
The Python/C API

Versión 3.9.18

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

febrero 07, 2024

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

1	Introducción	3
1.1	Estándares de codificación	4
1.2	Archivos de cabecera (<i>Include</i>)	4
1.3	Macros útiles	5
1.4	Objetos, tipos y conteos de referencias	6
1.4.1	Conteo de Referencias	7
1.4.2	Tipos	10
1.5	Excepciones	10
1.6	Integración de Python	12
1.7	Depuración de compilaciones	13
2	Interfaz binaria de aplicación estable	15
3	La capa de muy alto nivel	17
4	Conteo de referencias	23
5	Manejo de excepciones	25
5.1	Impresión y limpieza	26
5.2	Lanzando excepciones	26
5.3	Emitir advertencias	29
5.4	Consultando el indicador de error	30
5.5	Manejo de señal	31
5.6	Clases de Excepción	32
5.7	Objetos Excepción	32
5.8	Objetos Unicode de Excepción	33
5.9	Control de Recursión	34
5.10	Excepciones Estándar	35
5.11	Categorías de advertencia estándar	37
6	Utilidades	39
6.1	Utilidades del sistema operativo	39
6.2	Funciones del Sistema	42
6.3	Control de procesos	44
6.4	Importando Módulos	44
6.5	Soporte de empaquetado (<i>marshalling</i>) de datos	48
6.6	Analizando argumentos y construyendo valores	49

6.6.1	Analizando argumentos	49
6.6.2	Construyendo valores	55
6.7	Conversión y formato de cadenas de caracteres	57
6.8	Reflexión	59
6.9	Registro de códec y funciones de soporte	60
6.9.1	API de búsqueda de códec	60
6.9.2	API de registro para controladores de errores de codificación Unicode	61
7	Capa de objetos abstractos	63
7.1	Protocolo de objeto	63
7.2	Protocolo de llamada	67
7.2.1	El protocolo <i>tp_call</i>	67
7.2.2	El protocolo <i>vectorcall</i>	67
7.2.3	API para invocar objetos	69
7.2.4	API de soporte de llamadas	72
7.3	Protocolo de números	72
7.4	Protocolo de secuencia	75
7.5	Protocolo de mapeo	77
7.6	Protocolo iterador	78
7.7	Protocolo Búfer	79
7.7.1	Estructura de búfer	79
7.7.2	Tipos de solicitud búfer	81
7.7.3	Arreglos complejos	83
7.7.4	Funciones relacionadas a búfer	84
7.8	Protocolo de búfer antiguo	85
8	Capa de objetos concretos	87
8.1	Objetos fundamentales	87
8.1.1	Objetos Tipos	87
8.1.2	El objeto <i>None</i>	91
8.2	Objetos numéricos	91
8.2.1	Objetos Enteros	91
8.2.2	Objetos Booleanos	94
8.2.3	Objetos de punto flotante	95
8.2.4	Objetos de números complejos	95
8.3	Objetos de secuencia	97
8.3.1	Objetos Bytes	97
8.3.2	Objetos de arreglos de bytes (<i>bytearrays</i>)	99
8.3.3	Objetos y códecs Unicode	100
8.3.4	Objetos Tuplas	121
8.3.5	Objetos de secuencia de estructura	122
8.3.6	Objetos lista	123
8.4	Objetos contenedor	125
8.4.1	Objetos Diccionarios	125
8.4.2	Objetos Conjunto	127
8.5	Objetos de función	129
8.5.1	Objetos función	129
8.5.2	Objetos de método de instancia	130
8.5.3	Objetos método	131
8.5.4	Objetos Celda	131
8.5.5	Objetos Código	132
8.6	Otros objetos	133
8.6.1	Objetos archivo	133
8.6.2	Objetos Modulo	134

