# Extending and Embedding Python

*Versión 3.8.18*

**Guido van Rossum**
**and the Python development team**

**febrero 08, 2024**

**Python Software Foundation**
**Email: docs@python.org**

# Índice general

Este documento describe cómo escribir módulos en C o C++ para extender el intérprete de Python con nuevos módulos. Esos módulos no solo pueden definir nuevas funciones sino también nuevos tipos de objetos y sus métodos. El documento también describe cómo incrustar el intérprete de Python en otra aplicación, para usarlo como un lenguaje de extensión. Finalmente, muestra cómo compilar y vincular módulos de extensión para que puedan cargarse dinámicamente (en tiempo de ejecución) en el intérprete, si el sistema operativo subyacente admite esta característica.

Este documento asume conocimientos básicos sobre Python. Para una introducción informal al lenguaje, consulte tutorial-index. reference-index da una definición más formal del lenguaje. library-index documenta los tipos de objetos, funciones y módulos existentes (tanto incorporados como escritos en Python) que le dan al lenguaje su amplio rango de aplicaciones.

Para obtener una descripción detallada de toda la API de Python/C, consulte el apartado separado c-api-index.

CAPÍTULO 1

# Herramientas de terceros recomendadas

Esta guía solo cubre las herramientas básicas para crear extensiones proporcionadas como parte de esta versión de CPython. Herramientas de terceros como Cython, cffi, SWIG y Numba ofrecen enfoques más simples y sofisticados para crear extensiones C y C++ para Python.

**Ver también:**

**Guía del Usuario de Empaquetamiento de Python: Extensiones binarias** La Guía del Usuario de Empaquetamiento de Python no solo cubre varias herramientas disponibles que simplifican la creación de extensiones binarias, sino que también discute las diversas razones por las cuales crear un módulo de extensión puede ser deseable en primer lugar.

CAPÍTULO 2

# Crear extensiones sin herramientas de terceros

Esta sección de la guía cubre la creación de extensiones C y C++ sin la ayuda de herramientas de terceros. Está destinado principalmente a los creadores de esas herramientas, en lugar de ser una forma recomendada de crear sus propias extensiones C.

## 2.1 Extendiendo Python con C o C++

Es muy fácil agregar nuevos módulos incorporados a Python, si sabe cómo programar en C. Tales como *módulos de extensión* pueden hacer dos cosas que no se pueden hacer directamente en Python: pueden implementar nuevos tipos objetos incorporados, y pueden llamar a funciones de biblioteca C y llamadas de sistema.

Para admitir extensiones, la API de Python (interfaz de programadores de aplicaciones) define un conjunto de funciones, macros y variables que proporcionan acceso a la mayoría de los aspectos del sistema de tiempo de ejecución de Python. La API de Python se incorpora en un archivo fuente C incluyendo el encabezado `"Python.h"`.

La compilación de un módulo de extensión depende de su uso previsto, así como de la configuración de su sistema; los detalles se dan en capítulos posteriores.

---

**Nota:** La interfaz de extensión C es específica de CPython, y los módulos de extensión no funcionan en otras implementaciones de Python. En muchos casos, es posible evitar escribir extensiones C y preservar la portabilidad a otras implementaciones. Por ejemplo, si su caso de uso es llamar a funciones de biblioteca C o llamadas de sistema, debería considerar usar el módulo `ctypes` o la biblioteca cffi en lugar de escribir código personalizado C. Estos módulos le permiten escribir código Python para interactuar con el código C y son más portátiles entre las implementaciones de Python que escribir y compilar un módulo de extensión C.

---

### 2.1.1 Un ejemplo simple

Creemos un módulo de extensión llamado `spam` (la comida favorita de los fanáticos de Monty Python …) y digamos que queremos crear una interfaz de Python para la función de biblioteca C `system()`[1] . Esta función toma una cadena de caracteres con terminación nula como argumento y retorna un entero. Queremos que esta función se pueda llamar desde Python de la siguiente manera:

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

Comience creando un archivo `spammodule.c`. (Históricamente, si un módulo se llama `spam`, el archivo C que contiene su implementación se llama `spammodule.c`; si el nombre del módulo es muy largo, como `spammify`, el nombre del módulo puede sea solo `spammify.c`.)

Las dos primeras líneas de nuestro archivo pueden ser:

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

que extrae la API de Python (puede agregar un comentario que describa el propósito del módulo y un aviso de copyright si lo desea).

---

**Nota:** Dado que Python puede definir algunas definiciones de preprocesador que afectan los encabezados estándar en algunos sistemas, *debe* incluir `Python.h` antes de incluir encabezados estándar.

Se recomienda definir siempre `PY_SSIZE_T_CLEAN` antes de incluir `Python.h`. Consulte *Extracción de parámetros en funciones de extensión* para obtener una descripción de esta macro.

---

Todos los símbolos visibles para el usuario definidos por `Python.h` tienen un prefijo `Py` o `PY`, excepto los definidos en los archivos de encabezado estándar. Por conveniencia, y dado que el intérprete de Python los usa ampliamente, `"Python.h"` incluye algunos archivos de encabezado estándar: `<stdio.h>`, `<string.h>`, `<errno.h>`, y `<stdlib.h>`. Si el último archivo de encabezado no existe en su sistema, declara las funciones `malloc()`, `free()` y `realloc()` directamente.

Lo siguiente que agregamos a nuestro archivo de módulo es la función C que se llamará cuando se evalúe la expresión Python `spam.system(string)` (veremos en breve cómo termina siendo llamado):

```
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);
}
```

Hay una traducción directa de la lista de argumentos en Python (por ejemplo, la única expresión `"ls -l"`) a los argumentos pasados a la función C. La función C siempre tiene dos argumentos, llamados convencionalmente *self* y *args*.

El argumento *self* apunta al objeto del módulo para funciones a nivel de módulo; para un método apuntaría a la instancia del objeto.

[1] Ya existe una interfaz para esta función en el módulo estándar `os` — se eligió como un ejemplo simple y directo.

El argumento *args* será un puntero a un objeto de tupla de Python que contiene los argumentos. Cada elemento de la tupla corresponde a un argumento en la lista de argumentos de la llamada. Los argumentos son objetos de Python — para hacer algo con ellos en nuestra función C tenemos que convertirlos a valores C. La función `PyArg_ParseTuple()` en la API de Python verifica los tipos de argumento y los convierte a valores C. Utiliza una cadena de plantilla para determinar los tipos requeridos de los argumentos, así como los tipos de las variables C en las que almacenar los valores convertidos. Más sobre esto más tarde.

`PyArg_ParseTuple()` retorna verdadero (distinto de cero) si todos los argumentos tienen el tipo correcto y sus componentes se han almacenado en las variables cuyas direcciones se pasan. Retorna falso (cero) si se pasó una lista de argumentos no válidos. En el último caso, también genera una excepción apropiada para que la función de llamada pueda retornar `NULL` inmediatamente (como vimos en el ejemplo).

### 2.1.2 Intermezzo: errores y excepciones

Una convención importante en todo el intérprete de Python es la siguiente: cuando una función falla, debe establecer una condición de excepción y retornar un valor de error (generalmente un puntero `NULL`). Las excepciones se almacenan en una variable global estática dentro del intérprete; Si esta variable es `NULL`, no se ha producido ninguna excepción. Una segunda variable global almacena el «valor asociado» de la excepción (el segundo argumento para `raise`). Una tercera variable contiene el seguimiento de la pila en caso de que el error se origine en el código Python. Estas tres variables son los equivalentes en C del resultado en Python de `sys.exc_info()` (consulte la sección sobre el módulo `sys` en la Referencia de la biblioteca de Python). Es importante conocerlos para comprender cómo se transmiten los errores.

La API de Python define una serie de funciones para establecer varios tipos de excepciones.

El más común es `PyErr_SetString()`. Sus argumentos son un objeto de excepción y una cadena C. El objeto de excepción suele ser un objeto predefinido como `PyExc_ZeroDivisionError`. La cadena C indica la causa del error y se convierte en un objeto de cadena Python y se almacena como el «valor asociado» de la excepción.

Otra función útil es `PyErr_SetFromErrno()`, que solo toma un argumento de excepción y construye el valor asociado mediante la inspección de la variable global `errno`. La función más general es `PyErr_SetObject()`, que toma dos argumentos de objeto, la excepción y su valor asociado. No necesita `Py_INCREF()` los objetos pasados a cualquiera de estas funciones.

Puede probar de forma no destructiva si se ha establecido una excepción con `PyErr_Occurred()`. Esto retorna el objeto de excepción actual o `NULL` si no se ha producido ninguna excepción. Normalmente no necesita llamar a `PyErr_Occurred()` para ver si se produjo un error en una llamada a la función, ya que debería poder distinguir el valor de retorno.

Cuando una función *f* que llama a otra función *g* detecta que la última falla, *f* debería retornar un valor de error (generalmente `NULL` o `-1`). Debería *no* llamar a una de las funciones `PyErr_*()` — una ya ha sido llamada por *g*. Se supone que la persona que llama *f* también debe retornar una indicación de error a *su* persona que llama, de nuevo *sin* llamar `PyErr_*()`, y así sucesivamente — la causa más detallada del error ya fue informado por la función que lo detectó por primera vez. Una vez que el error llega al bucle principal del intérprete de Python, esto anula el código de Python que se está ejecutando actualmente e intenta encontrar un controlador de excepción especificado por el programador de Python.

(Hay situaciones en las que un módulo puede dar un mensaje de error más detallado llamando a otra función `PyErr_*()`, y en tales casos está bien hacerlo. Como regla general, sin embargo, esto es no es necesario y puede causar que se pierda información sobre la causa del error: la mayoría de las operaciones pueden fallar por varias razones.)

Para ignorar una excepción establecida por una llamada de función que falló, la condición de excepción debe borrarse explícitamente llamando a `PyErr_Clear()`. La única vez que el código C debe llamar `PyErr_Clear()` es si no quiere pasar el error al intérprete pero quiere manejarlo completamente por sí mismo (posiblemente probando algo más o pretendiendo que nada salió mal) )

Cada llamada fallida a `malloc()` debe convertirse en una excepción — la persona que llama directamente de `malloc()` (o `realloc()`) debe llamar `PyErr_NoMemory()` y retorna un indicador de falla en sí mismo. Todas las funciones de creación de objetos (por ejemplo, `PyLong_FromLong()`) ya hacen esto, por lo que esta nota solo

es relevante para aquellos que llaman `malloc()` directamente.

También tenga en cuenta que, con la importante excepción de `PyArg_ParseTuple()` y sus amigos, las funciones que retornan un estado entero generalmente retornan un valor positivo o cero para el éxito y `-1` para el fracaso, como las llamadas al sistema Unix.

Finalmente, tenga cuidado de limpiar la basura (haciendo `Py_XDECREF()` o `Py_DECREF()` requiere objetos que ya ha creado) cuando retorna un indicador de error!

La elección de qué excepción lanzar es totalmente suya. Hay objetos C declarados previamente que corresponden a todas las excepciones de Python incorporadas, como `PyExc_ZeroDivisionError`, que puede usar directamente. Por supuesto, debe elegir sabiamente las excepciones — no use `PyExc_TypeError` para significar que no se puede abrir un archivo (probablemente debería ser `PyExc_IOError`). Si algo anda mal con la lista de argumentos, la función `PyArg_ParseTuple()` generalmente genera `PyExc_TypeError`. Si tiene un argumento cuyo valor debe estar en un rango particular o debe satisfacer otras condiciones, `PyExc_ValueError` es apropiado.

También puede definir una nueva excepción que sea exclusiva de su módulo. Para esto, generalmente declara una variable de objeto estático al comienzo de su archivo:

```
static PyObject *SpamError;
```

y lo inicializa en la función de inicialización de su módulo (`PyInit_spam()`) con un objeto de excepción:

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

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

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

    return m;
}
```

Tenga en cuenta que el nombre de Python para el objeto de excepción es `spam.error`. La función `PyErr_NewException()` puede crear una clase con la clase base siendo `Exception` (a menos que se pase otra clase en lugar de `NULL`), descrita en bltin-exceptions.

Tenga en cuenta también que la variable `SpamError` retiene una referencia a la clase de excepción recién creada; esto es intencional! Como la excepción podría eliminarse del módulo mediante un código externo, se necesita una referencia propia de la clase para garantizar que no se descarte, lo que hace que `SpamError` se convierta en un puntero colgante. Si se convierte en un puntero colgante, el código C que genera la excepción podría causar un volcado del núcleo u otros efectos secundarios no deseados.

Discutimos el uso de `PyMODINIT_FUNC` como un tipo de retorno de función más adelante en esta muestra.

La excepción `spam.error` se puede generar en su módulo de extensión mediante una llamada a `PyErr_SetString()` como se muestra a continuación:

```
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 De vuelta al ejemplo

Volviendo a nuestra función de ejemplo, ahora debería poder comprender esta declaración:

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

Retorna `NULL` (el indicador de error para las funciones que retornan punteros de objeto) si se detecta un error en la lista de argumentos, basándose en la excepción establecida por `PyArg_ParseTuple()`. De lo contrario, el valor de cadena del argumento se ha copiado en la variable local `command`. Esta es una asignación de puntero y no se supone que modifique la cadena a la que apunta (por lo tanto, en el Estándar C, la variable `command` debería declararse correctamente como `const char * command`).

La siguiente declaración es una llamada a la función Unix `system()`, pasándole la cadena que acabamos de obtener de `PyArg_ParseTuple()`:

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

Nuestra función `spam.system()` debe retornar el valor de `sts` como un objeto Python. Esto se hace usando la función `PyLong_FromLong()`.

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

En este caso, retornará un objeto entero. (Sí, ¡incluso los enteros son objetos en el montículo (*heap*) en Python!)

Si tiene una función C que no retorna ningún argumento útil (una función que retorna `void`), la función Python correspondiente debe retornar `None`. Necesita este modismo para hacerlo (que se implementa mediante la macro `Py_RETURN_NONE`):

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

`Py_None` es el nombre C para el objeto especial de Python `None`. Es un objeto genuino de Python en lugar de un puntero `NULL`, que significa «error» en la mayoría de los contextos, como hemos visto.

### 2.1.4 La tabla de métodos del módulo y la función de inicialización

Prometí mostrar cómo `spam_system()` se llama desde los programas de Python. Primero, necesitamos enumerar su nombre y dirección en una «tabla de métodos»:

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

Tenga en cuenta la tercera entrada (`METH_VARARGS`). Esta es una bandera que le dice al intérprete la convención de llamada que se utilizará para la función C. Normalmente debería ser siempre `METH_VARARGS` o `METH_VARARGS | METH_KEYWORDS`; un valor de `0` significa que se usa una variante obsoleta de `PyArg_ParseTuple()`.

Cuando se usa solo `METH_VARARGS`, la función debe esperar que los parámetros a nivel de Python se pasen como una tupla aceptable para el análisis mediante `PyArg_ParseTuple()`; A continuación se proporciona más información sobre esta función.

El bit `METH_KEYWORDS` se puede establecer en el tercer campo si se deben pasar argumentos de palabras clave a la función. En este caso, la función C debería aceptar un tercer parámetro `PyObject *` que será un diccionario de palabras clave. Use `PyArg_ParseTupleAndKeywords()` para analizar los argumentos de dicha función.

La tabla de métodos debe ser referenciada en la estructura de definición del módulo:

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

Esta estructura, a su vez, debe pasarse al intérprete en la función de inicialización del módulo. La función de inicialización debe llamarse `PyInit_name()`, donde *name* es el nombre del módulo y debe ser el único elemento no `static` definido en el archivo del módulo:

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

Tenga en cuenta que PyMODINIT_FUNC declara la función como `PyObject *` tipo de retorno, declara cualquier declaración de vinculación especial requerida por la plataforma, y para C++ declara la función como `extern "C"`.

Cuando el programa Python importa el módulo `spam` por primera vez, se llama `PyInit_spam()`. (Consulte a continuación los comentarios sobre la incorporación de Python). Llama a `PyModule_Create()`, que retorna un objeto de módulo e inserta objetos de función incorporados en el módulo recién creado en función de la tabla (un arreglo de estructuras `PyMethodDef`) encontradas en la definición del módulo. `PyModule_Create()` retorna un puntero al objeto del módulo que crea. Puede abortar con un error fatal para ciertos errores, o retornar `NULL` si el módulo no se pudo inicializar satisfactoriamente. La función *init* debe retornar el objeto del módulo a su llamador, para que luego se inserte en `sys.modules`.

Al incrustar Python, la función `PyInit_spam()` no se llama automáticamente a menos que haya una entrada en la tabla `PyImport_Inittab`. Para agregar el módulo a la tabla de inicialización, use `PyImport_AppendInittab()`,

seguido opcionalmente por una importación del módulo:

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

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

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

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

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

    ...

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

**Nota:** Eliminar entradas de `sys.modules` o importar módulos compilados en múltiples intérpretes dentro de un proceso (o seguir un `fork()` sin una intervención `exec()`) puede crear problemas para algunas extensiones de módulos. Los autores de módulos de extensiones deben tener precaución al inicializar estructuras de datos internas.

Se incluye un módulo de ejemplo más sustancial en la distribución fuente de Python como `Modules/xxmodule.c`. Este archivo puede usarse como plantilla o simplemente leerse como ejemplo.

**Nota:** A diferencia de nuestro ejemplo de `spam`, `xxmodule` usa *inicialización de múltiples fases* (nuevo en Python 3.5), donde se retorna una estructura PyModuleDef de `PyInit_spam`, y la creación del módulo se deja al maquinaria de importación. Para obtener detalles sobre la inicialización múltiples fases, consulte **PEP 489**.

### 2.1.5 Compilación y Enlazamiento

Hay dos cosas más que hacer antes de que pueda usar su nueva extensión: compilarla y vincularla con el sistema Python. Si usa carga dinámica, los detalles pueden depender del estilo de carga dinámica que usa su sistema; Para obtener más información al respecto, consulte los capítulos sobre la creación de módulos de extensión (capítulo *Construyendo Extensiones C y C++*) e información adicional que se refiere solo a la construcción en Windows (capítulo *Creación de extensiones C y C++ en Windows*).

Si no puede utilizar la carga dinámica, o si desea que su módulo sea una parte permanente del intérprete de Python, tendrá que cambiar la configuración (*setup*) y reconstruir el intérprete. Afortunadamente, esto es muy simple en Unix: simplemente coloque su archivo (`spammodule.c` por ejemplo) en el directorio `Modules/ ` de una distribución fuente desempaquetada, agregue una línea al archivo :file:`Modules/Setup.local` que describe su archivo:

```
spam spammodule.o
```

y reconstruya el intérprete ejecutando **make** en el directorio de nivel superior. También puede ejecutar **make** en el subdirectorio `Modules/`, pero primero debe reconstruir `Makefile` ejecutando "**make** Makefile". (Esto es necesario cada vez que cambia el archivo `Configuración`).

Si su módulo requiere bibliotecas adicionales para vincular, también se pueden enumerar en la línea del archivo de configuración, por ejemplo:

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

### 2.1.6 Llamando funciones Python desde C

Hasta ahora nos hemos concentrado en hacer que las funciones de C puedan llamarse desde Python. Lo contrario también es útil: llamar a las funciones de Python desde C. Este es especialmente el caso de las bibliotecas que admiten las llamadas funciones de «retrollamada». Si una interfaz C utiliza retrollamadas, el Python equivalente a menudo necesita proporcionar un mecanismo de retrollamada al programador de Python; la implementación requerirá llamar a las funciones de retrollamada de Python desde una retrollamada en C. Otros usos también son imaginables.

Afortunadamente, el intérprete de Python se llama fácilmente de forma recursiva, y hay una interfaz estándar para llamar a una función de Python. (No me detendré en cómo llamar al analizador Python con una cadena particular como entrada — si está interesado, eche un vistazo a la implementación de la opción de línea de comando `-c` en `Modules/main.c` del código fuente de Python.)

Llamar a una función de Python es fácil. Primero, el programa Python debe de alguna manera pasar el objeto de función Python. Debe proporcionar una función (o alguna otra interfaz) para hacer esto. Cuando se llama a esta función, guarde un puntero en el objeto de la función Python (tenga cuidado de usar `Py_INCREF()`) En una variable global — o donde mejor le parezca. Por ejemplo, la siguiente función podría ser parte de una definición de módulo:

```
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;
```

(continué en la próxima página)

(proviene de la página anterior)

```
        }
        Py_XINCREF(temp);         /* Add a reference to new callback */
        Py_XDECREF(my_callback);  /* Dispose of previous callback */
        my_callback = temp;       /* Remember new callback */
        /* Boilerplate to return "None" */
        Py_INCREF(Py_None);
        result = Py_None;
    }
    return result;
}
```

Esta función debe registrarse con el intérprete utilizando el indicador `METH_VARARGS`; esto se describe en la sección *La tabla de métodos del módulo y la función de inicialización*. La función `PyArg_ParseTuple()` y sus argumentos están documentados en la sección *Extracción de parámetros en funciones de extensión*.

