同的授權。這些授權將會與其授權之程式碼一起被列出。關於這些授權的不完整清單，請參閱*被收錄軟體的授權與致謝*。

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") 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 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights
   Reserved" are retained in Python 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 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.

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

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, 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, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

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

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

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
```

```
    internet using the following unique, persistent identifier (known as a handle):
    1895.22/1013.  This Agreement may also be obtained from a proxy server on the
    internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

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

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

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

```
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 DOCUMENTA-TION

```
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 被收Ⓕ軟體的授權與致謝

本節是一個不完整但持續增加的授權與致謝清單，對象是在 Python 發Ⓕ版本中所收Ⓕ的第三方軟體。

C.3.1 Mersenne Twister

`random` 模組底下的 `_random` C 擴充程式包含了以 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 的下載Ⓕ容Ⓕ基礎的程式碼。以下是原始程式碼的完整聲明：

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

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 Sockets

socket 模組使用 `getaddrinfo()` 和 `getnameinfo()` 函式，它們在 WIDE 專案（https://www.wide.ad.jp/）中，於不同的原始檔案中被編碼：

```
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 非同步 socket 服務

`test.support.asynchat` 和 `test.support.asyncore` 模組包含以下聲明：

```
Copyright 1996 by Sam Rushing

                    All Rights Reserved

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

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
```

```
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie 管理

`http.cookies` 模組包含以下聲明：

C.3.5 執行追⬚

`trace` 模組包含以下聲明：

```
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 與 UUdecode 函式

uu 編解碼器包含以下聲明:

```
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 遠端程序呼叫

xmlrpc.client 模組包含以下聲明:

```
    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
```

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

C.3.8 test_epoll

`test.test_epoll` 模組包含以下聲明：

C.3.9 Select kqueue

`select` 模組對於 kqueue 介面包含以下聲明：

C.3.10 SipHash24

`Python/pyhash.c` 檔案包含 Marek Majkowski' 基於 Dan Bernstein 的 SipHash24 演算法的實作。它包含以下聲明：

```
<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 與 dtoa

`Python/dtoa.c` 檔案提供了 C 的 dtoa 和 strtod 函式，用於將 C 的雙精度浮點數和字串互相轉換。該檔案是衍生自 David M. Gay 建立的同名檔案，後者現在可以從 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c 下載。於 2009 年 3 月 16 日所檢索的原始檔案包含以下版權與授權聲明：

```
/****************************************************************
 *
 * 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 OpenSSL 函式庫可被作業系統使用，則 `hashlib`、`posix`、`ssl` 模組會使用它來提升效能。此外，因為 Windows 和 macOS 的 Python 安裝程式可能包含 OpenSSL 函式庫的副本，所以我們也在此收錄 OpenSSL 授權的副本。對於 OpenSSL 3.0 版本以及由此衍生的更新版本則適用 Apache 許可證 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
   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
       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.

   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

除非在建置 `pyexpat` 擴充時設定⬚ `--with-system-expat`，否則該擴充會用一個⬚含 expat 原始碼的副本來建置：

```
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

除非在建置 _ctypes 模組底下 _ctypes 擴充程式時設定 --with-system-libffi，否則該擴充會用一個含 libffi 原始碼的副本來建置：

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

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

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

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

如果在系統上找到的 zlib 版本太舊以致於無法用於建置 zlib 擴充，則該擴充會用一個含 zlib 原始碼的副本來建置：

```
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
```

C.3.16 cfuhash

tracemalloc 使用的雜⬚表 (hash table) 實作，是以 cfuhash 專案⬚基礎：

C.3.17 libmpdec

除非在建置 decimal 模組底下 _decimal C 擴充程式時設定⬚ --with-system-libmpdec，否則該模組會用一個⬚含 libmpdec 函式庫的副本來建置：

C.3.18 W3C C14N 測試套件

`test` 程式包中的 C14N 2.0 測試套件 (`Lib/test/xmltestdata/c14n-20/`) 是從 W3C 網站 https://www.w3.org/TR/xml-c14n2-testcases/ 被檢索，且是基於 3-clause BSD 授權被發⊞：

C.3.19 mimalloc

MIT 授權：

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

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.3.20 asyncio

asyncio 模組的部分 F 容是從 uvloop 0.16 中收 F 過來，其基於 MIT 授權來發 F：

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

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.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's "Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

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 unmodified, this list of conditions, and the following
```

版權宣告

Python 和這份□明文件的版權：

完整的授權條款資訊請參見沿革與授權。