8.6.3	Objetos iteradores	141
8.6.4	Objetos descriptores	141
8.6.5	Objeto rebanada (<i>slice</i>)	142
8.6.6	Objeto Elipsis	143
8.6.7	Objetos de vista de memoria (<i>MemoryView</i>)	143
8.6.8	Objetos de referencia débil	144
8.6.9	Cápsulas	145
8.6.10	Objetos Generadores	146
8.6.11	Objetos corrutina	147
8.6.12	Objetos de variables de contexto	147
8.6.13	Objetos <i>DateTime</i>	149
8.6.14	Objects for Type Hinting	152
9	Inicialización, Finalización e Hilos	153
9.1	Antes de la inicialización de Python	153
9.2	Variables de configuración global	154
9.3	Inicializando y finalizando el intérprete	156
9.4	Parámetros de todo el proceso	157
9.5	Estado del hilo y el bloqueo global del intérprete	160
9.5.1	Liberando el GIL del código de extensión	161
9.5.2	Hilos creados sin Python	161
9.5.3	Precauciones sobre <code>fork()</code>	162
9.5.4	API de alto nivel	162
9.5.5	API de bajo nivel	165
9.6	Soporte de subinterprete	167
9.6.1	Errores y advertencias	169
9.7	Notificaciones asincrónicas	169
9.8	Perfilado y Rastreo	170
9.9	Soporte avanzado del depurador	171
9.10	Soporte de almacenamiento local de hilo	172
9.10.1	API de almacenamiento específico de hilo (TSS, <i>Thread Specific Storage</i>)	172
9.10.2	API de almacenamiento local de hilos (TLS, <i>Thread Local Storage</i>)	173
10	Configuración de inicialización de Python	175
10.1	PyWideStringList	176
10.2	PyStatus	177
10.3	PyPreConfig	178
10.4	Preinicialización con PyPreConfig	179
10.5	PyConfig	180
10.6	Inicialización con PyConfig	185
10.7	Configuración aislada	186
10.8	Configuración de Python	187
10.9	Configuración de ruta	187
10.10	Py_RunMain()	189
10.11	Py_GetArgcArgv()	189
10.12	API Provisional Privada de Inicialización Multifásica	189
11	Gestión de la memoria	191
11.1	Visión general	191
11.2	Interfaz de memoria sin procesar	192
11.3	Interfaz de memoria	193
11.4	Asignadores de objetos	194
11.5	Asignadores de memoria predeterminados	195
11.6	Personalizar asignadores de memoria	196

11.7	El asignador pymalloc	197
11.7.1	Personalizar asignador de arena de pymalloc	198
11.8	tracemalloc C API	198
11.9	Ejemplos	198
12	Soporte de implementación de objetos	201
12.1	Asignación de objetos en el montículo	201
12.2	Estructuras de objetos comunes	202
12.2.1	Tipos objeto base y macros	202
12.2.2	Implementando funciones y métodos	203
12.2.3	Acceder a atributos de tipos de extensión	206
12.3	Objetos Tipo	208
12.3.1	Referencia rápida	208
12.3.2	Definición de PyTypeObject	213
12.3.3	Ranuras (<i>Slots</i>) PyObject	214
12.3.4	Ranuras PyVarObject	215
12.3.5	Ranuras PyTypeObject	215
12.3.6	Tipos Montículos (<i>Heap Types</i>)	233
12.4	Estructuras de Objetos de Números	233
12.5	Estructuras de Objetos Mapeo	235
12.6	Estructuras de objetos secuencia	236
12.7	Estructuras de Objetos Búfer	237
12.8	Estructuras de objetos asíncronos	238
12.9	Tipo Ranura <i>typedefs</i>	239
12.10	Ejemplos	240
12.11	Apoyo a la recolección de basura cíclica	242
13	Versiones de API y ABI	245
A	Glosario	247
B	Acerca de estos documentos	261
B.1	Contribuidores de la documentación de Python	261
C	Historia y Licencia	263
C.1	Historia del software	263
C.2	Términos y condiciones para acceder o usar Python	264
C.2.1	ACUERDO DE LICENCIA DE PSF PARA PYTHON lanzamiento 	264
C.2.2	ACUERDO DE LICENCIA DE BEOPEN.COM PARA PYTHON 2.0	265
C.2.3	ACUERDO DE LICENCIA CNRI PARA PYTHON 1.6.1	266
C.2.4	ACUERDO DE LICENCIA CWI PARA PYTHON 0.9.0 HASTA 1.2	267
C.2.5	LICENCIA BSD DE CLÁUSULA CERO PARA CÓDIGO EN EL PYTHON lanzamiento DOCUMENTACIÓN	268
C.3	Licencias y reconocimientos para software incorporado	268
C.3.1	Mersenne Twister	268
C.3.2	Sockets	269
C.3.3	Servicios de socket asincrónicos	270
C.3.4	Gestión de cookies	270
C.3.5	Seguimiento de ejecución	271
C.3.6	funciones UUencode y UUdecode	271
C.3.7	Llamadas a procedimientos remotos XML	272
C.3.8	test_epoll	272
C.3.9	Seleccionar kqueue	273
C.3.10	SipHash24	273
C.3.11	strtod y dtoa	274