Las macros `Py_XINCREF()` y `Py_XDECREF()` incrementan/disminuyen el recuento de referencia de un objeto y son seguros en presencia de punteros `NULL` (pero tenga en cuenta que *temp* no lo hará ser `NULL` en este contexto). Más información sobre ellos en la sección *Conteo de Referencias*.

Más tarde, cuando es hora de llamar a la función, llama a la función C `PyObject_CallObject()`. Esta función tiene dos argumentos, ambos punteros a objetos arbitrarios de Python: la función Python y la lista de argumentos. La lista de argumentos siempre debe ser un objeto de tupla, cuya longitud es el número de argumentos. Para llamar a la función Python sin argumentos, pase `NULL` o una tupla vacía; para llamarlo con un argumento, pasa una tupla singleton. `Py_BuildValue()` retorna una tupla cuando su cadena de formato consta de cero o más códigos de formato entre paréntesis. Por ejemplo:

```
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()` retorna un puntero de objeto Python: este es el valor de retorno de la función Python. `PyObject_CallObject()` es «recuento-referencia-neutral» con respecto a sus argumentos. En el ejemplo, se creó una nueva tupla para servir como lista de argumentos, a la cual se le llama `Py_DECREF()` inmediatamente después de la llamada `PyObject_CallObject()`.

El valor de retorno de `PyObject_CallObject()` es «nuevo»: o bien es un objeto nuevo o es un objeto existente cuyo recuento de referencias se ha incrementado. Por lo tanto, a menos que desee guardarlo en una variable global, debería de alguna manera `Py_DECREF()` el resultado, incluso (¡especialmente!) Si no está interesado en su valor.

Sin embargo, antes de hacer esto, es importante verificar que el valor de retorno no sea `NULL`. Si es así, la función de Python terminó generando una excepción. Si el código C que llamó `PyObject_CallObject()` se llama desde Python, ahora debería retornar una indicación de error a su llamador de Python, para que el intérprete pueda imprimir un seguimiento de la pila, o el código de Python que llama puede manejar la excepción. Si esto no es posible o deseable, la excepción se debe eliminar llamando a `PyErr_Clear()`. Por ejemplo:

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

Dependiendo de la interfaz deseada para la función de retrollamada de Python, es posible que también deba proporcionar una lista de argumentos para `PyObject_CallObject()`. En algunos casos, el programa Python también proporciona la lista de argumentos, a través de la misma interfaz que especificó la función de retrollamada. Luego se puede guardar y usar de la misma manera que el objeto de función. En otros casos, puede que tenga que construir una nueva tupla para pasarla como lista de argumentos. La forma más sencilla de hacer esto es llamar a `Py_BuildValue()`. Por ejemplo, si desea pasar un código de evento integral, puede usar el siguiente código:

```
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);
```

¡Observe la ubicación de `Py_DECREF(arglist)` inmediatamente después de la llamada, antes de la verificación de errores! También tenga en cuenta que, estrictamente hablando, este código no está completo: `Py_BuildValue()` puede quedarse sin memoria, y esto debe verificarse.

También puede llamar a una función con argumentos de palabras clave utilizando `PyObject_Call()`, que admite argumentos y argumentos de palabras clave. Como en el ejemplo anterior, usamos `Py_BuildValue()` para construir el diccionario.

```
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 Extracción de parámetros en funciones de extensión

La función `PyArg_ParseTuple()` se declara de la siguiente manera:

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

El argumento *arg* debe ser un objeto de tupla que contenga una lista de argumentos pasada de Python a una función C. El argumento *format* debe ser una cadena de formato, cuya sintaxis se explica en arg-parsing en el Manual de referencia de la API de Python/C. Los argumentos restantes deben ser direcciones de variables cuyo tipo está determinado por la cadena de formato.

Tenga en cuenta que si bien `PyArg_ParseTuple()` verifica que los argumentos de Python tengan los tipos requeridos, no puede verificar la validez de las direcciones de las variables C pasadas a la llamada: si comete errores allí, su código probablemente se bloqueará o al menos sobrescribir bits aleatorios en la memoria. ¡Así que ten cuidado!

Tenga en cuenta que las referencias de objetos de Python que se proporcionan a quien llama son referencias prestadas (*borrowed*); ¡no disminuya su recuento de referencias!

Algunas llamadas de ejemplo:

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

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

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

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

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

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

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

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

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

### 2.1.8 Parámetros de palabras clave para funciones de extensión

La función `PyArg_ParseTupleAndKeywords()` se declara de la siguiente manera:

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

Los parámetros *arg* y *format* son idénticos a los de la función `PyArg_ParseTuple()`. El parámetro *kwdict* es el diccionario de palabras clave recibidas como tercer parámetro del tiempo de ejecución de Python. El parámetro *kwlist* es una lista de cadenas terminadas en `NULL` que identifican los parámetros; los nombres se corresponden con la información de tipo de *format* de izquierda a derecha. En caso de éxito, `PyArg_ParseTupleAndKeywords()` retorna verdadero; de lo contrario, retorna falso y genera una excepción apropiada.

**Nota:** ¡Las tuplas anidadas no se pueden analizar al usar argumentos de palabras clave! Los parámetros de palabras clave pasados que no están presentes en la *kwlist* provocarán que se genere `TypeError`.

Aquí hay un módulo de ejemplo que usa palabras clave, basado en un ejemplo de *Geoff Philbrick (philbrick@hks.com)*:

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

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

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

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

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

    Py_RETURN_NONE;
}

static PyMethodDef keywdarg_methods[] = {
    /* The cast of the function is necessary since PyCFunction values
     * only take two PyObject* parameters, and keywdarg_parrot() takes
     * three.
     */
    {"parrot", (PyCFunction)(void(*)(void))keywdarg_parrot, METH_VARARGS | METH_
↪KEYWORDS,
     "Print a lovely skit to standard output."},
    {NULL, NULL, 0, NULL}   /* sentinel */
};

static struct PyModuleDef keywdargmodule = {
    PyModuleDef_HEAD_INIT,
    "keywdarg",
```

(continué en la próxima página)

(proviene de la página anterior)

```
    NULL,
    -1,
    keywdarg_methods
};

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

### 2.1.9 Construyendo Valores Arbitrarios

Esta función es la contraparte de `PyArg_ParseTuple()`. Se declara de la siguiente manera:

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

Reconoce un conjunto de unidades de formato similares a las reconocidas por `PyArg_ParseTuple()`, pero los argumentos (que son de entrada a la función, no de salida) no deben ser punteros, solo valores. Retorna un nuevo objeto Python, adecuado para regresar de una función C llamada desde Python.

Una diferencia con `PyArg_ParseTuple()`: mientras que este último requiere que su primer argumento sea una tupla (ya que las listas de argumentos de Python siempre se representan como tuplas internamente), `Py_BuildValue()` no siempre construye una tupla . Construye una tupla solo si su cadena de formato contiene dos o más unidades de formato. Si la cadena de formato está vacía, retorna `None`; si contiene exactamente una unidad de formato, retorna el objeto que describa esa unidad de formato. Para forzarlo a retornar una tupla de tamaño 0 o uno, agregar paréntesis a la cadena de formato.

Ejemplos (a la izquierda la llamada, a la derecha el valor de Python resultante):

```
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 Conteo de Referencias

En lenguajes como C o C++, el programador es responsable de la asignación dinámica y la desasignación de memoria en el montón. En C, esto se hace usando las funciones `malloc()` y `free()`. En C++, los operadores `new` y `delete` se usan esencialmente con el mismo significado y restringiremos la siguiente discusión al caso C.

Cada bloque de memoria asignado con `malloc()` eventualmente debería ser retorna al grupo de memoria disponible exactamente por una llamada a `free()`. Es importante llamar a `free()` en el momento adecuado. Si se olvida la dirección de un bloque pero `free()` no se solicita, la memoria que ocupa no se puede reutilizar hasta que finalice el programa. Esto se llama *fuga de memoria*. Por otro lado, si un programa llama `free()` para un bloque y luego continúa usando el bloque, crea un conflicto con la reutilización del bloque a través de otra llamada a `malloc()`. Esto se llama *usando memoria liberada*. Tiene las mismas malas consecuencias que hacer referencia a datos no inicializados: volcados de núcleos, resultados incorrectos, bloqueos misteriosos.

Las causas comunes de pérdidas de memoria son rutas inusuales a través del código. Por ejemplo, una función puede asignar un bloque de memoria, hacer algunos cálculos y luego liberar el bloque nuevamente. Ahora, un cambio en los requisitos de la función puede agregar una prueba al cálculo que detecta una condición de error y puede regresar prematuramente de la función. Es fácil olvidar liberar el bloque de memoria asignado al tomar esta salida prematura, especialmente cuando se agrega más tarde al código. Tales filtraciones, una vez introducidas, a menudo pasan desapercibidas durante mucho tiempo: la salida del error se toma solo en una pequeña fracción de todas las llamadas, y la mayoría de las máquinas modernas tienen mucha memoria virtual, por lo que la filtración solo se hace evidente en un proceso de larga ejecución que usa la función de fugas con frecuencia. Por lo tanto, es importante evitar que se produzcan fugas mediante una convención o estrategia de codificación que minimice este tipo de errores.

Dado que Python hace un uso intensivo de `malloc()` y `free()`, necesita una estrategia para evitar pérdidas de memoria, así como el uso de memoria liberada. El método elegido se llama *recuento de referencias*. El principio es simple: cada objeto contiene un contador, que se incrementa cuando se almacena una referencia al objeto en algún lugar, y que se reduce cuando se elimina una referencia al mismo. Cuando el contador llega a cero, la última referencia al objeto se ha eliminado y el objeto se libera.

Una estrategia alternativa se llama *recolección automática de basura*. (A veces, el recuento de referencias también se conoce como una estrategia de recolección de basura, de ahí mi uso de «automático» para distinguir los dos). La gran ventaja de la recolección automática de basura es que el usuario no necesita llamar a `free()` explícitamente. (Otra ventaja afirmada es una mejora en la velocidad o el uso de la memoria; sin embargo, esto no es un hecho difícil). La desventaja es que para C, no hay un recolector de basura automático verdaderamente portátil, mientras que el conteo de referencias se puede implementar de forma portátil (siempre que las funciones `malloc()` y `free()` están disponibles — que garantiza el estándar C). Tal vez algún día un recolector de basura automático lo suficientemente portátil estará disponible para C. Hasta entonces, tendremos que vivir con recuentos de referencia.

Si bien Python utiliza la implementación tradicional de conteo de referencias, también ofrece un detector de ciclos que funciona para detectar ciclos de referencia. Esto permite que las aplicaciones no se preocupen por crear referencias circulares directas o indirectas; Estas son las debilidades de la recolección de basura implementada utilizando solo el conteo de referencias. Los ciclos de referencia consisten en objetos que contienen referencias (posiblemente indirectas) a sí mismos, de modo que cada objeto en el ciclo tiene un recuento de referencias que no es cero. Las implementaciones típicas de recuento de referencias no pueden recuperar la memoria que pertenece a algún objeto en un ciclo de referencia, o referenciada a partir de los objetos en el ciclo, a pesar de que no hay más referencias al ciclo en sí.

El detector de ciclos puede detectar ciclos de basura y puede reclamarlos. El módulo `gc` expone una forma de ejecutar el detector (la función `collect()`), así como las interfaces de configuración y la capacidad de desactivar el detector en tiempo de ejecución. El detector de ciclo se considera un componente opcional; aunque se incluye de manera predeterminada, se puede deshabilitar en el momento de la compilación utilizando la opción `--without-cycle-gc` al script **`configure`** en plataformas Unix (incluido Mac OS X). Si el detector de ciclos está deshabilitado de esta manera, el módulo `gc` no estará disponible.

### Conteo de Referencias en Python

Hay dos macros, `Py_INCREF(x)` y `Py_DECREF(x)`, que manejan el incremento y la disminución del recuento de referencias. `Py_DECREF()` también libera el objeto cuando el recuento llega a cero. Por flexibilidad, no llama a `free()` directamente — más bien, realiza una llamada a través de un puntero de función en el objeto *type object*. Para este propósito (y otros), cada objeto también contiene un puntero a su objeto de tipo.

La gran pregunta ahora permanece: ¿cuándo usar `Py_INCREF(x)` y `Py_DECREF(x)`? Primero introduzcamos algunos términos. Nadie «posee» un objeto; sin embargo, puede *poseer una referencia* a un objeto. El recuento de referencias de un objeto ahora se define como el número de referencias que posee. El propietario de una referencia es responsable de llamar a `Py_DECREF()` cuando la referencia ya no es necesaria. La propiedad de una referencia puede ser transferida. Hay tres formas de deshacerse de una referencia de propiedad: pasarla, almacenarla o llamar a `Py_DECREF()`. Olvidar deshacerse de una referencia de propiedad crea una pérdida de memoria.

También es posible *tomar prestada*[2] una referencia a un objeto. El prestatario de una referencia no debe llamar a `Py_DECREF()`. El prestatario no debe retener el objeto por más tiempo que el propietario del cual fue prestado. El uso de una referencia prestada después de que el propietario la haya eliminado corre el riesgo de usar memoria liberada y debe evitarse por completo[3].

La ventaja de pedir prestado sobre tener una referencia es que no necesita ocuparse de disponer de la referencia en todas las rutas posibles a través del código — en otras palabras, con una referencia prestada no corre el riesgo de fugas cuando se toma una salida prematura. La desventaja de pedir prestado sobre la posesión es que hay algunas situaciones sutiles en las que, en un código aparentemente correcto, una referencia prestada se puede usar después de que el propietario del que se tomó prestado la haya eliminado.

Una referencia prestada se puede cambiar en una referencia de propiedad llamando a `Py_INCREF()`. Esto no afecta el estado del propietario del cual se tomó prestada la referencia: crea una nueva referencia de propiedad y otorga responsabilidades completas al propietario (el nuevo propietario debe disponer de la referencia correctamente, así como el propietario anterior).

### Reglas de Propiedad

Cuando una referencia de objeto se pasa dentro o fuera de una función, es parte de la especificación de la interfaz de la función si la propiedad se transfiere con la referencia o no.

La mayoría de las funciones que retornan una referencia a un objeto pasan de propiedad con la referencia. En particular, todas las funciones cuya función es crear un nuevo objeto, como `PyLong_FromLong()` y `Py_BuildValue()`, pasan la propiedad al receptor. Incluso si el objeto no es realmente nuevo, aún recibirá la propiedad de una nueva referencia a ese objeto. Por ejemplo, `PyLong_FromLong()` mantiene un caché de valores populares y puede retornar una referencia a un elemento en caché.

Muchas funciones que extraen objetos de otros objetos también transfieren la propiedad con la referencia, por ejemplo `PyObject_GetAttrString()`. Sin embargo, la imagen es menos clara aquí, ya que algunas rutinas comunes son excepciones: `PyTuple_GetItem()`, `PyList_GetItem()`, `PyDict_GetItem()`, y `PyDict_GetItemString()` todas las referencias retornadas que tomaste prestadas de la tupla, lista o diccionario.

La función `PyImport_AddModule()` también retorna una referencia prestada, aunque en realidad puede crear el objeto que retorna: esto es posible porque una referencia de propiedad del objeto se almacena en `sys.modules`.

Cuando pasa una referencia de objeto a otra función, en general, la función toma prestada la referencia de usted — si necesita almacenarla, usará `Py_INCREF()` para convertirse en un propietario independiente. Hay exactamente dos excepciones importantes a esta regla: `PyTuple_SetItem()` y `PyList_SetItem()`. Estas funciones se hacen cargo de la propiedad del artículo que se les pasa, ¡incluso si fallan! (Tenga en cuenta que `PyDict_SetItem()` y sus amigos no se hacen cargo de la propiedad — son «normales»)

[2] La metáfora de «pedir prestado» una referencia no es completamente correcta: el propietario todavía tiene una copia de la referencia.

[3] ¡Comprobar que el recuento de referencia es al menos 1 **no funciona** — el recuento de referencia en sí podría estar en la memoria liberada y, por lo tanto, puede reutilizarse para otro objeto!

Cuando se llama a una función C desde Python, toma de la persona que llama referencias a sus argumentos. Quien llama posee una referencia al objeto, por lo que la vida útil de la referencia prestada está garantizada hasta que la función regrese. Solo cuando dicha referencia prestada debe almacenarse o transmitirse, debe convertirse en una referencia propia llamando a `Py_INCREF()`.

La referencia de objeto retornada desde una función C que se llama desde Python debe ser una referencia de propiedad: la propiedad se transfiere de la función a su llamador.

### Hielo delgado

Hay algunas situaciones en las que el uso aparentemente inofensivo de una referencia prestada puede generar problemas. Todo esto tiene que ver con invocaciones implícitas del intérprete, lo que puede hacer que el propietario de una referencia se deshaga de él.

El primer y más importante caso que debe conocer es el uso de `Py_DECREF()` en un objeto no relacionado mientras toma prestada una referencia a un elemento de la lista. Por ejemplo:

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

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

Esta función primero toma prestada una referencia a `list[0]`, luego reemplaza `list[1]` con el valor `0`, y finalmente imprime la referencia prestada. Parece inofensivo, ¿verdad? ¡Pero no lo es!

Sigamos el flujo de control en `PyList_SetItem()`. La lista posee referencias a todos sus elementos, por lo que cuando se reemplaza el elemento 1, debe deshacerse del elemento original 1. Ahora supongamos que el elemento original 1 era una instancia de una clase definida por el usuario, y supongamos además que la clase definió un método `__del__()`. Si esta instancia de clase tiene un recuento de referencia de 1, al eliminarla llamará a su método `__del__()`.

Como está escrito en Python, el método `__del__()` puede ejecutar código arbitrario de Python. ¿Podría hacer algo para invalidar la referencia a `item` en `error()`? ¡Tenlo por seguro! Suponiendo que la lista pasado a `bug()` es accesible para el método `__del__()`, podría ejecutar una declaración en el sentido de `del list[0]`, y suponiendo que este fuera el última referencia a ese objeto, liberaría la memoria asociada con él, invalidando así el `elemento`.

La solución, una vez que conoce el origen del problema, es fácil: incremente temporalmente el recuento de referencia. La versión correcta de la función dice:

```
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);
}
```

Esta es una historia real. Una versión anterior de Python contenía variantes de este error y alguien pasó una cantidad considerable de tiempo en un depurador C para descubrir por qué sus métodos `__del__()` fallaban …

El segundo caso de problemas con una referencia prestada es una variante que involucra hilos. Normalmente, varios hilos en el intérprete de Python no pueden interponerse entre sí, porque hay un bloqueo global que protege to-

do el espacio de objetos de Python. Sin embargo, es posible liberar temporalmente este bloqueo usando la macro `Py_BEGIN_ALLOW_THREADS`, y volver a adquirirlo usando `Py_END_ALLOW_THREADS`. Esto es común al bloquear las llamadas de E/S, para permitir que otros subprocesos usen el procesador mientras esperan que se complete la E/S. Obviamente, la siguiente función tiene el mismo problema que la anterior:

```
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! */
}
```

### Punteros NULL

En general, las funciones que toman referencias de objetos como argumentos no esperan que les pase los punteros `NULL`, y volcará el núcleo (o causará volcados de núcleo posteriores) si lo hace. Las funciones que retornan referencias a objetos generalmente retornan `NULL` solo para indicar que ocurrió una excepción. La razón para no probar los argumentos `NULL` es que las funciones a menudo pasan los objetos que reciben a otra función — si cada función probara `NULL`, habría muchas pruebas redundantes y el código correría más lentamente.

Es mejor probar `NULL` solo en «source:» cuando se recibe un puntero que puede ser `NULL`, por ejemplo, de `malloc()` o de una función que puede plantear una excepción.

Las macros `Py_INCREF()` y `Py_DECREF()` no comprueban los punteros `NULL` — sin embargo, sus variantes `Py_XINCREF()` y `Py_XDECREF()` lo hacen.

Las macros para verificar un tipo de objeto en particular (`Pytype_Check()`) no verifican los punteros `NULL` — nuevamente, hay mucho código que llama a varios de estos en una fila para probar un objeto contra varios tipos esperados diferentes, y esto generaría pruebas redundantes. No hay variantes con comprobación `NULL`.

El mecanismo de llamada a la función C garantiza que la lista de argumentos pasada a las funciones C (`args` en los ejemplos) nunca sea `NULL` — de hecho, garantiza que siempre sea una tupla[4].

Es un error grave dejar que un puntero `NULL` «escape» al usuario de Python.

## 2.1.11 Escribiendo Extensiones en C++

Es posible escribir módulos de extensión en C++. Se aplican algunas restricciones. Si el compilador de C compila y vincula el programa principal (el intérprete de Python), no se pueden usar objetos globales o estáticos con constructores. Esto no es un problema si el programa principal está vinculado por el compilador de C++. Las funciones que serán llamadas por el intérprete de Python (en particular, las funciones de inicialización del módulo) deben declararse usando `extern "C"`. No es necesario encerrar los archivos de encabezado de Python en `extern "C" {...}` — ya usan este formulario si el símbolo `__cplusplus` está definido (todos los compiladores recientes de C++ definen este símbolo) .

[4] Estas garantías no se cumplen cuando utiliza la convención de llamadas de estilo «antiguo», que todavía se encuentra en muchos códigos existentes.

### 2.1.12 Proporcionar una API C para un módulo de extensión

Muchos módulos de extensión solo proporcionan nuevas funciones y tipos para ser utilizados desde Python, pero a veces el código en un módulo de extensión puede ser útil para otros módulos de extensión. Por ejemplo, un módulo de extensión podría implementar un tipo de «colección» que funciona como listas sin orden. Al igual que el tipo de lista Python estándar tiene una API C que permite a los módulos de extensión crear y manipular listas, este nuevo tipo de colección debe tener un conjunto de funciones C para la manipulación directa desde otros módulos de extensión.

A primera vista, esto parece fácil: simplemente escriba las funciones (sin declararlas `static`, por supuesto), proporcione un archivo de encabezado apropiado y documente la API de C. Y, de hecho, esto funcionaría si todos los módulos de extensión siempre estuvieran vinculados estáticamente con el intérprete de Python. Sin embargo, cuando los módulos se usan como bibliotecas compartidas, los símbolos definidos en un módulo pueden no ser visibles para otro módulo. Los detalles de visibilidad dependen del sistema operativo; algunos sistemas usan un espacio de nombres global para el intérprete de Python y todos los módulos de extensión (Windows, por ejemplo), mientras que otros requieren una lista explícita de símbolos importados en el momento del enlace del módulo (AIX es un ejemplo) u ofrecen una variedad de estrategias diferentes (la mayoría Unices). E incluso si los símbolos son visibles a nivel mundial, ¡el módulo cuyas funciones uno desea llamar podría no haberse cargado todavía!

Por lo tanto, la portabilidad requiere no hacer suposiciones sobre la visibilidad del símbolo. Esto significa que todos los símbolos en los módulos de extensión deben declararse `static`, excepto la función de inicialización del módulo, para evitar conflictos de nombres con otros módulos de extensión (como se discutió en la sección *La tabla de métodos del módulo y la función de inicialización*). Y significa que los símbolos que *deberían* ser accesibles desde otros módulos de extensión deben exportarse de una manera diferente.

Python proporciona un mecanismo especial para pasar información de nivel C (punteros) de un módulo de extensión a otro: Cápsulas. Una cápsula es un tipo de datos de Python que almacena un puntero (`void *`). Las cápsulas solo se pueden crear y acceder a través de su API de C, pero se pueden pasar como cualquier otro objeto de Python. En particular, pueden asignarse a un nombre en el espacio de nombres de un módulo de extensión. Otros módulos de extensión pueden importar este módulo, recuperar el valor de este nombre y luego recuperar el puntero de la Cápsula.

Hay muchas formas en que las Cápsulas se pueden usar para exportar la API de C de un módulo de extensión. Cada función podría tener su propia cápsula, o todos los punteros de API C podrían almacenarse en una matriz cuya dirección se publica en una cápsula. Y las diversas tareas de almacenamiento y recuperación de los punteros se pueden distribuir de diferentes maneras entre el módulo que proporciona el código y los módulos del cliente.

Sea cual sea el método que elija, es importante nombrar sus cápsulas correctamente. La función `PyCapsule_New()` toma un parámetro de nombre (`const char *`); se le permite pasar un nombre `NULL`, pero le recomendamos que especifique un nombre. Las cápsulas correctamente nombradas proporcionan un grado de seguridad de tipo de tiempo de ejecución; no hay una manera factible de distinguir una Cápsula sin nombre de otra.

En particular, las cápsulas utilizadas para exponer las API de C deben recibir un nombre siguiendo esta convención:

```
modulename.attributename
```

La función de conveniencia `PyCapsule_Import()` facilita la carga de una API C proporcionada a través de una cápsula, pero solo si el nombre de la cápsula coincide con esta convención. Este comportamiento brinda a los usuarios de C API un alto grado de certeza de que la Cápsula que cargan contiene la API de C correcta.

El siguiente ejemplo demuestra un enfoque que pone la mayor parte de la carga en el escritor del módulo de exportación, que es apropiado para los módulos de biblioteca de uso común. Almacena todos los punteros de API C (¡solo uno en el ejemplo!) En un arreglo de punteros `void` que se convierte en el valor de una cápsula. El archivo de encabezado correspondiente al módulo proporciona una macro que se encarga de importar el módulo y recuperar sus punteros de API C; Los módulos de cliente solo tienen que llamar a esta macro antes de acceder a la API de C.

El módulo de exportación es una modificación del módulo `spam` de la sección *Un ejemplo simple*. La función `spam.system()` no llama a la función de la biblioteca C `system()` directamente, sino una función `PySpam_System()`, que por supuesto haría algo más complicado en realidad (como agregar «spam» a cada comando). Esta función `PySpam_System()` también se exporta a otros módulos de extensión.

La función `PySpam_System()` es una función C simple, declarada `static` como todo lo demás:

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

La función `spam_system()` se modifica de manera trivial:

```
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);
}
```

Al comienzo del módulo, justo después de la línea:

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

se deben agregar dos líneas más:

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

El `#define` se usa para decirle al archivo de encabezado que se está incluyendo en el módulo de exportación, no en un módulo de cliente. Finalmente, la función de inicialización del módulo debe encargarse de inicializar la matriz de punteros de API 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_AddObject(m, "_C_API", c_api_object) < 0) {
        Py_XDECREF(c_api_object);
        Py_DECREF(m);
        return NULL;
    }
```

(continué en la próxima página)

(proviene de la página anterior)

```
    return m;
}
```

Tenga en cuenta que `PySpam_API` se declara `static`; de lo contrario, la matriz de punteros desaparecería cuando `PyInit_spam()` finalice!

La mayor parte del trabajo está en el archivo de encabezado `spammodule.h`, que se ve así:

```
#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) */
```

Todo lo que un módulo cliente debe hacer para tener acceso a la función `PySpam_System()` es llamar a la función (o

más bien macro) `import_spam()` en su función de inicialización:

```
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;
}
```

La principal desventaja de este enfoque es que el archivo `spammodule.h` es bastante complicado. Sin embargo, la estructura básica es la misma para cada función que se exporta, por lo que solo se debe aprender una vez.

Finalmente, debe mencionarse que las cápsulas ofrecen una funcionalidad adicional, que es especialmente útil para la asignación de memoria y la desasignación del puntero almacenado en una cápsula. Los detalles se describen en el Manual de referencia de Python/C API en la sección capsules y en la implementación de Capsules (archivos `Include/pycapsule.h` y `Objects/pycapsule.c` en la distribución del código fuente de Python).

### Notas al pie de página

## 2.2 Definición de tipos de extensión: Tutorial

Python le permite al escritor de un módulo de extensión C definir nuevos tipos que pueden ser manipulados desde el código Python, al igual que los tipos incorporados `str` y `list`. El código para todos los tipos de extensión sigue un patrón, pero hay algunos detalles que debe comprender antes de comenzar. Este documento es una introducción suave al tema.

### 2.2.1 Lo Básico

El tiempo de ejecución *CPython* ve todos los objetos de Python como variables de tipo `PyObject*`, que sirve como un «tipo base» para todos los objetos de Python. La estructura `PyObject` solo contiene el *reference count* del objeto y un puntero al «objeto tipo» del objeto. Aquí es donde está la acción; el objeto tipo determina qué funciones (C) llama el intérprete cuando, por ejemplo, se busca un atributo en un objeto, se llama un método o se multiplica por otro objeto. Estas funciones de C se denominan «métodos de tipo».

Por lo tanto, si desea definir un nuevo tipo de extensión, debe crear un nuevo objeto de tipo.

Este tipo de cosas solo se pueden explicar con un ejemplo, por lo que aquí hay un módulo mínimo, pero completo, que define un nuevo tipo llamado `Custom` dentro de un módulo de extensión C `custom`:

---

**Nota:** Lo que estamos mostrando aquí es la forma tradicional de definir tipos de extensión *estáticos*. Debe ser adecuado para la mayoría de los usos. La API de C también permite definir tipos de extensiones asignadas en el montón utilizando la función `PyType_FromSpec()`, que no se trata en este tutorial.

---

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

(continué en la próxima página)

(proviene de la página anterior)

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

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

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

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

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

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

    return m;
}
```

Ahora, eso es bastante para asimilar a la vez, pero espero que los fragmentos le resulten familiares del capítulo anterior. Este archivo define tres cosas:

1. Lo que contiene un **objeto** `Custom`: esta es la estructura `CustomObject`, que se asigna una vez para cada instancia de `Custom`.
2. Cómo se comporta `Custom` **type**: esta es la estructura `CustomType`, que define un conjunto de indicadores y punteros de función que el intérprete inspecciona cuando se solicitan operaciones específicas.
3. Cómo inicializar el módulo `custom`: esta es la función `PyInit_custom` y la estructura asociada `custommodule`.

La primera parte es:

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

Esto es lo que contendrá un objeto personalizado. `PyObject_HEAD` es obligatorio al comienzo de cada estructura de objeto y define un campo llamado `ob_base` de tipo `PyObject`, que contiene un puntero a un objeto de tipo y un recuento de referencia (estos pueden ser accedidos mediante las macros `Py_REFCNT` y `Py_TYPE` respectivamente). El motivo de la macro es abstraer el diseño y habilitar campos adicionales en las compilaciones de depuración.

---

**Nota:** No hay punto y coma (;) arriba después de la macro `PyObject_HEAD`. Tenga cuidado de agregar uno por accidente: algunos compiladores se quejarán.

---

Por supuesto, los objetos generalmente almacenan datos adicionales además del estándar `PyObject_HEAD` repetitivo; por ejemplo, aquí está la definición de puntos flotantes del estándar de Python:

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

La segunda parte es la definición del tipo de objecto.

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

---

**Nota:** Recomendamos utilizar los inicializadores designados al estilo C99 como se indica arriba, para evitar enumerar todos los campos `PyTypeObject` que no le interesan y también para evitar preocuparse por el orden de declaración de los campos.

---

La definición real de `PyTypeObject` en `object.h` tiene muchos más campos que la definición anterior. El compilador de C rellenará los campos restantes con ceros, y es una práctica común no especificarlos explícitamente a menos que los necesite.

Lo vamos a separar, un campo a la vez:

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

Esta línea es obligatoria para inicializar el campo `ob_base` mencionado anteriormente.

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

El nombre de nuestro tipo. Esto aparecerá en la representación textual predeterminada de nuestros objetos y en algunos mensajes de error, por ejemplo:

```
>>> "" + custom.Custom()
Traceback (most recent call last):
```

(continué en la próxima página)

(proviene de la página anterior)

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

Tenga en cuenta que el nombre es un nombre punteado que incluye tanto el nombre del módulo como el nombre del tipo dentro del módulo. El módulo en este caso es `custom` y el tipo es `Custom`, por lo que establecemos el nombre del tipo en `custom.Custom`. Usar la ruta de importación punteada real es importante para que su tipo sea compatible con los módulos `pydoc` y `pickle`.

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

Esto es para que Python sepa cuánta memoria asignar al crear instancias nuevas `Custom`. `tp_itemsize` solo se usa para objetos de tamaño variable y, de lo contrario, debería ser cero.

---

**Nota:** Si desea que su tipo pueda tener subclases desde Python, y su tipo tiene el mismo `tp_basicsize` como su tipo base, puede tener problemas con la herencia múltiple. Una subclase de Python de su tipo tendrá que enumerar su tipo primero en su `__bases__`, o de lo contrario no podrá llamar al método de su tipo `__new__()` sin obtener un error. Puede evitar este problema asegurándose de que su tipo tenga un valor mayor para `tp_basicsize` que su tipo base. La mayoría de las veces, esto será cierto de todos modos, porque su tipo base será `object`, o de lo contrario agregará miembros de datos a su tipo base y, por lo tanto, aumentará su tamaño.

---

Configuramos las banderas de clase a `Py_TPFLAGS_DEFAULT`.

```
.tp_flags = Py_TPFLAGS_DEFAULT,
```

Todos los tipos deben incluir esta constante en sus banderas. Habilita todos los miembros definidos hasta al menos Python 3.3. Si necesita más miembros, necesitará O (*OR*) las banderas correspondientes.

Proporcionamos una cadena de documentos para el tipo en `tp_doc`.

```
.tp_doc = "Custom objects",
```

Para habilitar la creación de objetos, debemos proporcionar un controlador `tp_new`. Este es el equivalente del método Python `__new__()`, pero debe especificarse explícitamente. En este caso, podemos usar la implementación predeterminada proporcionada por la función API `PyType_GenericNew()`.

```
.tp_new = PyType_GenericNew,
```

Todo lo demás en el archivo debe ser familiar, excepto algún código en `PyInit_custom()`:

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

Esto inicializa el tipo `Custom`, completando un número de miembros con los valores predeterminados apropiados, que incluyen `ob_type` que inicialmente configuramos en `NULL`.

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

Esto agrega el tipo al diccionario del módulo. Esto nos permite crear instancias `Custom` llamando la clase `Custom`:

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

¡Eso es! Todo lo que queda es construirlo; ponga el código anterior en un archivo llamado `custom.c` y:

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

en un archivo llamado `setup.py`; luego escribiendo

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

en un shell debería producir un archivo `custom.so` en un subdirectorio; muévete a ese directorio y abre Python — deberías poder `import custom` y jugar con objetos personalizados.

Eso no fue tan difícil, ¿verdad?

Por supuesto, el tipo personalizado actual es bastante poco interesante. No tiene datos y no hace nada. Ni siquiera se puede subclasificar.

---

**Nota:** Si bien esta documentación muestra el módulo estándar `distutils` para construir extensiones C, se recomienda en casos de uso del mundo real utilizar la biblioteca `setuptools` más nueva y mejor mantenida. La documentación sobre cómo hacer esto está fuera del alcance de este documento y se puede encontrar en la Guía de usuario del Empaquetamiento de Python.

---

### 2.2.2 Agregar datos y métodos al ejemplo básico

Extendamos el ejemplo básico para agregar algunos datos y métodos. También hagamos que el tipo sea utilizable como una clase base. Crearemos un nuevo módulo, `custom2` que agrega estas capacidades:

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

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

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

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
```

(continué en la próxima página)

(proviene de la página anterior)

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

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

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

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }
    return 0;
}

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

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    if (self->first == NULL) {
```

(continué en la próxima página)

(proviene de la página anterior)

```
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

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

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

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

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

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

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

    return m;
```

(continué en la próxima página)

(proviene de la página anterior)

```
}
```

Esta versión del módulo tiene una serie de cambios.

Hemos agregado una inclusión adicional:

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

Esto incluye declaraciones que usamos para manejar atributos, como se describe un poco más adelante.

El tipo `Custom` ahora tiene tres atributos de datos en su estructura C, *first*, *last* y *number*. Las variables *first* y *last* son cadenas de caracteres de Python que contienen nombres y apellidos. El atributo *number* es un entero C.

La estructura del objeto se actualiza en consecuencia:

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

Debido a que ahora tenemos datos para administrar, debemos ser más cuidadosos con la asignación de objetos y la desasignación. Como mínimo, necesitamos un método de desasignación:

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

que se asigna al miembro `tp_dealloc`:

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

Este método primero borra los recuentos de referencia de los dos atributos de Python. `Py_XDECREF()` maneja correctamente el caso donde su argumento es `NULL` (lo que podría ocurrir aquí si `tp_new` fallara a mitad de camino). Luego llama al miembro `tp_free` del tipo de objeto (calculado por `Py_TYPE(self)`) para liberar la memoria del objeto. Tenga en cuenta que el tipo de objeto podría no ser `CustomType`, porque el objeto puede ser una instancia de una subclase.

---

**Nota:** La conversión explícita a `destructor` anterior es necesaria porque definimos `Custom_dealloc` para tomar un argumento `CustomObject *`, pero el puntero de función `tp_dealloc` espera recibir un argumento `PyObject *`. De lo contrario, el compilador emitirá una advertencia. Este es un polimorfismo orientado a objetos, en C!

---

Queremos asegurarnos de que el nombre y el apellido se inicialicen en cadenas de caracteres vacías, por lo que proporcionamos una implementación `tp_new`:

```
static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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;
}
```

e instalarlo en el miembro `tp_new`:

```
.tp_new = Custom_new,
```

El controlador `tp_new` es responsable de crear (en lugar de inicializar) objetos del tipo. Está expuesto en Python como el método `__new__()`. No es necesario definir un miembro `tp_new`, y de hecho muchos tipos de extensiones simplemente reutilizarán `PyType_GenericNew()` como se hizo en la primera versión del tipo `Personalizado` anterior. En este caso, usamos el controlador `tp_new` para inicializar los atributos `first` y `last` a valores predeterminados que no sean `NULL`.

`tp_new` se pasa el tipo que se instancia (no necesariamente `CustomType`, si se instancia una subclase) y cualquier argumento pasado cuando se llamó al tipo, y se espera que retorna la instancia creada. Los manejadores `tp_new` siempre aceptan argumentos posicionales y de palabras clave, pero a menudo ignoran los argumentos, dejando el manejo de argumentos al inicializador (también conocido como, `tp_init` en C o `__init__` en Python).

---

**Nota:** `tp_new` no debería llamar explícitamente a `tp_init`, ya que el intérprete lo hará por sí mismo.

---

La implementación `tp_new` llama al `tp_alloc` para asignar memoria:

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

Como la asignación de memoria puede fallar, debemos verificar el resultado `tp_alloc` contra `NULL` antes de continuar.

---

**Nota:** No llenamos la ranura `tp_alloc` nosotros mismos. Más bien `PyType_Ready()` lo llena para nosotros al heredarlo de nuestra clase base, que es `object` por defecto. La mayoría de los tipos utilizan la estrategia de asignación predeterminada.

---

---

**Nota:** Si está creando una cooperativa `tp_new` (una que llama a un tipo base `tp_new` o `__new__()`), *no* debe intentar determinar a qué método llamar utilizando el orden de resolución del método en tiempo de ejecución. Siempre determine estáticamente a qué tipo va a llamar, y llame a su `tp_new` directamente, o mediante `type->tp_base->tp_new`. Si no hace esto, las subclases de Python de su tipo que también heredan de otras clases definidas por Python pueden no funcionar correctamente. (Específicamente, es posible que no pueda crear instancias de tales subclases sin obtener un `TypeError`).

---

También definimos una función de inicialización que acepta argumentos para proporcionar valores iniciales para nuestra instancia:

```
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;
}
```

rellenando la ranura `tp_init`.

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

La ranura `tp_init` está expuesta en Python como el método `__init__()`. Se utiliza para inicializar un objeto una vez creado. Los inicializadores siempre aceptan argumentos posicionales y de palabras clave, y deben retornar `0` en caso de éxito o `-1` en caso de error.

A diferencia del controlador `tp_new`, no hay garantía de que se llame a `tp_init` (por ejemplo, el módulo `pickle` por defecto no llama a `__init__()` en instancias no controladas ) También se puede llamar varias veces. Cualquiera puede llamar al método `__init__()` en nuestros objetos. Por esta razón, debemos tener mucho cuidado al asignar los nuevos valores de atributo. Podríamos sentirnos tentados, por ejemplo, a asignar el `primer` miembro de esta manera:

```
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.

Para ser paranoicos y protegernos de esta posibilidad, casi siempre reasignamos miembros antes de disminuir sus recuentos de referencias. ¿Cuándo no tenemos que hacer esto?

- cuando sabemos absolutamente que el recuento de referencia es mayor que 1;
- cuando sabemos que la desasignación del objeto[1] no liberará el *GIL* ni causará ninguna llamada al código de nuestro tipo;

[1] Esto es cierto cuando sabemos que el objeto es un tipo básico, como una cadena o un flotador.

- al disminuir un recuento de referencias en un manejador `tp_dealloc` en un tipo que no admite la recolección de basura cíclica[2].

Queremos exponer nuestras variables de instancia como atributos. Hay varias formas de hacerlo. La forma más simple es definir definiciones de miembros:

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

y poner las definiciones en la ranura `tp_members`:

```
.tp_members = Custom_members,
```

Cada definición de miembro tiene un nombre de miembro, tipo, desplazamiento, banderas de acceso y cadena de caracteres de documentación. Consulte la sección *Gestión de atributos genéricos* a continuación para obtener más detalles.