C.3.12	OpenSSL	274
C.3.13	expat	277
C.3.14	libffi	277
C.3.15	zlib	278
C.3.16	cfuhash	278
C.3.17	libmpdec	279
C.3.18	Conjunto de pruebas W3C C14N	279
D	Derechos de autor	281
	Índice	283

Este manual documenta la API utilizada por los programadores de C y C ++ que desean escribir módulos de extensión o incorporar Python. Es un complemento de `extending-index`, que describe los principios generales de la escritura de extensión pero no documenta las funciones API en detalle.

La interfaz del programador de aplicaciones (API) con Python brinda a los programadores de C y C++ acceso al intérprete de Python en una variedad de niveles. La API es igualmente utilizable desde C++, pero por brevedad generalmente se conoce como la API Python/C. Hay dos razones fundamentalmente diferentes para usar la API Python/C. La primera razón es escribir *módulos de extensión* para propósitos específicos; Estos son módulos C que extienden el intérprete de Python. Este es probablemente el uso más común. La segunda razón es usar Python como componente en una aplicación más grande; Esta técnica se conoce generalmente como integración (*embedding*) Python en una aplicación.

Escribir un módulo de extensión es un proceso relativamente bien entendido, donde un enfoque de «libro de cocina» (*cookbook*) funciona bien. Hay varias herramientas que automatizan el proceso hasta cierto punto. Si bien las personas han integrado Python en otras aplicaciones desde su existencia temprana, el proceso de integrar Python es menos sencillo que escribir una extensión.

Muchas funciones API son útiles independientemente de si está integrando o extendiendo Python; Además, la mayoría de las aplicaciones que integran Python también necesitarán proporcionar una extensión personalizada, por lo que probablemente sea una buena idea familiarizarse con la escritura de una extensión antes de intentar integrar Python en una aplicación real.

1.1 Estándares de codificación

Si está escribiendo código C para su inclusión en CPython, **debe** seguir las pautas y estándares definidos en [PEP 7](#). Estas pautas se aplican independientemente de la versión de Python a la que esté contribuyendo. Seguir estas convenciones no es necesario para sus propios módulos de extensión de terceros, a menos que eventualmente espere contribuir con ellos a Python.

1.2 Archivos de cabecera (*Include*)

Todas las definiciones de función, tipo y macro necesarias para usar la API Python/C se incluyen en su código mediante la siguiente línea:

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

Esto implica la inclusión de los siguientes archivos de encabezado estándar: `<stdio.h>`, `<string.h>`, `<errno.h>`, `<limits.h>`, `<assert.h>` y `<stdlib.h>` (si está disponible).

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 [Analizando argumentos y construyendo valores](#) para obtener una descripción de este macro.

Todos los nombres visibles del usuario definidos por `Python.h` (excepto los definidos por los encabezados estándar incluidos) tienen uno de los prefijos `Py` o `_Py`. Los nombres que comienzan con `_Py` son para uso interno de la implementación de Python y no deben ser utilizados por escritores de extensiones. Los nombres de miembros de estructura no tienen un prefijo reservado.

Nota: El código de usuario nunca debe definir nombres que comiencen con `Py` o `_Py`. Esto confunde al lector y pone en peligro la portabilidad del código de usuario para futuras versiones de Python, que pueden definir nombres adicionales que comienzan con uno de estos prefijos.

Los archivos de encabezado generalmente se instalan con Python. En Unix, estos se encuentran en los directorios `prefix/include/pythonversion/` y `exec_prefix/include/pythonversion/`, donde `prefix` y `exec_prefix` están definidos por los parámetros correspondientes al programa de Python **configure** y `version` es `'%d.%d' % sys.version_info[:2]`. En Windows, los encabezados se instalan en `prefix/include`, donde `prefix` es el directorio de instalación especificado para el instalador.