Una desventaja de este enfoque es que no proporciona una forma de restringir los tipos de objetos que se pueden asignar a los atributos de Python. Esperamos que el nombre y el apellido sean cadenas, pero se pueden asignar objetos de Python. Además, los atributos se pueden eliminar, configurando los punteros C en `NULL`. Si bien podemos asegurarnos de que los miembros se inicialicen en valores que no sean `NULL`, los miembros se pueden establecer en `NULL` si se eliminan los atributos.

Definimos un método único, `Custom.name()`, que genera el nombre de los objetos como la concatenación de los nombres y apellidos.

```
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);
}
```

El método se implementa como una función C que toma una instancia de `Custom` (o subclase `Custom`) como primer argumento. Los métodos siempre toman una instancia como primer argumento. Los métodos a menudo también toman argumentos posicionales y de palabras clave, pero en este caso no tomamos ninguno y no necesitamos aceptar una tupla de argumentos posicionales o un diccionario de argumentos de palabras clave. Este método es equivalente al método Python:

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

Tenga en cuenta que tenemos que verificar la posibilidad de que nuestros miembros `first` y `last` sean `NULL`. Esto se debe a que se pueden eliminar, en cuyo caso se establecen en `NULL`. Sería mejor evitar la eliminación de estos atributos y restringir los valores de los atributos para que sean cadenas de caracteres. Veremos cómo hacerlo en la siguiente sección.

[2] Nos basamos en esto en el manejador `tp_dealloc` en este ejemplo, porque nuestro tipo no admite la recolección de basura.

Ahora que hemos definido el método, necesitamos crear un arreglo de definiciones de métodos:

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

(tenga en cuenta que usamos el indicador `METH_NOARGS` para indicar que el método no espera argumentos distintos de *self*)

y asignarlo a la ranura `tp_methods`:

```
.tp_methods = Custom_methods,
```

Finalmente, haremos que nuestro tipo sea utilizable como una clase base para la subclase. Hemos escrito nuestros métodos cuidadosamente hasta ahora para que no hagan suposiciones sobre el tipo de objeto que se está creando o utilizando, por lo que todo lo que tenemos que hacer es agregar `Py_TPFLAGS_BASETYPE` a nuestra definición de bandera de clase:

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

Cambiamos el nombre de `PyInit_custom()` a `PyInit_custom2()`, actualizamos el nombre del módulo en la estructura `PyModuleDef` y actualizamos el nombre completo de la clase en la estructura `PyTypeObject`.

Finalmente, actualizamos nuestro archivo `setup.py` para construir el nuevo módulo:

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

### 2.2.3 Proporcionar un control más preciso sobre los atributos de datos

En esta sección, proporcionaremos un control más preciso sobre cómo se establecen los atributos `first` y `last` en el ejemplo `Custom`. En la versión anterior de nuestro módulo, las variables de instancia `first` y `last` podrían establecerse en valores que no sean de cadena o incluso eliminarse. Queremos asegurarnos de que estos atributos siempre contengan cadenas.

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

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

static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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, *tmp;

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

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}

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

(continué en la próxima página)

(proviene de la página anterior)

```
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;
}

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

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

static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
```

(continué en la próxima página)

(proviene de la página anterior)

```
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL}  /* Sentinel */
};

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

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

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

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

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

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

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

(continué en la próxima página)

(proviene de la página anterior)

```
    return m;
}
```

Para proporcionar un mayor control sobre los atributos `first` y `last`, usaremos funciones personalizadas *getter* y *setter*. Estas son las funciones para obtener y configurar el atributo `first`:

```
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;
}
```

La función *getter* se pasa al objeto `Custom` y un «cierre» (*closure*), que es un puntero nulo. En este caso, se ignora el cierre. (El cierre admite un uso avanzado en el que los datos de definición se pasan al captador y al definidor. Esto podría, por ejemplo, usarse para permitir un solo conjunto de funciones de captador y definidor que deciden que el atributo se obtenga o establezca en función de los datos en el cierre.)

La función *setter* pasa el objeto `Custom`, el nuevo valor y el cierre. El nuevo valor puede ser `NULL`, en cuyo caso se está eliminando el atributo. En nuestro *setter*, generamos un error si el atributo se elimina o si su nuevo valor no es una cadena.

Creamos un arreglo de estructuras `PyGetSetDef`:

```
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 */
};
```

y lo registra en la ranura `tp_getset`:

```
.tp_getset = Custom_getsetters,
```

El último elemento en la estructura `PyGetSetDef` es el «cierre» (*closure*) mencionado anteriormente. En este caso, no estamos usando un cierre, por lo que simplemente pasamos `NULL`.

También eliminamos las definiciones de miembro para estos atributos:

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

También necesitamos actualizar el manejador `tp_init` para permitir que solo se pasen las cadenas[3]:

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

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

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}
```

Con estos cambios, podemos asegurar que los miembros `primero` y `último` nunca sean `NULL`, por lo que podemos eliminar las comprobaciones de los valores `NULL` en casi todos los casos. Esto significa que la mayoría de las llamadas `Py_XDECREF()` se pueden convertir en llamadas `Py_DECREF()`. El único lugar donde no podemos cambiar estas llamadas es en la implementación `tp_dealloc`, donde existe la posibilidad de que la inicialización de estos miembros falle en `tp_new`.

También cambiamos el nombre de la función de inicialización del módulo y el nombre del módulo en la función de inicialización, como lo hicimos antes, y agregamos una definición adicional al archivo `setup.py`.

[3] Ahora sabemos que el primer y el último miembro son cadenas de caracteres, por lo que quizás podríamos ser menos cuidadosos al disminuir sus recuentos de referencia, sin embargo, aceptamos instancias de subclases de cadenas. A pesar de que la desasignación de cadenas normales no volverá a llamar a nuestros objetos, no podemos garantizar que la desasignación de una instancia de una subclase de cadena de caracteres no vuelva a llamar a nuestros objetos.

### 2.2.4 Apoyo a la recolección de basura cíclica

Python tiene un *recolector de basura cíclico (GC)* que puede identificar objetos innecesarios incluso cuando sus recuentos de referencia no son cero. Esto puede suceder cuando los objetos están involucrados en ciclos. Por ejemplo, considere:

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

En este ejemplo, creamos una lista que se contiene a sí misma. Cuando lo eliminamos, todavía tiene una referencia de sí mismo. Su recuento de referencia no cae a cero. Afortunadamente, el recolector cíclico de basura de Python finalmente descubrirá que la lista es basura y la liberará.

En la segunda versión del ejemplo `Custom`, permitimos que cualquier tipo de objeto se almacene en `first` o `last` atributos[4]. Además, en la segunda y tercera versión, permitimos subclases `Custom`, y las subclases pueden agregar atributos arbitrarios. Por cualquiera de esos dos motivos, los objetos `Custom` pueden participar en ciclos:

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

Para permitir que una instancia de `Custom` que participa en un ciclo de referencia sea detectada y recolectada correctamente por el GC cíclico, nuestro tipo `Custom` necesita llenar dos espacios adicionales y habilitar un indicador que permita estos espacios:

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

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

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

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

static void
```

(continué en la próxima página)

[4] Además, incluso con nuestros atributos restringidos a instancias de cadenas, el usuario podría pasar subclases arbitrarias `str` y, por lo tanto, seguir creando ciclos de referencia.

(proviene de la página anterior)

```
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, *tmp;

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

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
```

(continué en la próxima página)

(proviene de la página anterior)

```
    {NULL}  /* Sentinel */
};

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

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

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

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

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

(continué en la próxima página)

(proviene de la página anterior)

```
     "last name", NULL},
    {NULL}  /* Sentinel */
};

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

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

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom4.Custom",
    .tp_doc = "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 = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom4",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

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

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

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

(continué en la próxima página)

(proviene de la página anterior)

```
    }

    return m;
}
```

Primero, el método transversal permite que el GC cíclico conozca los subobjetos que podrían participar en los ciclos:

```
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;
}
```

Para cada subobjeto que puede participar en ciclos, necesitamos llamar a la función `visit()`, que se pasa al método transversal. La función `visit()` toma como argumentos el subobjeto y el argumento extra *arg* pasados al método transversal. Retorna un valor entero que debe retornarse si no es cero.

Python proporciona una macro `Py_VISIT()` que automatiza las funciones de visita de llamada. Con `Py_VISIT()`, podemos minimizar la cantidad de repeticiones en `Custom_traverse`:

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

**Nota:** La implementación `tp_traverse` debe nombrar sus argumentos exactamente *visit* y *arg* para usar `Py_VISIT()`.

En segundo lugar, debemos proporcionar un método para borrar cualquier subobjeto que pueda participar en los ciclos:

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

Observe el uso de la macro `Py_CLEAR()`. Es la forma recomendada y segura de borrar los atributos de datos de tipos arbitrarios al tiempo que disminuye sus recuentos de referencia. Si tuviera que llamar a `Py_XDECREF()` en lugar del atributo antes de establecerlo en `NULL`, existe la posibilidad de que el destructor del atributo vuelva a llamar al código

que lee el atributo nuevamente (*especialmente* si hay un ciclo de referencia).

---

**Nota:** Puede emular `Py_CLEAR()` escribiendo:

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

Sin embargo, es mucho más fácil y menos propenso a errores usar siempre `Py_CLEAR()` al eliminar un atributo. ¡No intentes micro-optimizar a expensas de la robustez!

---

El desasignador `Custom_dealloc` puede llamar a un código arbitrario al borrar los atributos. Significa que el GC circular se puede activar dentro de la función. Dado que el GC asume que el recuento de referencias no es cero, debemos destrabar el objeto del GC llamando a `PyObject_GC_UnTrack()` antes de borrar los miembros. Aquí está nuestro reubicador reimplementado usando `PyObject_GC_UnTrack()` y `Custom_clear`:

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

Finalmente, agregamos el indicador `Py_TPFLAGS_HAVE_GC` a los indicadores de clase:

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

Eso es prácticamente todo. Si hubiéramos escrito controladores personalizados `tp_alloc` o `tp_free`, tendríamos que modificarlos para la recolección de basura cíclica. La mayoría de las extensiones usarán las versiones proporcionadas automáticamente.

### 2.2.5 Subclases de otros tipos

Es posible crear nuevos tipos de extensión que se derivan de los tipos existentes. Es más fácil heredar de los tipos incorporados, ya que una extensión puede usar fácilmente `PyTypeObject` que necesita. Puede ser difícil compartir estas estructuras `PyTypeObject` entre módulos de extensión.

En este ejemplo crearemos un tipo `SubList` que hereda del tipo incorporado `list`. El nuevo tipo será completamente compatible con las listas regulares, pero tendrá un método adicional `increment()` que aumenta un contador interno:

```
>>> 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>
```

(continué en la próxima página)

(proviene de la página anterior)

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

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

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

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

static PyTypeObject SubListType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "sublist.SubList",
    .tp_doc = "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;
```

(continué en la próxima página)

(proviene de la página anterior)

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

    return m;
}
```

Como puede ver, el código fuente se parece mucho a los ejemplos `Custom` en secciones anteriores. Desglosaremos las principales diferencias entre ellos.

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

La diferencia principal para los objetos de tipo derivado es que la estructura de objeto del tipo base debe ser el primer valor. El tipo base ya incluirá `PyObject_HEAD()` al comienzo de su estructura.

Cuando un objeto Python es una instancia de `SubList`, su puntero `PyObject *` se puede convertir de forma segura tanto en `PyListObject *` como en `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;
}
```

Vemos arriba cómo llamar al método `__init__` del tipo base.

Este patrón es importante cuando se escribe un tipo con miembros personalizados `tp_new` y `tp_dealloc`. El manejador `tp_new` no debería crear realmente la memoria para el objeto con su `tp_alloc`, pero deja que la clase base lo maneje llamando a su propio `tp_new`.

La estructura `PyTypeObject` admite a `tp_base` especificando la clase base concreta del tipo. Debido a problemas de compilación multiplataforma, no puede llenar ese campo directamente con una referencia a `PyList_Type`; debe hacerse más tarde en la función de inicialización del módulo:

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

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

    Py_INCREF(&SubListType);
```

(continué en la próxima página)

(proviene de la página anterior)

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

    return m;
}
```

Antes de llamar a `PyType_Ready()`, la estructura de tipo debe tener el espacio `tp_base` rellenado. Cuando derivamos un tipo existente, no es necesario completar el `tp_alloc` ranura con `PyType_GenericNew()` – la función de asignación del tipo base será heredada.

Después de eso, llamar a `PyType_Ready()` y agregar el objeto tipo al módulo es lo mismo que con los ejemplos básicos `Custom`.

### Notas al pie

## 2.3 Definición de tipos de extensión: temas variados

Esta sección tiene como objetivo dar un vistazo rápido a los diversos métodos de tipo que puede implementar y lo que hacen.

Aquí está la definición de `PyTypeObject`, con algunos campos solo utilizados en las versiones de depuración omitidas:

```
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;
```

(continué en la próxima página)

(proviene de la página anterior)

```
    /* 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 */

    /* call function for all accessible objects */
    traverseproc tp_traverse;

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

    /* 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;
    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;

} PyTypeObject;
```

Esos son *muchos* métodos. Sin embargo, no se preocupe demasiado: si tiene un tipo que desea definir, es muy probable que solo implemente un puñado de estos.

Como probablemente espera ahora, vamos a repasar esto y daremos más información sobre los diversos controladores. No iremos en el orden en que se definen en la estructura, porque hay mucho equipaje histórico que afecta el orden de los

campos. A menudo es más fácil encontrar un ejemplo que incluya los campos que necesita y luego cambiar los valores para adaptarlos a su nuevo tipo.

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

El nombre del tipo – como se mencionó en el capítulo anterior, aparecerá en varios lugares, casi por completo para fines de diagnóstico. ¡Intente elegir algo que sea útil en tal situación!

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

Estos campos le dicen al tiempo de ejecución cuánta memoria asignar cuando se crean nuevos objetos de este tipo. Python tiene algún soporte incorporado para estructuras de longitud variable (piense: cadenas, tuplas) que es donde entra el campo `tp_itemsize`. Esto se tratará más adelante.

```
const char *tp_doc;
```

Aquí puede poner una cadena de caracteres (o su dirección) que desea que se retorne cuando el script de Python haga referencia a `obj.__doc__` para recuperar el docstring.

Ahora llegamos a los métodos de tipo básicos: los que implementarán la mayoría de los tipos de extensión.

### 2.3.1 Finalización y desasignación

```
destructor tp_dealloc;
```

Se llama a esta función cuando el recuento de referencia de la instancia de su tipo se reduce a cero y el intérprete de Python quiere reclamarlo. Si su tipo tiene memoria para liberar u otra limpieza para realizar, puede ponerla aquí. El objeto en sí mismo necesita ser liberado aquí también. Aquí hay un ejemplo de esta función:

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

Un requisito importante de la función desasignador es que deja solo las excepciones pendientes. Esto es importante ya que los desasignadores se llaman con frecuencia cuando el intérprete desenrolla la pila de Python; cuando la pila se desenrolla debido a una excepción (en lugar de retornos normales), no se hace nada para proteger a los desasignadores de memoria (*deallocator*) de ver que ya se ha establecido una excepción. Cualquier acción que realice un desasignador que pueda hacer que se ejecute código Python adicional puede detectar que se ha establecido una excepción. Esto puede conducir a errores engañosos del intérprete. La forma correcta de protegerse contra esto es guardar una excepción pendiente antes de realizar la acción insegura y restaurarla cuando haya terminado. Esto se puede hacer usando las funciones `PyErr_Fetch()` y `PyErr_Restore()`:

```
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 */
```

(continué en la próxima página)

(proviene de la página anterior)

```
        PyErr_Fetch(&err_type, &err_value, &err_traceback);

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

        /* 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);
}
```

**Nota:** Existen limitaciones para lo que puede hacer de manera segura en una función de desasignación. Primero, si su tipo admite la recolección de basura (usando `tp_traverse` o `tp_clear`), algunos de los miembros del objeto pueden haber sido borrados o finalizados por el time `tp_dealloc` es llamado. Segundo, en `tp_dealloc`, su objeto está en un estado inestable: su recuento de referencia es igual a cero. Cualquier llamada a un objeto o API no trivial (como en el ejemplo anterior) podría terminar llamando `tp_dealloc` nuevamente, causando una doble liberación y un bloqueo.

Comenzando con Python 3.4, se recomienda no poner ningún código de finalización complejo en `tp_dealloc`, y en su lugar use el nuevo método de tipo `tp_finalize`.

**Ver también:**

**PEP 442** explica el nuevo esquema de finalización.

### 2.3.2 Presentación de Objetos

En Python, hay dos formas de generar una representación textual de un objeto: la función `repr()`, y la función `str()`. (La función `print()` solo llama a `str()`.) Estos controladores son opcionales.

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

El manejador `tp_repr` debe retornar un objeto de cadena que contenga una representación de la instancia para la que se llama. Aquí hay un ejemplo simple:

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

Si no se especifica `tp_repr`, el intérprete proporcionará una representación que utiliza los tipos `tp_name` y un valor de identificación único para el objeto.

El manejador `tp_str` es para `str()` lo que el manejador `tp_repr` descrito arriba es para `repr()`; es decir, se llama cuando el código Python llama `str()` en una instancia de su objeto. Su implementación es muy similar a la función `tp_repr`, pero la cadena resultante está destinada al consumo humano. Si `tp_str` no se especifica, en su lugar se utiliza el controlador `tp_repr`.

Aquí hay un ejemplo simple:

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

### 2.3.3 Gestión de atributos

Para cada objeto que puede soportar atributos, el tipo correspondiente debe proporcionar las funciones que controlan cómo se resuelven los atributos. Es necesario que haya una función que pueda recuperar atributos (si hay alguna definida), y otra para establecer atributos (si se permite establecer atributos). La eliminación de un atributo es un caso especial, para el cual el nuevo valor pasado al controlador es `NULL`.

Python admite dos pares de controladores de atributos; un tipo que admite atributos solo necesita implementar las funciones para un par. La diferencia es que un par toma el nombre del atributo como a `char*`, mientras que el otro acepta un `PyObject*`. Cada tipo puede usar el par que tenga más sentido para la conveniencia de la implementación.

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

Si acceder a los atributos de un objeto es siempre una operación simple (esto se explicará en breve), existen implementaciones genéricas que se pueden utilizar para proporcionar la versión `PyObject*` de las funciones de gestión de atributos. La necesidad real de controladores de atributos específicos de tipo desapareció casi por completo a partir de Python 2.2, aunque hay muchos ejemplos que no se han actualizado para utilizar algunos de los nuevos mecanismos genéricos que están disponibles.

#### Gestión de atributos genéricos

La mayoría de los tipos de extensión solo usan atributos *simple*. Entonces, ¿qué hace que los atributos sean simples? Solo hay un par de condiciones que se deben cumplir:

1. El nombre de los atributos debe ser conocido cuando `PyType_Ready()` es llamado.
2. No se necesita un procesamiento especial para registrar que un atributo se buscó o se configuró, ni se deben tomar acciones basadas en el valor.

Tenga en cuenta que esta lista no impone restricciones a los valores de los atributos, cuándo se calculan los valores o cómo se almacenan los datos relevantes.

Cuando se llama a `PyType_Ready()`, utiliza tres tablas a las que hace referencia el objeto de tipo para crear *descriptor* que se colocan en el diccionario del objeto de tipo. Cada descriptor controla el acceso a un atributo del objeto de instancia. Cada una de las tablas es opcional; si los tres son `NULL`, las instancias del tipo solo tendrán atributos que se heredan de su tipo base, y deberían dejar `tp_getattro` y los campos `tp_setattro` `NULL` también, permitiendo que el tipo base maneje los atributos.

Las tablas se declaran como tres campos del tipo objeto:

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

Si `tp_methods` no es `NULL`, debe referirse a un arreglo de estructuras `PyMethodDef`. Cada entrada en la tabla es una instancia de esta estructura:

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

Se debe definir una entrada para cada método proporcionado por el tipo; No se necesitan entradas para los métodos heredados de un tipo base. Se necesita una entrada adicional al final; es un centinela el que marca el final del arreglo. El campo `ml_name` del centinela debe ser `NULL`.

La segunda tabla se utiliza para definir atributos que se asignan directamente a los datos almacenados en la instancia. Se admite una variedad de tipos C primitivos, y el acceso puede ser de solo lectura o lectura-escritura. Las estructuras en la tabla se definen como:

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

Para cada entrada en la tabla, se construirá un *descriptor* y se agregará al tipo que podrá extraer un valor de la estructura de la instancia. El campo `type` debe contener uno de los códigos de tipo definidos en el encabezado `structmember.h`; el valor se usará para determinar cómo convertir los valores de Python hacia y desde los valores de C. El campo `flags` se usa para almacenar flags que controlan cómo se puede acceder al atributo.

Las siguientes constantes de flag se definen en `structmember.h`; se pueden combinar usando OR bit a bit (*bitwise-OR*).

| Constante | Significado |
|---|---|
| `READONLY` | Nunca escribible. |
| `READ_RESTRICTED` | No legible en modo restringido. |
| `WRITE_RESTRICTED` | No se puede escribir en modo restringido. |
| `RESTRICTED` | No se puede leer ni escribir en modo restringido. |

Una ventaja interesante de usar la tabla `tp_members` para crear descriptores que se usan en tiempo de ejecución es que cualquier atributo definido de esta manera puede tener un docstring asociada simplemente al proporcionar el texto en la tabla. Una aplicación puede usar la API de introspección para recuperar el descriptor del objeto de clase y obtener el docstring utilizando su atributo `__doc__`.

Al igual que con la tabla `tp_methods`, se requiere una entrada de centinela con un valor `name` de `NULL`.

### Gestión de atributos específicos de tipo

Para simplificar, aquí solo se demostrará la versión `char *`; el tipo de parámetro de nombre es la única diferencia entre las variaciones de la interfaz `char*` y `PyObject*`. Este ejemplo efectivamente hace lo mismo que el ejemplo genérico anterior, pero no utiliza el soporte genérico agregado en Python 2.2. Explica cómo se llaman las funciones del controlador, de modo que si necesita ampliar su funcionalidad, comprenderá lo que debe hacerse.

Se llama al manejador `tp_getattr` cuando el objeto requiere una búsqueda de atributo. Se llama en las mismas situaciones donde se llamaría el método `__getattr__()` de una clase.

Aquí hay un ejemplo:

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

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

Se llama al manejador `tp_setattr` cuando se llama al método `__setattr__()` o `__delattr__()` de una instancia de clase. Cuando se debe eliminar un atributo, el tercer parámetro será `NULL`. Aquí hay un ejemplo que simplemente plantea una excepción; si esto fuera realmente todo lo que deseaba, el controlador `tp_setattr` debería establecerse en `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 Comparación de Objetos

```
richcmpfunc tp_richcompare;
```

Se llama al manejador `tp_richcompare` cuando se necesitan comparaciones. Es análogo a métodos de comparación ricos, como `__lt__()`, y también llamado por `PyObject_RichCompare()` y `PyObject_RichCompareBool()`.

Esta función se llama con dos objetos Python y el operador como argumentos, donde el operador es uno de `Py_EQ`, `Py_NE`, `Py_LE`, `Py_GT`, `Py_LT` o `Py_GT`. Debe comparar los dos objetos con respecto al operador especificado y retornar `Py_True` o `Py_False` si la comparación es exitosa, `Py_NotImplemented` para indicar que la comparación no está implementada y el método de comparación del otro objeto debería intentarse, o `NULL` si se estableció una excepción.

Aquí hay una implementación de muestra, para un tipo de datos que se considera igual si el tamaño de un puntero interno es igual:

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

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

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

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

### 2.3.5 Soporte de protocolo abstracto

Python admite una variedad de protocolos *abstractos*; las interfaces específicas proporcionadas para usar estas interfaces están documentadas en abstract.

Varias de estas interfaces abstractas se definieron temprano en el desarrollo de la implementación de Python. En particular, los protocolos de número, mapeo y secuencia han sido parte de Python desde el principio. Se han agregado otros protocolos con el tiempo. Para los protocolos que dependen de varias rutinas de controlador de la implementación de tipo, los protocolos más antiguos se han definido como bloques opcionales de controladores a los que hace referencia el objeto de tipo. Para los protocolos más nuevos, hay espacios adicionales en el objeto de tipo principal, con un bit de marca que se establece para indicar que los espacios están presentes y el intérprete debe verificarlos. (El bit de indicador no indica que los valores de intervalo no son `NULL`. El indicador puede establecerse para indicar la presencia de un intervalo, pero un intervalo aún puede estar vacío.):

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

Si desea que su objeto pueda actuar como un número, una secuencia o un objeto de mapeo, entonces coloca la dirección de una estructura que implementa el tipo C `PyNumberMethods`, `PySequenceMethods`, o `PyMappingMethods`, respectivamente. Depende de usted completar esta estructura con los valores apropiados. Puede encontrar ejemplos del uso de cada uno de estos en el directorio `Objects` de la distribución fuente de Python.

```
hashfunc tp_hash;
```

Esta función, si elige proporcionarla, debería retornar un número hash para una instancia de su tipo de datos. Aquí hay un ejemplo simple:

```
static Py_hash_t
newdatatype_hash(newdatatypeobject *obj)
```

(continué en la próxima página)

(proviene de la página anterior)

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

`Py_hash_t` es un tipo entero con signo con un ancho que varia dependiendo de la plataforma.retornar `-1` de `tp_hash` indica un error, por lo que debe tener cuidado de evitar retornarlo cuando el cálculo de hash sea exitoso, como se vio anteriormente.

```
ternaryfunc tp_call;
```

Esta función se llama cuando una instancia de su tipo de datos se «llama», por ejemplo, si `obj1` es una instancia de su tipo de datos y el script de Python contiene `obj1('hello')`, el controlador `tp_call` se invoca.

Esta función toma tres argumentos:

1. *self* es la instancia del tipo de datos que es el sujeto de la llamada. Si la llamada es `obj1('hola')`, entonces *self* es `obj1`.
2. *args* es una tupla que contiene los argumentos de la llamada. Puede usar `PyArg_ParseTuple()` para extraer los argumentos.
3. *kwds* es un diccionario de argumentos de palabras clave que se pasaron. Si no es `NULL` y admite argumentos de palabras clave, use `PyArg_ParseTupleAndKeywords()` para extraer los argumentos. Si no desea admitir argumentos de palabras clave y esto no es `NULL`, genere un `TypeError` con un mensaje que indique que los argumentos de palabras clave no son compatibles.

Aquí hay una implementación de juguete `tp_call`:

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

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

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

Estas funciones proporcionan soporte para el protocolo iterador. Ambos manejadores toman exactamente un parámetro, la instancia para la que están siendo llamados, y retornan una nueva referencia. En el caso de un error, deben establecer una excepción y retornar `NULL`. `tp_iter` corresponde al método Python `__iter__()`, mientras que `tp_iternext` corresponde al método Python `__next__()`.

Cualquier objeto *iterable* debe implementar el manejador `tp_iter`, que debe retornar un objeto *iterator*. Aquí se aplican las mismas pautas que para las clases de Python:

- Para colecciones (como listas y tuplas) que pueden admitir múltiples iteradores independientes, cada llamada debe crear y retornar un nuevo iterador a `tp_iter`.
- Los objetos que solo se pueden iterar una vez (generalmente debido a los efectos secundarios de la iteración, como los objetos de archivo) pueden implementar `tp_iter` retornando una nueva referencia a ellos mismos y, por lo tanto, también deben implementar el manejador `tp_iternext`.

Cualquier objeto *iterator* debe implementar tanto `tp_iter` como `tp_iternext`. El manejador de un iterador `tp_iter` debería retornar una nueva referencia al iterador. Su controlador `tp_iternext` debería retornar una nueva referencia al siguiente objeto en la iteración, si hay uno. Si la iteración ha llegado al final, `tp_iternext` puede retornar `NULL` sin establecer una excepción, o puede establecer `StopIteration` *además* para retornar `NULL`; evitar la excepción puede producir un rendimiento ligeramente mejor. Si se produce un error real, `tp_iternext` siempre debe establecer una excepción y retornar `NULL`.

### 2.3.6 Soporte de referencia débil

Uno de los objetivos de la implementación de referencia débil de Python es permitir que cualquier tipo participe en el mecanismo de referencia débil sin incurrir en la sobrecarga de objetos críticos para el rendimiento (como los números).

**Ver también:**

Documentación para el módulo `weakref`.

Para que un objeto sea débilmente referenciable, el tipo de extensión debe hacer dos cosas:

1. Incluya el campo a `PyObject*` en la estructura del objeto C dedicada al mecanismo de referencia débil. El constructor del objeto debe dejarlo `NULL` (que es automático cuando se usa el valor predeterminado `tp_alloc`).
2. Establezca el miembro de tipo `tp_weaklistoffset` en el desplazamiento del campo mencionado anteriormente en la estructura del objeto C, para que el intérprete sepa cómo acceder y modificar ese campo.

Concretamente, así es como una estructura de objeto trivial se aumentaría con el campo requerido:

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

Y el miembro correspondiente en el objeto de tipo declarado estáticamente:

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

La única adición adicional es que `tp_dealloc` necesita borrar cualquier referencia débil (llamando a `PyObject_ClearWeakRefs()`) si el campo no es `NULL`

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

(continué en la próxima página)

(proviene de la página anterior)

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

### 2.3.7 Más Sugerencias

Para aprender a implementar cualquier método específico para su nuevo tipo de datos, obtenga el código fuente *CPython*. Vaya al directorio: file: *Objects*, luego busque en los archivos fuente C `tp_` más la función que desee (por ejemplo, `tp_richcompare`). Encontrará ejemplos de la función que desea implementar.

Cuando necesite verificar que un objeto es una instancia concreta del tipo que está implementando, use la función `PyObject_TypeCheck()`. Una muestra de su uso podría ser algo como lo siguiente:

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

**Ver también:**

**Descargue las versiones de origen de CPython.** https://www.python.org/downloads/source/

**El proyecto CPython en GitHub, donde se desarrolla el código fuente de CPython.** https://github.com/python/cpython

## 2.4 Construyendo Extensiones C y C++

Una extensión C para CPython es una biblioteca compartida (por ejemplo, un archivo `.so` en Linux, `.pyd` en Windows), que exporta una *función de inicialización*.

Para que sea importable, la biblioteca compartida debe estar disponible en `PYTHONPATH`, y debe tener el nombre del módulo, con una extensión adecuada. Cuando se usan distutils, el nombre de archivo correcto se genera automáticamente.

La función de inicialización tiene la firma:

PyObject* **`PyInit_modulename`**(void)

Retorna un módulo completamente inicializado o una instancia `PyModuleDef`. Ver initializing-modules para más detalles.

Para los módulos con nombres solo ASCII, la función debe llamarse `PyInit_<modulename>`, con `<modulename>` reemplazado por el nombre del módulo. Cuando se usa multi-phase-initialization, se permiten nombres de módulos que no sean ASCII. En este caso, el nombre de la función de inicialización es `PyInitU_<modulename>`, con `<modulename>` codificado usando la codificación *punycode* de Python con guiones reemplazados por guiones bajos. En 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
```

Es posible exportar múltiples módulos desde una única biblioteca compartida definiendo múltiples funciones de inicialización. Sin embargo, importarlos requiere el uso de enlaces simbólicos o un importador personalizado, porque de

forma predeterminada solo se encuentra la función correspondiente al nombre del archivo. Consulte la sección *«Múltiples módulos en una biblioteca»* en **PEP 489** para más detalles.

### 2.4.1 Construyendo Extensiones C y C++ con distutils

Los módulos de extensión se pueden construir utilizando distutils, que se incluye en Python. Dado que distutils también admite la creación de paquetes binarios, los usuarios no necesitan necesariamente un compilador y distutils para instalar la extensión.

Un paquete distutils contiene un script de controlador, `setup.py`. Este es un archivo Python simple, que, en el caso más simple, podría verse así:

```
from distutils.core import setup, Extension

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

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

Con esto `setup.py`, y un archivo `demo.c`, ejecutando:

```
python setup.py build
```

compilará `demo.c`, y producirá un módulo de extensión llamado `demo` en el directorio `build`. Dependiendo del sistema, el archivo del módulo terminará en un subdirectorio `build/lib.system`, y puede tener un nombre como `demo.so` o `demo.pyd`.

En `setup.py`, toda la ejecución se realiza llamando a la función `setup`. Esto toma un número variable de argumentos de palabras clave, de los cuales el ejemplo anterior usa solo un subconjunto. Específicamente, el ejemplo especifica metainformación para construir paquetes, y especifica el contenido del paquete. Normalmente, un paquete contendrá módulos adicionales, como módulos fuente Python, documentación, subpaquetes, etc. Consulte la documentación de distutils en distutils-index para obtener más información sobre las características de distutils; Esta sección explica la construcción de módulos de extensión solamente.

Es común precalcular argumentos para `setup()`, para estructurar mejor el script del controlador. En el ejemplo anterior, el argumento `ext_modules` para `setup()` es una lista de módulos de extensión, cada uno de los cuales es una instancia de `Extension`. En el ejemplo, la instancia define una extensión llamada `demo` que se construye compilando un solo archivo fuente `demo.c`.

En muchos casos, construir una extensión es más complejo, ya que es posible que se necesiten preprocesadores adicionales y bibliotecas. Esto se demuestra en el siguiente ejemplo.

```
from distutils.core import setup, Extension

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

setup (name = 'PackageName',
       version = '1.0',
```

(continué en la próxima página)

(proviene de la página anterior)

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

En este ejemplo, se llama a `setup()` con metainformación adicional, que se recomienda cuando se deben construir paquetes de distribución. Para la extensión en sí, especifica las definiciones de preprocesador, incluye directorios, directorios de biblioteca y bibliotecas. Dependiendo del compilador, distutils pasa esta información de diferentes maneras al compilador. Por ejemplo, en Unix, esto puede resultar en los comandos de compilación:

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

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

Estas líneas son solo para fines de demostración; Los usuarios de distutils deben confiar en que distutils obtiene las invocaciones correctas.

### 2.4.2 Distribuyendo sus módulos de extensión

Cuando una extensión se ha creado correctamente, hay tres formas de usarla.

Los usuarios finales generalmente querrán instalar el módulo, lo hacen ejecutando:

```
python setup.py install
```

Los mantenedores de módulos deben producir paquetes fuente; para hacerlo, ejecutan:

```
python setup.py sdist
```

En algunos casos, se deben incluir archivos adicionales en una distribución de origen; esto se hace a través de un archivo `MANIFEST.in`; ver manifest para más detalles.

Si la distribución de origen se ha creado correctamente, los encargados del mantenimiento también pueden crear distribuciones binarias. Dependiendo de la plataforma, se puede usar uno de los siguientes comandos para hacerlo.:

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

## 2.5 Creación de extensiones C y C++ en Windows

Este capítulo explica brevemente cómo crear un módulo de extensión de Windows para Python usando Microsoft Visual C++, y sigue con información de fondo más detallada sobre cómo funciona. El material explicativo es útil tanto para el programador de Windows que está aprendiendo a construir extensiones de Python como para el programador de Unix interesado en producir software que se pueda construir con éxito tanto en Unix como en Windows.

Se alienta a los autores de módulos a utilizar el enfoque distutils para construir módulos de extensión, en lugar del descrito en esta sección. Aún necesitará el compilador de C que se utilizó para construir Python; típicamente Microsoft Visual C++.

---

**Nota:** Este capítulo menciona varios nombres de archivo que incluyen un número de versión codificado de Python. Estos nombres de archivo se representan con el número de versión que se muestra como `XY`; en la práctica, `'X'` será el número de versión principal y `'Y'` será el número de versión menor de la versión de Python con la que está trabajando. Por ejemplo, si está utilizando Python 2.2.1, `XY` en realidad será `22`.

---

### 2.5.1 Un enfoque de libro de cocina

Hay dos enfoques para construir módulos de extensión en Windows, al igual que en Unix: use el paquete `distutils` para controlar el proceso de construcción, o haga las cosas manualmente. El enfoque distutils funciona bien para la mayoría de las extensiones; La documentación sobre el uso de `distutils` para compilar y empaquetar módulos de extensión está disponible en distutils-index. Si encuentra que realmente necesita hacer las cosas manualmente, puede ser instructivo estudiar el archivo del proyecto para el módulo de biblioteca estándar winsound.

### 2.5.2 Diferencias entre Unix y Windows

Unix y Windows usan paradigmas completamente diferentes para la carga de código en tiempo de ejecución. Antes de intentar construir un módulo que se pueda cargar dinámicamente, tenga en cuenta cómo funciona su sistema.

En Unix, un archivo de objeto compartido (`.so`) contiene código para ser utilizado por el programa, y también los nombres de funciones y datos que espera encontrar en el programa. Cuando el archivo se une al programa, todas las referencias a esas funciones y datos en el código del archivo se cambian para apuntar a las ubicaciones reales en el programa donde las funciones y los datos se colocan en la memoria. Esto es básicamente una operación de enlace.

En Windows, un archivo de biblioteca de enlace dinámico (`.dll`) no tiene referencias colgantes. En cambio, un acceso a funciones o datos pasa por una tabla de búsqueda. Por lo tanto, el código DLL no tiene que repararse en tiempo de ejecución para referirse a la memoria del programa; en cambio, el código ya usa la tabla de búsqueda de la DLL, y la tabla de búsqueda se modifica en tiempo de ejecución para apuntar a las funciones y los datos.

En Unix, solo hay un tipo de archivo de biblioteca (`.a`) que contiene código de varios archivos de objeto (`.o`). Durante el paso de enlace para crear un archivo de objeto compartido (`.so`), el enlazador puede encontrar que no sabe dónde se define un identificador. El enlazador lo buscará en los archivos de objetos en las bibliotecas; si lo encuentra, incluirá todo el código de ese archivo de objeto.

En Windows, hay dos tipos de biblioteca, una biblioteca estática y una biblioteca de importación (ambas llamadas `.lib`). Una biblioteca estática es como un archivo Unix `.a`; Contiene código para ser incluido según sea necesario. Una biblioteca de importación se usa básicamente solo para asegurarle al enlazador que cierto identificador es legal y estará presente en el programa cuando se cargue la DLL. Por lo tanto, el enlazador utiliza la información de la biblioteca de importación para crear la tabla de búsqueda para usar identificadores que no están incluidos en la DLL. Cuando se vincula una aplicación o una DLL, se puede generar una biblioteca de importación, que deberá usarse para todas las DLL futuras que dependan de los símbolos en la aplicación o DLL.

Suponga que está creando dos módulos de carga dinámica, B y C, que deberían compartir otro bloque de código A. En Unix, *no* pasaría `A.a` al enlazador para `B.so` y `C.so`; eso haría que se incluyera dos veces, de modo que B y C tengan cada uno su propia copia. En Windows, compilar `A.dll` también compilará `A.lib`. Usted *si* pasa `A.lib` al enlazador para B y C. `A.lib` no contiene código; solo contiene información que se usará en tiempo de ejecución para acceder al código de A.

En Windows, usar una biblioteca de importación es como usar `importar spam`; le da acceso a los nombres de spam, pero no crea una copia separada. En Unix, vincular con una biblioteca es más como `from spam import*`; crea una copia separada.

### 2.5.3 Usar DLL en la práctica

Windows Python está construido en Microsoft Visual C++; el uso de otros compiladores puede o no funcionar (aunque Borland parece funcionar). El resto de esta sección es específica de MSVC++.

Al crear archivos DLL en Windows, debe pasar `pythonXY.lib` al enlazador. Para construir dos DLL, spam y ni (que usa funciones C que se encuentran en el spam), puede usar estos comandos:

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

El primer comando creó tres archivos: `spam.obj`, `spam.dll` y `spam.lib`. `Spam.dll` no contiene ninguna función de Python (como `PyArg_ParseTuple()`), pero sabe cómo encontrar el código de Python gracias a `pythonXY.lib`.

El segundo comando creó `ni.dll` (y `.obj` y `.lib`), que sabe cómo encontrar las funciones necesarias del spam, y también del ejecutable de Python.

No todos los identificadores se exportan a la tabla de búsqueda. Si desea que cualquier otro módulo (incluido Python) pueda ver sus identificadores, debe decir `_declspec(dllexport)`, como en `void _declspec(dllexport) initspam(void)` o `PyObject_declspec(dllexport) *NiGetSpamData(void)`.

Developer Studio incluirá muchas bibliotecas de importación que realmente no necesita, agregando aproximadamente 100K a su ejecutable. Para deshacerse de ellos, use el cuadro de diálogo Configuración del proyecto, pestaña Enlace, para especificar *ignorar las bibliotecas predeterminadas*. Agregue el archivo correcto `msvcrtxx.lib` a la lista de bibliotecas.

CAPÍTULO 3

# Incrustar el tiempo de ejecución de CPython en una aplicación más grande

A veces, en lugar de crear una extensión que se ejecute dentro del intérprete de Python como la aplicación principal, es conveniente incorporar el tiempo de ejecución de CPython dentro de una aplicación más grande. Esta sección cubre algunos de los detalles involucrados en hacerlo con éxito.

## 3.1 Incrustando Python en Otra Aplicación

Los capítulos anteriores discutieron cómo extender Python, es decir, cómo extender la funcionalidad de Python al adjuntarle una biblioteca de funciones C. También es posible hacerlo al revés: enriquezca su aplicación C/C++ incrustando Python en ella. La incrustación proporciona a su aplicación la capacidad de implementar parte de la funcionalidad de su aplicación en Python en lugar de C o C++. Esto se puede usar para muchos propósitos; Un ejemplo sería permitir a los usuarios adaptar la aplicación a sus necesidades escribiendo algunos scripts en Python. También puede usarlo usted mismo si parte de la funcionalidad se puede escribir en Python más fácilmente.