Para incluir los encabezados, coloque ambos directorios (si son diferentes) en la ruta de búsqueda de su compilador para incluir. *No* coloque los directorios principales en la ruta de búsqueda y luego use `#include <pythonX.Y/Python.h>`; esto se romperá en las compilaciones multiplataforma ya que los encabezados independientes de la plataforma bajo `prefix` incluyen los encabezados específicos de la plataforma de `exec_prefix`.

Los usuarios de C++ deben tener en cuenta que aunque la API se define completamente usando C, los archivos de encabezado declaran correctamente que los puntos de entrada son `extern "C"`. Como resultado, no es necesario hacer nada especial para usar la API desde C++.

1.3 Macros útiles

Varias macros útiles se definen en los archivos de encabezado de Python. Muchos se definen más cerca de donde son útiles (por ejemplo `Py_RETURN_NONE`). Otros de una utilidad más general se definen aquí. Esto no es necesariamente una lista completa.

Py_UNREACHABLE()

Use esto cuando tenga una ruta de código a la que no se pueda acceder por diseño. Por ejemplo, en la cláusula `default`: en una declaración `switch` para la cual todos los valores posibles están cubiertos en declaraciones `case`. Use esto en lugares donde podría tener la tentación de poner una llamada `assert(0)` o `abort()`.

En el modo de lanzamiento, la macro ayuda al compilador a optimizar el código y evita una advertencia sobre el código inalcanzable. Por ejemplo, la macro se implementa con `__builtin_unreachable()` en GCC en modo de lanzamiento.

Un uso de `Py_UNREACHABLE()` es seguir una llamada a una función que nunca retorna pero que no está declarada `_Py_NO_RETURN`.

Si una ruta de código es un código muy poco probable pero se puede acceder en casos excepcionales, esta macro no debe utilizarse. Por ejemplo, en condiciones de poca memoria o si una llamada al sistema retorna un valor fuera del rango esperado. En este caso, es mejor informar el error a la persona que llama. Si no se puede informar del error a la persona que llama, se puede utilizar `Py_FatalError()`.

Nuevo en la versión 3.7.

Py_ABS(x)

Retorna el valor absoluto de `x`.

Nuevo en la versión 3.3.

Py_MIN(x, y)

Retorna el valor mínimo entre `x` e `y`.

Nuevo en la versión 3.3.

Py_MAX(x, y)

Retorna el valor máximo entre `x` e `y`.

Nuevo en la versión 3.3.

Py_STRINGIFY(x)

Convierte `x` en una cadena de caracteres C. Por ejemplo, `Py_STRINGIFY(123)` retorna `"123"`.

Nuevo en la versión 3.4.

Py_MEMBER_SIZE(type, member)

Retorna el tamaño de una estructura (`type`) `member` en bytes.

Nuevo en la versión 3.6.

Py_CHARMASK(c)

El argumento debe ser un carácter o un número entero en el rango `[-128, 127]` o `[0, 255]`. Este macro retorna la conversión `c` a un `unsigned char`.

Py_GETENV(s)

Al igual que `getenv(s)`, pero retorna `NULL` si: la opción `-E` se pasó en la línea de comando (es decir, si se establece `Py_IgnoreEnvironmentFlag`).

Py_UNUSED(arg)

Use esto para argumentos no utilizados en una definición de función para silenciar las advertencias del compilador. Ejemplo: `int func(int a, int Py_UNUSED(b)) {return a; }`.

Nuevo en la versión 3.4.

Py_DEPRECATED (version)

Use esto para declaraciones obsoletas. El macro debe colocarse antes del nombre del símbolo.

Ejemplo:

```
Py_DEPRECATED(3.8) PyAPI_FUNC(int) Py_OldFunction(void);
```

Distinto en la versión 3.8: Soporte para MSVC fue agregado.

PyDoc_STRVAR (name, str)

Crea una variable con el nombre `name` que se puede usar en *docstrings*. Si Python se construye sin *docstrings*, el valor estará vacío.

Utilice `PyDoc_STRVAR` para que los *docstrings* admitan la construcción de Python sin *docstrings*, como se especifica en [PEP 7](#).

Ejemplo:

```
PyDoc_STRVAR(pop_doc, "Remove and return the rightmost element.");

static PyMethodDef deque_methods[] = {
    // ...
    {"pop", (PyCFunction)deque_pop, METH_NOARGS, pop_doc},
    // ...
}
```

PyDoc_STR (str)

Crea un *docstring* para la cadena de caracteres de entrada dada o una cadena vacía si los *docstrings* están deshabilitados.

Utilice `PyDoc_STR` al especificar *docstrings* para admitir la construcción de Python sin *docstrings*, como se especifica en [PEP 7](#).

Ejemplo:

```
static PyMethodDef sqlite_row_methods[] = {
    {"keys", (PyCFunction)sqlite_row_keys, METH_NOARGS,
     PyDoc_STR("Returns the keys of the row.")},
    {NULL, NULL}
};
```

1.4 Objetos, tipos y conteos de referencias

La mayoría de las funciones de Python/C API tienen uno o más argumentos, así como un valor de retorno de tipo `PyObject*`. Este tipo es un puntero a un tipo de datos opaco que representa un objeto arbitrario de Python. Dado que todos los tipos de objetos Python son tratados de la misma manera por el lenguaje Python en la mayoría de las situaciones (por ejemplo, asignaciones, reglas de alcance y paso de argumentos), es apropiado que estén representados por un solo tipo C. Casi todos los objetos de Python viven en el montículo (*heap*): nunca declaras una variable automática o estática de tipo `PyObject`, solo se pueden declarar variables de puntero de tipo `PyObject*`. La única excepción son los objetos tipo; como nunca deben desasignarse, son típicamente objetos estáticos `PyTypeObject`.

Todos los objetos de Python (incluso los enteros de Python) tienen un tipo (*type*) y un conteo de referencia (*reference count*). El tipo de un objeto determina qué tipo de objeto es (por ejemplo, un número entero, una lista o una función definida por el usuario; hay muchos más como se explica en *types*). Para cada uno de los tipos conocidos hay un macro para verificar si un objeto es de ese tipo; por ejemplo, `PyList_Check(a)` es verdadero si (y solo si) el objeto al que apunta *a* es una lista de Python.

1.4.1 Conteo de Referencias

El conteo de referencia es importante porque las computadoras de hoy tienen un tamaño de memoria finito (y a menudo muy limitado); cuenta cuántos lugares diferentes hay los cuales tienen una referencia a un objeto. Tal lugar podría ser otro objeto, o una variable C global (o estática), o una variable local en alguna función C. Cuando el recuento de referencia de un objeto se convierte en cero, el objeto se desasigna. Si contiene referencias a otros objetos, su recuento de referencias se reduce. Esos otros objetos pueden ser desasignados a su vez, si esta disminución hace que su recuento de referencia sea cero, y así sucesivamente. (Hay un problema obvio con los objetos que se refieren entre sí aquí; por ahora, la solución es «no hagas eso»).

Los conteos de referencias siempre se manipulan explícitamente. La forma normal es usar el macro `Py_INCREF()` para incrementar el conteo de referencia de un objeto en uno, y `Py_DECREF()` para disminuirlo en uno. El macro `Py_DECREF()` es considerablemente más compleja que la `incref`, ya que debe verificar si el recuento de referencia se convierte en cero y luego hacer que se llame al desasignador (*deallocador*) del objeto. El desasignador es un puntero de función contenido en la estructura de tipo del objeto. El desasignador específico del tipo se encarga de disminuir los recuentos de referencia para otros objetos contenidos en el objeto si este es un tipo de objeto compuesto, como una lista, así como realizar cualquier finalización adicional que sea necesaria. No hay posibilidad de que el conteo de referencia se desborde; se utilizan al menos tantos bits para contener el recuento de referencia como ubicaciones de memoria distintas en la memoria virtual (suponiendo `sizeof(Py_ssize_t) >= sizeof(void*)`). Por lo tanto, el incremento del recuento de referencia es una operación simple.

No es necesario incrementar el conteo de referencia de un objeto para cada variable local que contiene un puntero a un objeto. En teoría, el conteo de referencia del objeto aumenta en uno cuando se hace que la variable apunte hacia él y disminuye en uno cuando la variable se sale del alcance. Sin embargo, estos dos se cancelan entre sí, por lo que al final el recuento de referencias no ha cambiado. La única razón real para usar el recuento de referencia es evitar que el objeto pierda su asignación mientras nuestra variable lo apunte. Si sabemos que hay al menos otra referencia al objeto que vive al menos tanto como nuestra variable, no hay necesidad de incrementar el recuento de referencias temporalmente. Una situación importante donde esto surge es en los objetos que se pasan como argumentos a las funciones de C en un módulo de extensión que se llama desde Python; El mecanismo de llamada garantiza mantener una referencia a cada argumento durante la duración de la llamada.