Incrustar Python es similar a extenderlo, pero no del todo. La diferencia es que cuando extiende Python, el programa principal de la aplicación sigue siendo el intérprete de Python, mientras que si incrusta Python, el programa principal puede no tener nada que ver con Python — en cambio, algunas partes de la aplicación ocasionalmente llaman al Intérprete de Python para ejecutar algún código de Python.

Entonces, si está incrustando Python, está proporcionando su propio programa principal. Una de las cosas que tiene que hacer este programa principal es inicializar el intérprete de Python. Como mínimo, debe llamar a la función `Py_Initialize()`. Hay llamadas opcionales para pasar argumentos de línea de comandos a Python. Luego, puede llamar al intérprete desde cualquier parte de la aplicación.

Hay varias formas diferentes de llamar al intérprete: puede pasar una cadena que contiene declaraciones de Python a `PyRun_SimpleString()`, o puede pasar un puntero de archivo estándar y un nombre de archivo (solo para identificación en mensajes de error) a `PyRun_SimpleFile()`. También puede llamar a las operaciones de nivel inferior descritas en los capítulos anteriores para construir y usar objetos de Python.

**Ver también:**

**c-api-index** Los detalles de la interfaz C de Python se dan en este manual. Una gran cantidad de información necesaria se puede encontrar aquí.

### 3.1.1 Incrustación de muy alto nivel

La forma más simple de incrustar Python es el uso de la interfaz de muy alto nivel. Esta interfaz está diseñada para ejecutar un script de Python sin necesidad de interactuar directamente con la aplicación. Esto puede usarse, por ejemplo, para realizar alguna operación en un archivo.

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

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

La función `Py_SetProgramName()` debe llamarse antes de `Py_Initialize()` para informar al intérprete sobre las rutas a las bibliotecas de tiempo de ejecución de Python. A continuación, el intérprete de Python se inicializa con `Py_Initialize()`, seguido de la ejecución de un script Python codificado que imprime la fecha y la hora. Luego, la llamada `Py_FinalizeEx()` cierra el intérprete, seguido por el final del programa. En un programa real, es posible que desee obtener el script de Python de otra fuente, tal vez una rutina de editor de texto, un archivo o una base de datos. Obtener el código Python de un archivo se puede hacer mejor usando la función `PyRun_SimpleFile()`, que le ahorra la molestia de asignar espacio de memoria y cargar el contenido del archivo.

### 3.1.2 Más allá de la incrustación de muy alto nivel: una visión general

La interfaz de alto nivel le permite ejecutar piezas arbitrarias de código Python desde su aplicación, pero el intercambio de valores de datos es bastante engorroso, por decir lo menos. Si lo desea, debe usar llamadas de nivel inferior. A costa de tener que escribir más código C, puede lograr casi cualquier cosa.

Cabe señalar que extender Python e incrustar Python es la misma actividad, a pesar de la intención diferente. La mayoría de los temas tratados en los capítulos anteriores siguen siendo válidos. Para mostrar esto, considere lo que realmente hace el código de extensión de Python a C:

1. Convierte valores de datos de Python a C,
2. Realice una llamada de función a una rutina C usando los valores convertidos, y
3. Convierte los valores de datos de la llamada de C a Python.

Al incrustar Python, el código de interfaz hace:

1. Convierte valores de datos de C a Python,
2. Realice una llamada de función a una rutina de interfaz de Python utilizando los valores convertidos, y
3. Convierte los valores de datos de la llamada de Python a C.

Como puede ver, los pasos de conversión de datos simplemente se intercambian para acomodar la dirección diferente de la transferencia de idiomas cruzados. La única diferencia es la rutina que llama entre ambas conversiones de datos. Al extender, llama a una rutina C, al incrustar, llama a una rutina Python.

Este capítulo no discutirá cómo convertir datos de Python a C y viceversa. Además, se supone que se entiende el uso adecuado de las referencias y el tratamiento de errores. Dado que estos aspectos no difieren de extender el intérprete, puede consultar los capítulos anteriores para obtener la información requerida.

### 3.1.3 Incrustación pura

El primer programa tiene como objetivo ejecutar una función en un script Python. Al igual que en la sección sobre la interfaz de muy alto nivel, el intérprete de Python no interactúa directamente con la aplicación (pero eso cambiará en la siguiente sección).

El código para ejecutar una función definida en un script de Python es:

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

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

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

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

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

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

        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(argc - 3);
            for (i = 0; i < argc - 3; ++i) {
                pValue = PyLong_FromLong(atoi(argv[i + 3]));
                if (!pValue) {
                    Py_DECREF(pArgs);
                    Py_DECREF(pModule);
                    fprintf(stderr, "Cannot convert argument\n");
                    return 1;
                }
                /* pValue reference stolen here: */
                PyTuple_SetItem(pArgs, i, pValue);
            }
            pValue = PyObject_CallObject(pFunc, pArgs);
            Py_DECREF(pArgs);
```

(continué en la próxima página)

(proviene de la página anterior)

```
            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;
}
```

Este código carga un script de Python usando `argv[1]` y llama a la función nombrada en `argv[2]`. Sus argumentos enteros son los otros valores del arreglo `argv`. Si usted *compila y enlaza* este programa (llamemos al ejecutable terminado **`call`**), y úselo para ejecutar un script Python, como:

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

entonces el resultado debería ser:

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

Aunque el programa es bastante grande por su funcionalidad, la mayor parte del código es para la conversión de datos entre Python y C, y para informes de errores. La parte interesante con respecto a incrustar Python comienza con:

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

Después de inicializar el intérprete, el script se carga usando `PyImport_Import()`. Esta rutina necesita una cadena

Python como argumento, que se construye utilizando la rutina de conversión de datos `PyUnicode_FromString()`.

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

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

Una vez que se carga el script, el nombre que estamos buscando se recupera usando `PyObject_GetAttrString()`. Si el nombre existe y el objeto retornado es invocable, puede asumir con seguridad que es una función. Luego, el programa continúa construyendo una tupla de argumentos como de costumbre. La llamada a la función Python se realiza con:

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

Al regresar la función, `pValue` es `NULL` o contiene una referencia al valor de retorno de la función. Asegúrese de liberar la referencia después de examinar el valor.

### 3.1.4 Extendiendo Python Incrustado

Hasta ahora, el intérprete de Python incorporado no tenía acceso a la funcionalidad de la aplicación misma. La API de Python lo permite al extender el intérprete incorporado. Es decir, el intérprete incorporado se amplía con las rutinas proporcionadas por la aplicación. Si bien suena complejo, no es tan malo. Simplemente olvide por un momento que la aplicación inicia el intérprete de Python. En cambio, considere que la aplicación es un conjunto de subrutinas y escriba un código de pegamento que le otorgue a Python acceso a esas rutinas, al igual que escribiría una extensión normal de Python. Por ejemplo:

```
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);
}
```

Inserte el código anterior justo encima de la función `main()`. Además, inserte las siguientes dos declaraciones antes de la llamada a `Py_Initialize()`:

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

Estas dos líneas inicializan la variable `numargs` y hacen que la función `emb.numargs()` sea accesible para el intérprete de Python incorporado. Con estas extensiones, el script de Python puede hacer cosas como

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

En una aplicación real, los métodos expondrán una API de la aplicación a Python.

### 3.1.5 Incrustando Python en C++

También es posible incrustar Python en un programa C++; precisamente cómo se hace esto dependerá de los detalles del sistema C++ utilizado; en general, necesitará escribir el programa principal en C++ y usar el compilador de C++ para compilar y vincular su programa. No es necesario volver a compilar Python usando C++.

### 3.1.6 Compilar y enlazar bajo sistemas tipo Unix

No es necesariamente trivial encontrar los indicadores correctos para pasar a su compilador (y enlazador) para incrustar el intérprete de Python en su aplicación, particularmente porque Python necesita cargar módulos de biblioteca implementados como extensiones dinámicas en C (archivos `.so`) enlazados en su contra.

Para conocer los indicadores necesarios del compilador y el enlazador, puede ejecutar el script `pythonX.Y-config` que se genera como parte del proceso de instalación (también puede estar disponible un script `python3-config` ) Este script tiene varias opciones, de las cuales las siguientes serán directamente útiles para usted:

- `pythonX.Y-config --cflags` le dará las banderas recomendadas al compilar:

```
$ /opt/bin/python3.4-config --cflags
-I/opt/include/python3.4m -I/opt/include/python3.4m -DNDEBUG -g -fwrapv -O3 -Wall␣
↪-Wstrict-prototypes
```

- `pythonX.Y-config --ldflags` le dará las banderas recomendadas al vincular:

```
$ /opt/bin/python3.4-config --ldflags
-L/opt/lib/python3.4/config-3.4m -lpthread -ldl -lutil -lm -lpython3.4m -Xlinker -
↪export-dynamic
```

**Nota:** Para evitar confusiones entre varias instalaciones de Python (y especialmente entre el sistema Python y su propio Python compilado), se recomienda que use la ruta absoluta a `pythonX.Y-config`, como en el ejemplo anterior.

Si este procedimiento no funciona para usted (no se garantiza que funcione para todas las plataformas tipo Unix; sin embargo, le damos la bienvenida informes de errores) deberá leer la documentación de su sistema sobre dinámica vincular o examinar Python `Makefile` (use `sysconfig.get_makefile_filename()` para encontrar su ubicación) y las opciones de compilación. En este caso, el módulo `sysconfig` es una herramienta útil para extraer mediante programación los valores de configuración que querrá combinar. Por ejemplo:

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

# APÉNDICE A

## Glosario

**>>>** El prompt en el shell interactivo de Python por omisión. Frecuentemente vistos en ejemplos de código que pueden ser ejecutados interactivamente en el intérprete.

**. . .** Puede referirse a:

- El prompt en el shell interactivo de Python por omisión cuando se ingresa código para un bloque indentado de código, y cuando se encuentra entre dos delimitadores que emparejan (paréntesis, corchetes, llaves o comillas triples), o después de especificar un decorador.
- La constante incorporada `Ellipsis`.

**2to3** Una herramienta que intenta convertir código de Python 2.x a Python 3.x arreglando la mayoría de las incompatibilidades que pueden ser detectadas analizando el código y recorriendo el árbol de análisis sintáctico.

2to3 está disponible en la biblioteca estándar como `lib2to3`; un punto de entrada independiente es provisto como `Tools/scripts/2to3`. Vea 2to3-reference.

**clase base abstracta** Las clases base abstractas (ABC, por sus siglas en inglés *Abstract Base Class*) complementan al *duck-typing* brindando un forma de definir interfaces con técnicas como `hasattr()` que serían confusas o sutilmente erróneas (por ejemplo con magic methods). Las ABC introduce subclases virtuales, las cuales son clases que no heredan desde una clase pero aún así son reconocidas por `isinstance()` y `issubclass()`; vea la documentación del módulo `abc`. Python viene con muchas ABC incorporadas para las estructuras de datos( en el módulo `collections.abc`), números (en el módulo `numbers` ) , flujos de datos (en el módulo `io` ) , buscadores y cargadores de importaciones (en el módulo `importlib.abc` ) . Puede crear sus propios ABCs con el módulo `abc`.

**anotación** Una etiqueta asociada a una variable, atributo de clase, parámetro de función o valor de retorno, usado por convención como un *type hint*.

Las anotaciones de variables no pueden ser accedidas en tiempo de ejecución, pero las anotaciones de variables globales, atributos de clase, y funciones son almacenadas en el atributo especial `__annotations__` de módulos, clases y funciones, respectivamente.

Vea *variable annotation*, *function annotation*, **PEP 484** y **PEP 526**, los cuales describen esta funcionalidad.

**argumento** Un valor pasado a una *function* (o *method*) cuando se llama a la función. Hay dos clases de argumentos:

- *argumento nombrado*: es un argumento precedido por un identificador (por ejemplo, `nombre=`) en una llamada a una función o pasado como valor en un diccionario precedido por `**`. Por ejemplo `3` y `5` son argumentos nombrados en las llamadas a `complex()`:

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

- *argumento posicional* son aquellos que no son nombrados. Los argumentos posicionales deben aparecer al principio de una lista de argumentos o ser pasados como elementos de un *iterable* precedido por `*`. Por ejemplo, `3` y `5` son argumentos posicionales en las siguientes llamadas:

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

Los argumentos son asignados a las variables locales en el cuerpo de la función. Vea en la sección calls las reglas que rigen estas asignaciones. Sintácticamente, cualquier expresión puede ser usada para representar un argumento; el valor evaluado es asignado a la variable local.

Vea también el *parameter* en el glosario, la pregunta frecuente la diferencia entre argumentos y parámetros, y **PEP 362**.

**administrador asincrónico de contexto** Un objeto que controla el entorno visible en un sentencia `async with` al definir los métodos `__aenter__()` `__aexit__()`. Introducido por **PEP 492**.

**generador asincrónico** Una función que retorna un *asynchronous generator iterator*. Es similar a una función corrutina definida con `async def` excepto que contiene expresiones `yield` para producir series de variables usadas en un ciclo `async for`.

Usualmente se refiere a una función generadora asincrónica, pero puede referirse a un *iterador generador asincrónico* en ciertos contextos. En aquellos casos en los que el significado no está claro, usar los términos completos evita la ambigüedad.

Una función generadora asincrónica puede contener expresiones `await` así como sentencias `async for`, y `async with`.

**iterador generador asincrónico** Un objeto creado por una función *asynchronous generator*.

Este es un *asynchronous iterator* el cual cuando es llamado usa el método `__anext__()` retornando un objeto a la espera (*awaitable*) el cual ejecutará el cuerpo de la función generadora asincrónica hasta la siguiente expresión `yield`.

Cada `yield` suspende temporalmente el procesamiento, recordando el estado local de ejecución (incluyendo a las variables locales y las sentencias *try* pendientes). Cuando el *iterador del generador asincrónico* vuelve efectivamente con otro objeto a la espera (*awaitable*) retornado por el método `__anext__()`, retoma donde lo dejó. Vea **PEP 492** y **PEP 525**.

**iterable asincrónico** Un objeto, que puede ser usado en una sentencia `async for`. Debe retornar un *asynchronous iterator* de su método `__aiter__()`. Introducido por **PEP 492**.

**iterador asincrónico** Un objeto que implementa los métodos `__aiter__()` y `__anext__()`. `__anext__` debe retornar un objeto *awaitable*. `async for` resuelve los esperables retornados por un método de iterador asincrónico `__anext__()` hasta que lanza una excepción `StopAsyncIteration`. Introducido por **PEP 492**.

**atributo** Un valor asociado a un objeto que es referencias por el nombre usado expresiones de punto. Por ejemplo, si un objeto *o* tiene un atributo *a* sería referenciado como *o.a*.

**a la espera** Es un objeto a la espera (*awaitable*) que puede ser usado en una expresión `await`. Puede ser una *coroutine* o un objeto con un método `__await__()`. Vea también **PEP 492**.

**BDFL** Sigla de *Benevolent Dictator For Life*, benevolente dictador vitalicio, es decir Guido van Rossum, el creador de Python.

**archivo binario** Un *file object* capaz de leer y escribir *objetos tipo binarios*. Ejemplos de archivos binarios son los abiertos en modo binario (`'rb'`, `'wb'` o `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, e instancias de `io.BytesIO` y de `gzip.GzipFile`.

Vea también *text file* para un objeto archivo capaz de leer y escribir objetos `str`.

**objetos tipo binarios** Un objeto que soporta bufferobjects y puede exportar un búfer C-*contiguous*. Esto incluye todas los objetos `bytes`, `bytearray`, y `array.array`, así como muchos objetos comunes `memoryview`. Los objetos tipo binarios pueden ser usados para varias operaciones que usan datos binarios; éstas incluyen compresión, salvar a archivos binarios, y enviarlos a través de un socket.

Algunas operaciones necesitan que los datos binarios sean mutables. La documentación frecuentemente se refiere a éstos como «objetos tipo binario de lectura y escritura». Ejemplos de objetos de búfer mutables incluyen a `bytearray` y `memoryview` de la `bytearray`. Otras operaciones que requieren datos binarios almacenados en objetos inmutables («objetos tipo binario de sólo lectura»); ejemplos de éstos incluyen `bytes` y `memoryview` del objeto `bytes`.

**bytecode** El código fuente Python es compilado en *bytecode*, la representación interna de un programa python en el intérprete CPython. El *bytecode* también es guardado en caché en los archivos *.pyc* de tal forma que ejecutar el mismo archivo es más fácil la segunda vez (la recompilación desde el código fuente a *bytecode* puede ser evitada). Este «lenguaje intermedio» deberá corren en una *virtual machine* que ejecute el código de máquina correspondiente a cada *bytecode*. Note que los *bytecodes* no tienen como requisito trabajar en las diversas máquina virtuales de Python, ni de ser estable entre versiones Python.

Una lista de las instrucciones en *bytecode* está disponible en la documentación de el módulo dis.

**callback** A subroutine function which is passed as an argument to be executed at some point in the future.

**clase** Una plantilla para crear objetos definidos por el usuario. Las definiciones de clase normalmente contienen definiciones de métodos que operan una instancia de la clase.

**variable de clase** Una variable definida en una clase y prevista para ser modificada sólo a nivel de clase (es decir, no en una instancia de la clase).

**coerción** La conversión implícita de una instancia de un tipo en otra durante una operación que involucra dos argumentos del mismo tipo. Por ejemplo, `int(3.15)` convierte el número de punto flotante al entero `3`, pero en `3 + 4.5`, cada argumento es de un tipo diferente (uno entero, otro flotante), y ambos deben ser convertidos al mismo tipo antes de que puedan ser sumados o emitiría un `TypeError`. Sin coerción, todos los argumentos, incluso de tipos compatibles, deberían ser normalizados al mismo tipo por el programador, por ejemplo `float(3)+4.5` en lugar de `3+4.5`.

**número complejo** Una extensión del sistema familiar de número reales en el cual los números son expresados como la suma de una parte real y una parte imaginaria. Los números imaginarios son múltiplos de la unidad imaginaria (la raíz cuadrada de `-1`), usualmente escrita como `i` en matemáticas o `j` en ingeniería. Python tiene soporte incorporado para números complejos, los cuales son escritos con la notación mencionada al final.; la parte imaginaria es escrita con un sufijo `j`, por ejemplo, `3+1j`. Para tener acceso a los equivalentes complejos del módulo `math` module, use `cmath`. El uso de números complejos es matemática bastante avanzada. Si no le parecen necesarios, puede ignorarlos sin inconvenientes.

**administrador de contextos** Un objeto que controla el entorno en la sentencia `with` definiendo los métodos `__enter__()` y `__exit__()`. Vea **PEP 343**.

**variable de contexto** Una variable que puede tener diferentes valores dependiendo del contexto. Esto es similar a un almacenamiento de hilo local *Thread-Local Storage* en el cual cada hilo de ejecución puede tener valores diferentes para una variable. Sin embargo, con las variables de contexto, podría haber varios contextos en un hilo de ejecución y el uso principal de las variables de contexto es mantener registro de las variables en tareas concurrentes asíncronas. Vea `contextvars`.

**contiguo** Un búfer es considerado contiguo con precisión si es *C-contiguo* o *Fortran contiguo*. Los búferes cero dimensionales con C y Fortran contiguos. En los arreglos unidimensionales, los ítems deben ser dispuestos en memoria

uno siguiente al otro, ordenados por índices que comienzan en cero. En arreglos unidimensionales C-contiguos, el último índice varía más velozmente en el orden de las direcciones de memoria. Sin embargo, en arreglos Fortran contiguos, el primer índice vería más rápidamente.

**corrutina** Las corrutinas son una forma más generalizadas de las subrutinas. A las subrutinas se ingresa por un punto y se sale por otro punto. Las corrutinas pueden se iniciadas, finalizadas y reanudadas en muchos puntos diferentes. Pueden ser implementadas con la sentencia `async def`. Vea además **PEP 492**.

**función corrutina** Un función que retorna un objeto *coroutine* . Una función corrutina puede ser definida con la sentencia `async def`, y puede contener las palabras claves `await`, `async for`, y `async with`. Las mismas son introducidas en **PEP 492**.

**CPython** La implementación canónica del lenguaje de programación Python, como se distribuye en python.org. El término «CPython» es usado cuando es necesario distinguir esta implementación de otras como *Jython* o *IronPython*.

**decorador** Una función que retorna otra función, usualmente aplicada como una función de transformación empleando la sintaxis `@envoltorio`. Ejemplos comunes de decoradores son `classmethod()` y `staticmethod()`.

La sintaxis del decorador es meramente azúcar sintáctico, las definiciones de las siguientes dos funciones son semánticamente equivalentes:

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

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

El mismo concepto existe para clases, pero son menos usadas. Vea la documentación de function definitions y class definitions para mayor detalle sobre decoradores.

**descriptor** Cualquier objeto que define los métodos `__get__()`, `__set__()`, o `__delete__()`. Cuando un atributo de clase es un descriptor, su conducta enlazada especial es disparada durante la búsqueda del atributo. Normalmente, usando *a.b* para consultar, establecer o borrar un atributo busca el objeto llamado *b* en el diccionario de clase de *a*, pero si *b* es un descriptor, el respectivo método descriptor es llamado. Entender descriptores es clave para lograr una comprensión profunda de Python porque son la base de muchas de las capacidades incluyendo funciones, métodos, propiedades, métodos de clase, métodos estáticos, y referencia a súper clases.

Para mayor información sobre los métodos de los descriptores vea descriptors.

**diccionario** Un arreglo asociativo, con claves arbitrarias que son asociadas a valores. Las claves pueden ser cualquier objeto con los métodos `__hash__()` y `__eq__()` . Son llamadas hash en Perl.

**dictionary comprehension** A compact way to process all or part of the elements in an iterable and return a dictionary with the results. `results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key `n` mapped to value `n ** 2`. See comprehensions.

**vista de diccionario** Los objetos retornados por los métodos `dict.keys()`, `dict.values()`, y `dict.items()` son llamados vistas de diccionarios. Proveen una vista dinámica de las entradas de un diccionario, lo que significa que cuando el diccionario cambia, la vista refleja éstos cambios. Para forzar a la vista de diccionario a convertirse en una lista completa, use `list(dictview)`. Vea dict-views.

**docstring** Una cadena de caracteres literal que aparece como la primera expresión en una clase, función o módulo. Aunque es ignorada cuando se ejecuta, es reconocida por el compilador y puesta en el atributo `__doc__` de la clase, función o módulo comprendida. Como está disponible mediante introspección, es el lugar canónico para ubicar la documentación del objeto.

**tipado de pato** Un estilo de programación que no revisa el tipo del objeto para determinar si tiene la interfaz correcta; en vez de ello, el método o atributo es simplemente llamado o usado («Si se ve como un pato y grazna como un pato, debe ser un pato»). Enfatizando las interfaces en vez de hacerlo con los tipos específicos, un código bien diseñado

pues tener mayor flexibilidad permitiendo la sustitución polimórfica. El tipado de pato *duck-typing* evita usar pruebas llamando a `type()` o `isinstance()`. (Nota: si embargo, el tipado de pato puede ser complementado con *abstract base classes*. En su lugar, generalmente pregunta con `hasattr()` o *EAFP*.

**EAFP** Del inglés *Easier to ask for forgiveness than permission*, es más fácil pedir perdón que pedir permiso. Este estilo de codificación común en Python asume la existencia de claves o atributos válidos y atrapa las excepciones si esta suposición resulta falsa. Este estilo rápido y limpio está caracterizado por muchas sentencias `try` y `except`. Esta técnica contrasta con estilo *LBYL* usual en otros lenguajes como C.

**expresión** Una construcción sintáctica que puede ser evaluada, hasta dar un valor. En otras palabras, una expresión es una acumulación de elementos de expresión tales como literales, nombres, accesos a atributos, operadores o llamadas a funciones, todos ellos retornando valor. A diferencia de otros lenguajes, no toda la sintaxis del lenguaje son expresiones. También hay *statement*s que no pueden ser usadas como expresiones, como la `while`. Las asignaciones también son sentencias, no expresiones.

**módulo de extensión** Un módulo escrito en C o C++, usando la API para C de Python para interactuar con el núcleo y el código del usuario.

**f-string** Son llamadas *f-strings* las cadenas literales que usan el prefijo `'f'` o `'F'`, que es una abreviatura para formatted string literals. Vea también **PEP 498**.

**objeto archivo** Un objeto que expone una API orientada a archivos (con métodos como `read()` o `write()`) al objeto subyacente. Dependiendo de la forma en la que fue creado, un objeto archivo, puede mediar el acceso a un archivo real en el disco u otro tipo de dispositivo de almacenamiento o de comunicación (por ejemplo, entrada/salida estándar, búfer de memoria, sockets, pipes, etc.). Los objetos archivo son también denominados *objetos tipo archivo* o *flujos*.

Existen tres categorías de objetos archivo: crudos *raw archivos binarios*, con búfer *archivos binarios* y *archivos de texto*. Sus interfaces son definidas en el módulo `io`. La forma canónica de crear objetos archivo es usando la función `open()`.

**objetos tipo archivo** Un sinónimo de *file object*.

**buscador** Un objeto que trata de encontrar el *loader* para el módulo que está siendo importado.

Desde la versión 3.3 de Python, existen dos tipos de buscadores: *meta buscadores de ruta* para usar con `sys.meta_path`, y *buscadores de entradas de rutas* para usar con `sys.path_hooks`.

Vea **PEP 302**, **PEP 420** y **PEP 451** para mayores detalles.

**división entera** Una división matemática que se redondea hacia el entero menor más cercano. El operador de la división entera es `//`. Por ejemplo, la expresión `11 // 4` evalúa `2` a diferencia del `2.75` retornado por la verdadera división de números flotantes. Note que `(-11) // 4` es `-3` porque es `-2.75` redondeado *para abajo*. Ver **PEP 238**.

**función** Una serie de sentencias que retornan un valor al que las llama. También se le puede pasar cero o más *argumentos* los cuales pueden ser usados en la ejecución de la misma. Vea también *parameter*, *method*, y la sección function.

**anotación de función** Una *annotation* del parámetro de una función o un valor de retorno.

Las anotaciones de funciones son usadas frecuentemente para *indicadores de tipo*, por ejemplo, se espera que una función tome dos argumentos de clase `int` y también se espera que retorne dos valores `int`:

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

La sintaxis de las anotaciones de funciones son explicadas en la sección function.

Vea *variable annotation* y **PEP 484**, que describen esta funcionalidad.

**__future__** Un pseudo-módulo que los programadores pueden usar para habilitar nuevas capacidades del lenguaje que no son compatibles con el intérprete actual.

Al importar el módulo `__future__` y evaluar sus variables, puede verse cuándo las nuevas capacidades fueron agregadas por primera vez al lenguaje y cuando se quedaron establecidas por defecto:

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

**recolección de basura** El proceso de liberar la memoria de lo que ya no está en uso. Python realiza recolección de basura (*garbage collection*) llevando la cuenta de las referencias, y el recogedor de basura cíclico es capaz de detectar y romper las referencias cíclicas. El recogedor de basura puede ser controlado mediante el módulo `gc`.

**generador** Una función que retorna un *generator iterator*. Luce como una función normal excepto que contiene la expresión `yield` para producir series de valores utilizables en un bucle *for* o que pueden ser obtenidas una por una con la función `next()`.

Usualmente se refiere a una función generadora, pero puede referirse a un *iterador generador* en ciertos contextos. En aquellos casos en los que el significado no está claro, usar los términos completos evita la ambigüedad.

**iterador generador** Un objeto creado por una función *generator*.

Cada `yield` suspende temporalmente el procesamiento, recordando el estado de ejecución local (incluyendo las variables locales y las sentencias *try* pendientes). Cuando el «iterador generado» vuelve, retoma donde ha dejado, a diferencia de lo que ocurre con las funciones que comienzan nuevamente con cada invocación.

**expresión generadora** Una expresión que retorna un iterador. Luce como una expresión normal seguida por la cláusula `for` definiendo así una variable de bucle, un rango y una cláusula opcional `if`. La expresión combinada genera valores para la función contenedora:

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

**función genérica** Una función compuesta de muchas funciones que implementan la misma operación para diferentes tipos. Qué implementación deberá ser usada durante la llamada a la misma es determinado por el algoritmo de despacho.

Vea también la entrada de glosario *single dispatch*, el decorador `functools.singledispatch()`, y **PEP 443**.

**GIL** Vea *global interpreter lock*.

**bloqueo global del intérprete** Mecanismo empleado por el intérprete *CPython* para asegurar que sólo un hilo ejecute el *bytecode* Python por vez. Esto simplifica la implementación de CPython haciendo que el modelo de objetos (incluyendo algunos críticos como `dict`) están implícitamente a salvo de acceso concurrente. Bloqueando el intérprete completo se simplifica hacerlo multi-hilos, a costa de mucho del paralelismo ofrecido por las máquinas con múltiples procesadores.

Sin embargo, algunos módulos de extensión, tanto estándar como de terceros, están diseñados para liberar el GIL cuando se realizan tareas computacionalmente intensivas como la compresión o el *hashing*. Además, el GIL siempre es liberado cuando se hace entrada/salida.

Esfuerzos previos hechos para crear un intérprete «sin hilos» (uno que bloquee los datos compartidos con una granularidad mucho más fina) no han sido exitosos debido a que el rendimiento sufrió para el caso más común de un solo procesador. Se cree que superar este problema de rendimiento haría la implementación mucho más compleja y por tanto, más costosa de mantener.

**hash-based pyc** Un archivo cache de *bytecode* que usa el *hash* en vez de usar el tiempo de la última modificación del archivo fuente correspondiente para determinar su validez. Vea pyc-invalidation.

**hashable** Un objeto es *hashable* si tiene un valor de hash que nunca cambiará durante su tiempo de vida (necesita un método `__hash__()`), y puede ser comparado con otro objeto (necesita el método `__eq__()`). Los objetos hashables que se comparan iguales deben tener el mismo número hash.

Ser *hashable* hace a un objeto utilizable como clave de un diccionario y miembro de un set, porque éstas estructuras de datos usan los valores de hash internamente.

La mayoría de los objetos inmutables incorporados en Python son *hashables*; los contenedores mutables (como las listas o los diccionarios) no lo son; los contenedores inmutables (como tuplas y conjuntos *frozensets*) son *hashables* si sus elementos son *hashables* . Los objetos que son instancias de clases definidas por el usuario son *hashables* por defecto. Todos se comparan como desiguales (excepto consigo mismos), y su valor de hash está derivado de su función `id()`.

**IDLE** El entorno integrado de desarrollo de Python, o *Integrated Development Environment for Python*. IDLE es un editor básico y un entorno de intérprete que se incluye con la distribución estándar de Python.

**inmutable** Un objeto con un valor fijo. Los objetos inmutables son números, cadenas y tuplas. Éstos objetos no pueden ser alterados. Un nuevo objeto debe ser creado si un valor diferente ha de ser guardado. Juegan un rol importante en lugares donde es necesario un valor de hash constante, por ejemplo como claves de un diccionario.

**ruta de importación** Una lista de las ubicaciones (o *entradas de ruta*) que son revisadas por *path based finder* al importar módulos. Durante la importación, ésta lista de localizaciones usualmente viene de `sys.path`, pero para los subpaquetes también puede incluir al atributo `__path__` del paquete padre.

**importar** El proceso mediante el cual el código Python dentro de un módulo se hace alcanzable desde otro código Python en otro módulo.

**importador** Un objeto que buscan y lee un módulo; un objeto que es tanto *finder* como *loader*.

**interactivo** Python tiene un intérprete interactivo, lo que significa que puede ingresar sentencias y expresiones en el prompt del intérprete, ejecutarlos de inmediato y ver sus resultados. Sólo ejecute `python` sin argumentos (podría seleccionarlo desde el menú principal de su computadora). Es una forma muy potente de probar nuevas ideas o inspeccionar módulos y paquetes (recuerde `help(x)`).

**interpretado** Python es un lenguaje interpretado, a diferencia de uno compilado, a pesar de que la distinción puede ser difusa debido al compilador a *bytecode*. Esto significa que los archivos fuente pueden ser corridos directamente, sin crear explícitamente un ejecutable que es corrido luego. Los lenguajes interpretados típicamente tienen ciclos de desarrollo y depuración más cortos que los compilados, sin embargo sus programas suelen correr más lentamente. Vea también *interactive*.

**apagado del intérprete** Cuando se le solicita apagarse, el intérprete Python ingresa a un fase especial en la cual gradualmente libera todos los recursos reservados, como módulos y varias estructuras internas críticas. También hace varias llamadas al *recolector de basura*. Esto puede disparar la ejecución de código de destructores definidos por el usuario o *weakref callbacks*. El código ejecutado durante la fase de apagado puede encontrar varias excepciones debido a que los recursos que necesita pueden no funcionar más (ejemplos comunes son los módulos de bibliotecas o los artefactos de advertencias *warnings machinery*)

La principal razón para el apagado del intérpreter es que el módulo `__main__` o el script que estaba corriendo termine su ejecución.

**iterable** 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.

Los iterables pueden ser usados en el bucle `for` y en muchos otros sitios donde una secuencia es necesaria (`zip()`, `map()`, ...). Cuando un objeto iterable es pasado como argumento a la función incorporada `iter()`, retorna un iterador para el objeto. Este iterador pasa así el conjunto de valores. Cuando se usan iterables, normalmente no es necesario llamar a la función `iter()` o tratar con los objetos iteradores usted mismo. La sentencia `for` lo hace automáticamente por usted, creando un variable temporal sin nombre para mantener el iterador mientras dura el bucle. Vea también *iterator*, *sequence*, y *generator*.

**iterador** Un objeto que representa un flujo de datos. Llamadas repetidas al método `__next__()` del iterador (o al pasar la función incorporada `next()`) retorna ítems sucesivos del flujo. Cuando no hay más datos disponibles, una excepción `StopIteration` es disparada. En este momento, el objeto iterador está exhausto y cualquier llamada

posterior al método `__next__()` sólo dispara otra vez `StopIteration`. Los iteradores necesitan tener un método `__iter__()` que retorna el objeto iterador mismo así cada iterador es también un iterable y puede ser usado en casi todos los lugares donde los iterables son aceptados. Una excepción importante es el código que intenta múltiples pases de iteración. Un objeto contenedor (como la `list`) produce un nuevo iterador cada vez que pasa a una función `iter()` o se usa en un bucle `for`. Intentar ésto con un iterador simplemente retornaría el mismo objeto iterador exhausto usado en previas iteraciones, haciéndolo aparecer como un contenedor vacío.

Puede encontrar más información en typeiter.

**función clave** Una función clave o una función de colación es un invocable que retorna un valor usado para el ordenamiento o clasificación. Por ejemplo, `locale.strxfrm()` es usada para producir claves de ordenamiento que se adaptan a las convenciones específicas de ordenamiento de un *locale*.

Cierta cantidad de herramientas de Python aceptan funciones clave para controlar como los elementos son ordenados o agrupados. Incluyendo a `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, y `itertools.groupby()`.

Hay varias formas de crear una función clave. Por ejemplo, el método `str.lower()` puede servir como función clave para ordenamientos que no distingan mayúsculas de minúsculas. Como alternativa, una función clave puede ser realizada con una expresión `lambda` como `lambda r: (r[0], r[2])`. También, el módulo `operator` provee tres constructores de funciones clave: `attrgetter()`, `itemgetter()`, y `methodcaller()`. Vea en Sorting HOW TO ejemplos de cómo crear y usar funciones clave.

**argumento nombrado** Vea *argument*.

**lambda** Una función anónima de una línea consistente en un sola *expression* que es evaluada cuando la función es llamada. La sintaxis para crear una función lambda es `lambda [parameters]: expression`

**LBYL** Del inglés *Look before you leap*, «mira antes de saltar». Es un estilo de codificación que prueba explícitamente las condiciones previas antes de hacer llamadas o búsquedas. Este estilo contrasta con la manera *EAFP* y está caracterizado por la presencia de muchas sentencias `if`.

En entornos multi-hilos, el método LBYL tiene el riesgo de introducir condiciones de carrera entre los hilos que están «mirando» y los que están «saltando». Por ejemplo, el código, *if key in mapping: return mapping[key]*‘ puede fallar si otro hilo remueve *key* de *mapping* después del test, pero antes de retornar el valor. Este problema puede ser resuelto usando bloqueos o empleando el método EAFP.

**lista** Es una *sequence* Python incorporada. A pesar de su nombre es más similar a un arreglo en otros lenguajes que a una lista enlazada porque el acceso a los elementos es O(1).

**comprensión de listas** Una forma compacta de procesar todos o parte de los elementos en una secuencia y retornar una lista como resultado. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` genera una lista de cadenas conteniendo números hexadecimales (0x..) entre 0 y 255. La cláusula `if` es opcional. Si es omitida, todos los elementos en `range(256)` son procesados.

**cargador** Un objeto que carga un módulo. Debe definir el método llamado `load_module()`. Un cargador es normalmente retornados por un *finder*. Vea **PEP 302** para detalles y `importlib.abc.Loader` para una *abstract base class*.

**método mágico** Una manera informal de llamar a un *special method*.

**mapeado** Un objeto contenedor que permite recupero de claves arbitrarias y que implementa los métodos especificados en la `Mapping` o `MutableMapping` abstract base classes. Por ejemplo, `dict`, `collections.defaultdict`, `collections.OrderedDict` y `collections.Counter`.

**meta buscadores de ruta** Un *finder* retornado por una búsqueda de `sys.meta_path`. Los meta buscadores de ruta están relacionados a *buscadores de entradas de rutas*, pero son algo diferente.

Vea en `importlib.abc.MetaPathFinder` los métodos que los meta buscadores de ruta implementan.

**metaclase** La clase de una clase. Las definiciones de clases crean nombres de clase, un diccionario de clase, y una lista de clases base. Las metaclases son responsables de tomar estos tres argumentos y crear la clase. La mayoría de los

objetos de un lenguaje de programación orientado a objetos provienen de una implementación por defecto. Lo que hace a Python especial que es posible crear metaclases a medida. La mayoría de los usuario nunca necesitarán esta herramienta, pero cuando la necesidad surge, las metaclases pueden brindar soluciones poderosas y elegantes. Han sido usadas para *loggear* acceso de atributos, agregar seguridad a hilos, rastrear la creación de objetos, implementar *singletons*, y muchas otras tareas.

Más información hallará en metaclasses.

**método** Una función que es definida dentro del cuerpo de una clase. Si es llamada como un atributo de una instancia de otra clase, el método tomará el objeto instanciado como su primer *argument* (el cual es usualmente denominado *self*). Vea *function* y *nested scope*.

**orden de resolución de métodos** Orden de resolución de métodos es el orden en el cual una clase base es buscada por un miembro durante la búsqueda. Mire en The Python 2.3 Method Resolution Order los detalles del algoritmo usado por el intérprete Python desde la versión 2.3.

**módulo** Un objeto que sirve como unidad de organización del código Python. Los módulos tienen espacios de nombres conteniendo objetos Python arbitrarios. Los módulos son cargados en Python por el proceso de *importing*.

Vea también *package*.

**especificador de módulo** Un espacio de nombres que contiene la información relacionada a la importación usada al leer un módulo. Una instancia de `importlib.machinery.ModuleSpec`.

**MRO** Vea *method resolution order*.

**mutable** Los objetos mutables pueden cambiar su valor pero mantener su `id()`. Vea también *immutable*.

**tupla nombrada** La denominación «tupla nombrada» se aplica a cualquier tipo o clase que hereda de una tupla y cuyos elementos indexables son también accesibles usando atributos nombrados. Este tipo o clase puede tener además otras capacidades.

Varios tipos incorporados son tuplas nombradas, incluyendo los valores retornados por `time.localtime()` y `os.stat()`. Otro ejemplo es `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
```

Algunas tuplas nombradas con tipos incorporados (como en los ejemplo precedentes). También puede ser creada con una definición regular de clase que hereda de la clase `tuple` y que define campos nombrados. Una clase como esta puede ser hechas personalizadamente o puede ser creada con la función factoría `collections.namedtuple()`. Esta última técnica automáticamente brinda métodos adicionales que pueden no estar presentes en las tuplas nombradas personalizadas o incorporadas.

**espacio de nombres** El lugar donde la variable es almacenada. Los espacios de nombres son implementados como diccionarios. Hay espacio de nombre local, global, e incorporado así como espacios de nombres anidados en objetos (en métodos). Los espacios de nombres soportan modularidad previniendo conflictos de nombramiento. Por ejemplo, las funciones `builtins.open` y `os.open()` se distinguen por su espacio de nombres. Los espacios de nombres también ayuda a la legibilidad y mantenibilidad dejando claro qué módulo implementa una función. Por ejemplo, escribiendo `random.seed()` o `itertools.islice()` queda claro que éstas funciones están implementadas en los módulos `random` y `itertools`, respectivamente.

**paquete de espacios de nombres** Un **PEP 420** *package* que sirve sólo para contener subpaquetes. Los paquetes de espacios de nombres pueden no tener representación física, y específicamente se diferencian de los *regular package* porque no tienen un archivo `__init__.py`.

Vea también *module*.

**alcances anidados** La habilidad de referirse a una variable dentro de una definición encerrada. Por ejemplo, una función definida dentro de otra función puede referir a variables en la función externa. Note que los alcances anidados por defecto sólo funcionan para referencia y no para asignación. Las variables locales leen y escriben sólo en el alcance más interno. De manera semejante, las variables globales pueden leer y escribir en el espacio de nombres global. Con `nonlocal` se puede escribir en alcances exteriores.