Sin embargo, una trampa común es extraer un objeto de una lista y mantenerlo por un tiempo sin incrementar su conteo de referencia. Es posible que alguna otra operación elimine el objeto de la lista, disminuya su conteo de referencias y posiblemente lo desasigne. El peligro real es que las operaciones de aspecto inocente pueden invocar código arbitrario de Python que podría hacer esto; hay una ruta de código que permite que el control vuelva al usuario desde `Py_DECREF()`, por lo que casi cualquier operación es potencialmente peligrosa.

Un enfoque seguro es utilizar siempre las operaciones genéricas (funciones cuyo nombre comienza con `PyObject_`, `PyNumber_`, `PySequence_` o `PyMapping_`). Estas operaciones siempre incrementan el recuento de referencia del objeto que retornan. Esto deja a la persona que llama con la responsabilidad de llamar `Py_DECREF()` cuando hayan terminado con el resultado; Esto pronto se convierte en una segunda naturaleza.

Detalles del conteo de referencia

El comportamiento del conteo de referencias de funciones en la API de Python/C se explica mejor en términos de *propiedad de las referencias*. La propiedad pertenece a referencias, nunca a objetos (los objetos no son propiedad: siempre se comparten). «Poseer una referencia» significa ser responsable de llamar a `Py_DECREF` cuando ya no se necesita la referencia. La propiedad también se puede transferir, lo que significa que el código que recibe la propiedad de la referencia se hace responsable de eventualmente decretarla llamando a `Py_DECREF()` o `Py_XDECREF()` cuando ya no es necesario — o transmitir esta responsabilidad (generalmente a la persona que llama). Cuando una función transfiere la propiedad de una referencia a su llamador, se dice que el que llama recibe una *nueva* referencia. Cuando no se transfiere ninguna propiedad, se dice que la persona que llama *toma prestada* la referencia. No es necesario hacer nada para obtener una referencia prestada.

Por el contrario, cuando una función de llamada pasa una referencia a un objeto, hay dos posibilidades: la función *roba* una referencia al objeto, o no lo hace. *Robar una referencia* significa que cuando pasa una referencia a una función, esa

función asume que ahora posee esa referencia, y usted ya no es responsable de ella.

Pocas funciones roban referencias; las dos excepciones notables son `PyList_SetItem()` y `PyTuple_SetItem()`, que roban una referencia al elemento (¡pero no a la tupla o lista en la que se coloca el elemento!). Estas funciones fueron diseñadas para robar una referencia debido a un idioma común para poblar una tupla o lista con objetos recién creados; por ejemplo, el código para crear la tupla `(1, 2, "tres")` podría verse así (olvidando el manejo de errores por el momento; una mejor manera de codificar esto se muestra a continuación):

```
PyObject *t;

t = PyTuple_New(3);
PyTuple_SetItem(t, 0, PyLong_FromLong(1L));
PyTuple_SetItem(t, 1, PyLong_FromLong(2L));
PyTuple_SetItem(t, 2, PyUnicode_FromString("three"));
```

Aquí `PyLong_FromLong()` retorna una nueva referencia que es inmediatamente robada por `PyTuple_SetItem()`. Cuando quiera seguir usando un objeto aunque se le robe la referencia, use `Py_INCREF()` para tomar otra referencia antes de llamar a la función de robo de referencias.

Por cierto, `PyTuple_SetItem()` es la *única* forma de establecer elementos de tupla; `PySequence_SetItem()` y `PyObject_SetItem()` se niegan a hacer esto ya que las tuplas son un tipo de datos inmutable. Solo debe usar `PyTuple_SetItem()` para las tuplas que está creando usted mismo.

El código equivalente para llenar una lista se puede escribir usando `PyList_New()` y `PyList_SetItem()`.