**clase de nuevo estilo** Vieja denominación usada para el estilo de clases ahora empleado en todos los objetos de clase. En versiones más tempranas de Python, sólo las nuevas clases podían usar capacidades nuevas y versátiles de Python como `__slots__`, descriptores, propiedades, `__getattribute__()`, métodos de clase y métodos estáticos.

**objeto** Cualquier dato con estado (atributo o valor) y comportamiento definido (métodos). También es la más básica clase base para cualquier *new-style class*.

**paquete** Un *module* Python que puede contener submódulos o recursivamente, subpaquetes. Técnicamente, un paquete es un módulo Python con un atributo `__path__`.

Vea también *regular package* y *namespace package*.

**parámetro** Una entidad nombrada en una definición de una *function* (o método) que especifica un *argument* (o en algunos casos, varios argumentos) que la función puede aceptar. Existen cinco tipos de argumentos:

- *posicional o nombrado*: especifica un argumento que puede ser pasado tanto como *posicional* o como *nombrado*. Este es el tipo por defecto de parámetro, como *foo* y *bar* en el siguiente ejemplo:

```
def func(foo, bar=None): ...
```

- *sólo posicional*: especifica un argumento que puede ser pasado sólo por posición. Los parámetros sólo posicionales pueden ser definidos incluyendo un carácter `/` en la lista de parámetros de la función después de ellos, como *posonly1* y *posonly2* en el ejemplo que sigue:

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

- *sólo nombrado*: especifica un argumento que sólo puede ser pasado por nombre. Los parámetros sólo por nombre pueden ser definidos incluyendo un parámetro posicional de una sola variable o un simple `*` ` antes de ellos en la lista de parámetros en la definición de la función, como *kw_only1* y *kw_only2* en el ejemplo siguiente:

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

- *variable posicional*: especifica una secuencia arbitraria de argumentos posicionales que pueden ser brindados (además de cualquier argumento posicional aceptado por otros parámetros). Este parámetro puede ser definido anteponiendo al nombre del parámetro `*`, como a *args* en el siguiente ejemplo:

```
def func(*args, **kwargs): ...
```

- *variable nombrado*: especifica que arbitrariamente muchos argumentos nombrados pueden ser brindados (además de cualquier argumento nombrado ya aceptado por cualquier otro parámetro). Este parámetro puede ser definido anteponiendo al nombre del parámetro con `**`, como *kwargs* en el ejemplo precedente.

Los parámetros puede especificar tanto argumentos opcionales como requeridos, así como valores por defecto para algunos argumentos opcionales.

Vea también el glosario de *argument*, la pregunta respondida en la diferencia entre argumentos y parámetros, la clase `inspect.Parameter`, la sección function , y **PEP 362**.

**entrada de ruta** Una ubicación única en el *import path* que el *path based finder* consulta para encontrar los módulos a importar.

**buscador de entradas de ruta** Un *finder* retornado por un invocable en `sys.path_hooks` (esto es, un *path entry hook*) que sabe cómo localizar módulos dada una *path entry*.

Vea en `importlib.abc.PathEntryFinder` los métodos que los buscadores de entradas de ruta implementan.

**gancho a entrada de ruta** Un invocable en la lista `sys.path_hook` que retorna un *path entry finder* si éste sabe cómo encontrar módulos en un *path entry* específico.

**buscador basado en ruta** Uno de los *meta buscadores de ruta* por defecto que busca un *import path* para los módulos.

**objeto tipo ruta** Un objeto que representa una ruta del sistema de archivos. Un objeto tipo ruta puede ser tanto una `str` como un `bytes` representando una ruta, o un objeto que implementa el protocolo `os.PathLike`. Un objeto que soporta el protocolo `os.PathLike` puede ser convertido a ruta del sistema de archivo de clase `str` o `bytes` usando la función `os.fspath()`; `os.fsdecode()` `os.fsencode()` pueden emplearse para garantizar que retorne respectivamente `str` o `bytes`. Introducido por **PEP 519**.

**PEP** Propuesta de mejora de Python, del inglés *Python Enhancement Proposal*. Un PEP es un documento de diseño que brinda información a la comunidad Python, o describe una nueva capacidad para Python, sus procesos o entorno. Los PEPs deberían dar una especificación técnica concisa y una fundamentación para las capacidades propuestas.

Los PEPs tienen como propósito ser los mecanismos primarios para proponer nuevas y mayores capacidad, para recoger la opinión de la comunidad sobre un tema, y para documentar las decisiones de diseño que se han hecho en Python. El autor del PEP es el responsable de lograr consenso con la comunidad y documentar las opiniones disidentes.

Vea **PEP 1**.

**porción** Un conjunto de archivos en un único directorio (posiblemente guardo en un archivo comprimido *zip*) que contribuye a un espacio de nombres de paquete, como está definido en **PEP 420**.

**argumento posicional** Vea *argument*.

**API provisoria** Una API provisoria es aquella que deliberadamente fue excluida de las garantías de compatibilidad hacia atrás de la biblioteca estándar. Aunque no se esperan cambios fundamentales en dichas interfaces, como están marcadas como provisionales, los cambios incompatibles hacia atrás (incluso remover la misma interfaz) podrían ocurrir si los desarrolladores principales lo estiman. Estos cambios no se hacen gratuitamente – solo ocurrirán si fallas fundamentales y serias son descubiertas que no fueron vistas antes de la inclusión de la API.

Incluso para APIs provisorias, los cambios incompatibles hacia atrás son vistos como una «solución de último recurso» - se intentará todo para encontrar una solución compatible hacia atrás para los problemas identificados.

Este proceso permite que la biblioteca estándar continúe evolucionando con el tiempo, sin bloquearse por errores de diseño problemáticos por períodos extensos de tiempo. Vea **PEP 411** para más detalles.

**paquete provisorio** Vea *provisional API*.

**Python 3000** Apodo para la fecha de lanzamiento de Python 3.x (acuñada en un tiempo cuando llegar a la versión 3 era algo distante en el futuro.) También se lo abrevió como *Py3k*.

**Pythónico** Una idea o pieza de código que sigue ajustadamente la convenciones idiomáticas comunes del lenguaje Python, en vez de implementar código usando conceptos comunes a otros lenguajes. Por ejemplo, una convención común en Python es hacer bucles sobre todos los elementos de un iterable con la sentencia `for`. Muchos otros lenguajes no tienen este tipo de construcción, así que los que no están familiarizados con Python podrían usar contadores numéricos:

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

En contraste, un método Pythónico más limpio:

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

**nombre calificado** Un nombre con puntos mostrando la ruta desde el alcance global del módulo a la clase, función o método definido en dicho módulo, como se define en **PEP 3155**. Para las funciones o clases de más alto nivel, el nombre calificado es el igual al nombre del objeto:

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

Cuando es usado para referirse a los módulos, *nombre completamente calificado* significa la ruta con puntos completo al módulo, incluyendo cualquier paquete padre, por ejemplo, *email.mime.text*':

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

**contador de referencias** El número de referencias a un objeto. Cuando el contador de referencias de un objeto cae hasta cero, éste es desalojable. En conteo de referencias no suele ser visible en el código de Python, pero es un elemento clave para la implementación de *CPython*. El módulo `sys` define la `getrefcount()` que los programadores pueden emplear para retornar el conteo de referencias de un objeto en particular.

**paquete regular** Un *package* tradicional, como aquellos con un directorio conteniendo el archivo `__init__.py`.

Vea también *namespace package*.

**__slots__** Es una declaración dentro de una clase que ahorra memoria predeclarando espacio para las atributos de la instancia y eliminando diccionarios de la instancia. Aunque es popular, esta técnica es algo dificultosa de lograr correctamente y es mejor reservarla para los casos raros en los que existen grandes cantidades de instancias en aplicaciones con uso crítico de memoria.

**secuencia** Un *iterable* que logra un acceso eficiente a los elementos usando índices enteros a través del método especial `__getitem__()` y que define un método `__len__()` que retorna la longitud de la secuencia. Algunas de las secuencias incorporadas son `list`, `str`, `tuple`, y `bytes`. Observe que `dict` también soporta `__getitem__()` y `__len__()`, pero es considerada un mapeo más que una secuencia porque las búsquedas son por claves arbitraria *immutable* y no por enteros.

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

**set comprehension** A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See comprehensions.

**despacho único** Una forma de despacho de una *generic function* donde la implementación es elegida a partir del tipo de un sólo argumento.

**rebanada** Un objeto que contiene una porción de una *sequence*. Una rebanada es creada usando la notación de suscripto, `[]` con dos puntos entre los números cuando se ponen varios, como en `nombre_variable[1:3:5]`. La notación con corchete (suscrito) usa internamente objetos `slice`.

**método especial** Un método que es llamado implícitamente por Python cuando ejecuta ciertas operaciones en un tipo, como la adición. Estos métodos tienen nombres que comienzan y terminan con doble barra baja. Los métodos especiales están documentados en specialnames.

**sentencia** Una sentencia es parte de un conjunto (un «bloque» de código). Una sentencia tanto es una *expression* como alguna de las varias sintaxis usando una palabra clave, como `if`, `while` o `for`.

**codificación de texto** Un códec que codifica las cadenas Unicode a bytes.

**archivo de texto** Un *file object* capaz de leer y escribir objetos `str`. Frecuentemente, un archivo de texto también accede a un flujo de datos binario y maneja automáticamente el *text encoding*. Ejemplos de archivos de texto que son abiertos en modo texto (`'r'` o `'w'`), `sys.stdin`, `sys.stdout`, y las instancias de `io.StringIO`.

Vea también *binary file* por objeto de archivos capaces de leer y escribir *objeto tipo binario*.

**cadena con triple comilla** Una cadena que está enmarcada por tres instancias de comillas (») o apostrofes (“). Aunque no brindan ninguna funcionalidad que no está disponible usando cadenas con comillas simple, son útiles por varias razones. Permiten incluir comillas simples o dobles sin escapar dentro de las cadenas y pueden abarcar múltiples líneas sin el uso de caracteres de continuación, haciéndolas particularmente útiles para escribir docstrings.

**tipo** El tipo de un objeto Python determina qué tipo de objeto es; cada objeto tiene un tipo. El tipo de un objeto puede ser accedido por su atributo `__class__` o puede ser conseguido usando `type(obj)`.

**alias de tipos** Un sinónimo para un tipo, creado al asignar un tipo a un identificador.

Los alias de tipos son útiles para simplificar los *indicadores de tipo*. Por ejemplo:

```
from typing import List, Tuple

def remove_gray_shades(
        colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

podría ser más legible así:

```
from typing import List, Tuple

Color = Tuple[int, int, int]

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

Vea `typing` y **PEP 484**, que describen esta funcionalidad.

**indicador de tipo** Una *annotation* que especifica el tipo esperado para una variable, un atributo de clase, un parámetro para una función o un valor de retorno.

Los indicadores de tipo son opcionales y no son obligados por Python pero son útiles para las herramientas de análisis de tipos estático, y ayuda a las IDE en el completado del código y la refactorización.

Los indicadores de tipo de las variables globales, atributos de clase, y funciones, no de variables locales, pueden ser accedidos usando `typing.get_type_hints()`.

Vea `typing` y **PEP 484**, que describen esta funcionalidad.

**saltos de líneas universales** Una manera de interpretar flujos de texto en la cual son reconocidos como finales de línea todas siguientes formas: la convención de Unix para fin de línea `'\n'`, la convención de Windows `'\r\n'`, y la vieja convención de Macintosh `'\r'`. Vea **PEP 278** y **PEP 3116**, además de `bytes.splitlines()` para usos adicionales.

**anotación de variable** Una *annotation* de una variable o un atributo de clase.

Cuando se anota una variable o un atributo de clase, la asignación es opcional:

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

Las anotaciones de variables son frecuentemente usadas para *type hints*: por ejemplo, se espera que esta variable tenga valores de clase `int`:

```
count: int = 0
```

La sintaxis de la anotación de variables está explicada en la sección annassign.

Vea *function annotation*, **PEP 484** y **PEP 526**, los cuales describen esta funcionalidad.

**entorno virtual** Un entorno cooperativamente aislado de ejecución que permite a los usuarios de Python y a las aplicaciones instalar y actualizar paquetes de distribución de Python sin interferir con el comportamiento de otras aplicaciones de Python en el mismo sistema.

Vea también `venv`.

**máquina virtual** Una computadora definida enteramente por software. La máquina virtual de Python ejecuta el *bytecode* generado por el compilador de *bytecode*.

**Zen de Python** Un listado de los principios de diseño y la filosofía de Python que son útiles para entender y usar el lenguaje. El listado puede encontrarse ingresando «`import this`» en la consola interactiva.

APÉNDICE B

# Acerca de estos documentos

Estos documentos son generados por reStructuredText desarrollado por Sphinx, un procesador de documentos específicamente escrito para la documentación de Python.

El desarrollo de la documentación y su cadena de herramientas es un esfuerzo enteramente voluntario, al igual que Python. Si tu quieres contribuir, por favor revisa la página reporting-bugs para más información de cómo hacerlo. Los nuevos voluntarios son siempre bienvenidos!

Agradecemos a:

- Fred L. Drake, Jr., el creador original de la documentación del conjunto de herramientas de Python y escritor de gran parte del contenido;
- el proyecto Docutils para creación de reStructuredText y el juego de Utilidades de Documentación;
- Fredrik Lundh por su proyecto Referencia Alternativa de Python para la cual Sphinx tuvo muchas ideas.

## B.1 Contribuidores de la documentación de Python

Muchas personas han contribuido para el lenguaje de Python, la librería estándar de Python, y la documentación de Python. Revisa Misc/ACKS la distribución de Python para una lista parcial de contribuidores.

Es solamente con la aportación y contribuciones de la comunidad de Python que Python tiene tan fantástica documentación – Muchas gracias!

APÉNDICE C

# History and License

## C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see https://www.zope.org/). In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

| Release | Derived from | Year | Owner | GPL compatible? |
|---|---|---|---|---|
| 0.9.0 thru 1.2 | n/a | 1991-1995 | CWI | yes |
| 1.3 thru 1.5.2 | 1.2 | 1995-1999 | CNRI | yes |
| 1.6 | 1.5.2 | 2000 | CNRI | no |
| 2.0 | 1.6 | 2000 | BeOpen.com | no |
| 1.6.1 | 1.6 | 2001 | CNRI | no |
| 2.1 | 2.0+1.6.1 | 2001 | PSF | no |
| 2.0.1 | 2.0+1.6.1 | 2001 | PSF | yes |
| 2.1.1 | 2.1+2.0.1 | 2001 | PSF | yes |
| 2.1.2 | 2.1.1 | 2002 | PSF | yes |
| 2.1.3 | 2.1.2 | 2002 | PSF | yes |
| 2.2 and above | 2.1.1 | 2001-now | PSF | yes |

**Nota:** GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

## C.2 Terms and conditions for accessing or otherwise using Python

Python software and documentation are licensed under the *PSF License Agreement*.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Agreement and the *Zero-Clause BSD license*.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See *Licenses and Acknowledgements for Incorporated Software* for an incomplete list of these licenses.

### C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.8.18

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   3.8.18 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 3.8.18 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.8.18 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.8.18 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python
   3.8.18.

4. PSF is making Python 3.8.18 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION␣
↪OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT␣
↪THE
   USE OF PYTHON 3.8.18 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
```

```
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.8.18
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.8.18, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

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

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This␣
↪License
   Agreement does not grant permission to use PSF trademarks or trade name in␣
↪a
   trademark sense to endorse or promote products or services of Licensee, or␣
↪any
   third party.

8. By copying, installing or otherwise using Python 3.8.18, 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.
```

(continué en la próxima página)

(proviene de la página anterior)

```
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.
```

(continué en la próxima página)

(proviene de la página anterior)

```
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 3.8.18 DOCUMENTATION

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

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
```

(continué en la próxima página)

(proviene de la página anterior)

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

## C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

### C.3.1 Mersenne Twister

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

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

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

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

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

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

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

### C.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

### C.3.3 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

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

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

### C.3.4 Cookie management

The `http.cookies` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

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

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

### C.3.5 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

### C.3.6 UUencode and UUdecode functions

The `uu` module contains the following notice:

```
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 Remote Procedure Calls

The `xmlrpc.client` module contains the following notice:

```
    The XML-RPC client interface is

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

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

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

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

(continué en la próxima página)

(proviene de la página anterior)

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

### C.3.8 test_epoll

The `test_epoll` module contains the following notice:

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

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

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

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

### C.3.9 Select kqueue

The `select` module contains the following notice for the kqueue interface:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
```

(continué en la próxima página)

(proviene de la página anterior)

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

### C.3.10 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski" implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

```
<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 and dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from http://www.netlib.org/fp/. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/****************************************************************
 *
 * 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.
```

(continué en la próxima página)

(proviene de la página anterior)

```
*
*****************************************************************/
```

### C.3.12 OpenSSL

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   *
   * 1. Redistributions of source code must retain the above copyright
   *    notice, this list of conditions and the following disclaimer.
   *
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in
   *    the documentation and/or other materials provided with the
   *    distribution.
   *
   * 3. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
   *    acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
   *
```

(continué en la próxima página)

(proviene de la página anterior)

```
    * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
    * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
    * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
    * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
    * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
    * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
    * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
    * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
    * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
    * OF THE POSSIBILITY OF SUCH DAMAGE.
    * ====================================================================
    *
    * This product includes cryptographic software written by Eric Young
    * (eay@cryptsoft.com).  This product includes software written by Tim
    * Hudson (tjh@cryptsoft.com).
    *
    */

Original SSLeay License
-----------------------

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
    * All rights reserved.
    *
    * This package is an SSL implementation written
    * by Eric Young (eay@cryptsoft.com).
    * The implementation was written so as to conform with Netscapes SSL.
    *
    * This library is free for commercial and non-commercial use as long as
    * the following conditions are aheared to.  The following conditions
    * apply to all code found in this distribution, be it the RC4, RSA,
    * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
    * included with this distribution is covered by the same copyright terms
    * except that the holder is Tim Hudson (tjh@cryptsoft.com).
    *
    * Copyright remains Eric Young's, and as such any Copyright notices in
    * the code are not to be removed.
    * If this package is used in a product, Eric Young should be given attribution
    * as the author of the parts of the library used.
    * This can be in the form of a textual message at program startup or
    * in documentation (online or textual) provided with the package.
    *
    * Redistribution and use in source and binary forms, with or without
    * modification, are permitted provided that the following conditions
    * are met:
    * 1. Redistributions of source code must retain the copyright
    *    notice, this list of conditions and the following disclaimer.
    * 2. Redistributions in binary form must reproduce the above copyright
    *    notice, this list of conditions and the following disclaimer in the
    *    documentation and/or other materials provided with the distribution.
    * 3. All advertising materials mentioning features or use of this software
    *    must display the following acknowledgement:
    *    "This product includes cryptographic software written by
    *     Eric Young (eay@cryptsoft.com)"
    *    The word 'cryptographic' can be left out if the rouines from the library
```

(continué en la próxima página)

(proviene de la página anterior)

```
    *    being used are not cryptographic related :-).
    * 4. If you include any Windows specific code (or a derivative thereof) from
    *    the apps directory (application code) you must include an acknowledgement:
    *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
    *
    * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
    * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
    * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
    * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
    * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
    * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
    * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
    * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
    * SUCH DAMAGE.
    *
    * The licence and distribution terms for any publically available version or
    * derivative of this code cannot be changed.  i.e. this code cannot simply be
    * copied and put under another distribution licence
    * [including the GNU Public Licence.]
    */
```

### C.3.13 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

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

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

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

### C.3.14 libffi

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

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

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

The `zlib` extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

### C.3.16 cfuhash

The implementation of the hash table used by the `tracemalloc` is based on the cfuhash project:

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

This code is released under the BSD license:

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

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

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

### C.3.17 libmpdec

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

```
Copyright (c) 2008-2016 Stefan Krah. All rights reserved.

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

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

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
```

(continué en la próxima página)

(proviene de la página anterior)

```
THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

### C.3.18 W3C C14N test suite

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at https://www.w3.org/TR/xml-c14n2-testcases/ and is distributed under the 3-clause BSD license:

> Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang), All Rights Reserved.
>
> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> - Redistributions of works must retain the original copyright notice, this list of conditions and the following disclaimer.
> - Redistributions in binary form must reproduce the original copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
> - Neither the name of the W3C nor the names of its contributors may be used to endorse or promote products derived from this work without specific prior written permission.
>
> THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS «AS IS» AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

# APÉNDICE D

## Copyright



See *History and License* for complete license and permissions information.

# Índice

## No alfabético



## A



## B



## C



## D



## E



## F

## R



## S



## T



## V



## W



## Z