Sin embargo, en la práctica, rara vez utilizará estas formas de crear y completar una tupla o lista. Hay una función genérica, `Py_BuildValue()`, que puede crear los objetos más comunes a partir de valores C, dirigidos por una cadena de caracteres de formato (*format string*). Por ejemplo, los dos bloques de código anteriores podrían reemplazarse por lo siguiente (que también se ocupa de la comprobación de errores):

```
PyObject *tuple, *list;

tuple = Py_BuildValue("(iis)", 1, 2, "three");
list = Py_BuildValue("[iis]", 1, 2, "three");
```

Es mucho más común usar `PyObject_SetItem()` y amigos con elementos cuyas referencias solo está prestando, como argumentos que se pasaron a la función que está escribiendo. En ese caso, su comportamiento con respecto a los recuentos de referencias es mucho más sensato, ya que no tiene que incrementar un recuento de referencias para poder regalar una referencia («robarla»). Por ejemplo, esta función establece todos los elementos de una lista (en realidad, cualquier secuencia mutable) en un elemento dado:

```
int
set_all(PyObject *target, PyObject *item)
{
    Py_ssize_t i, n;

    n = PyObject_Length(target);
    if (n < 0)
        return -1;
    for (i = 0; i < n; i++) {
        PyObject *index = PyLong_FromSsize_t(i);
        if (!index)
            return -1;
        if (PyObject_SetItem(target, index, item) < 0) {
            Py_DECREF(index);
            return -1;
        }
        Py_DECREF(index);
    }
}
```

(continué en la próxima página)

(proviene de la página anterior)

```

    }
    return 0;
}

```

La situación es ligeramente diferente para los valores de retorno de la función. Si bien pasar una referencia a la mayoría de las funciones no cambia sus responsabilidades de propiedad para esa referencia, muchas funciones que retornan una referencia a un objeto le otorgan la propiedad de la referencia. La razón es simple: en muchos casos, el objeto retornado se crea sobre la marcha, y la referencia que obtiene es la única referencia al objeto. Por lo tanto, las funciones genéricas que retornan referencias de objeto, como `PyObject_GetItem()` y `PySequence_GetItem()`, siempre retornan una nueva referencia (la entidad que llama se convierte en el propietario de la referencia).

Es importante darse cuenta de que si posee una referencia retornada por una función depende de a qué función llame únicamente — *el plumaje* (el tipo del objeto pasado como argumento a la función) *no entra en él!* Por lo tanto, si extrae un elemento de una lista usando `PyList_GetItem()`, no posee la referencia — pero si obtiene el mismo elemento de la misma lista usando `PySequence_GetItem()` (que toma exactamente los mismos argumentos), usted posee una referencia al objeto retornado.

Aquí hay un ejemplo de cómo podría escribir una función que calcule la suma de los elementos en una lista de enteros; una vez usando `PyList_GetItem()`, y una vez usando `PySequence_GetItem()`.

```

long
sum_list(PyObject *list)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;

    n = PyList_Size(list);
    if (n < 0)
        return -1; /* Not a list */
    for (i = 0; i < n; i++) {
        item = PyList_GetItem(list, i); /* Can't fail */
        if (!PyLong_Check(item)) continue; /* Skip non-integers */
        value = PyLong_AsLong(item);
        if (value == -1 && PyErr_Occurred())
            /* Integer too big to fit in a C long, bail out */
            return -1;
        total += value;
    }
    return total;
}

```

```

long
sum_sequence(PyObject *sequence)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;
    n = PySequence_Length(sequence);
    if (n < 0)
        return -1; /* Has no length */
    for (i = 0; i < n; i++) {
        item = PySequence_GetItem(sequence, i);
        if (item == NULL)
            return -1; /* Not a sequence, or other failure */
        if (PyLong_Check(item)) {

```

(continué en la próxima página)

(proviene de la página anterior)

```

        value = PyLong_AsLong(item);
        Py_DECREF(item);
        if (value == -1 && PyErr_Occurred())
            /* Integer too big to fit in a C long, bail out */
            return -1;
        total += value;
    }
    else {
        Py_DECREF(item); /* Discard reference ownership */
    }
}
return total;
}

```

1.4.2 Tipos

Hay algunos otros tipos de datos que juegan un papel importante en la API de Python/C; la mayoría son tipos C simples como `int`, `long`, `double` y `char*`. Algunos tipos de estructura se usan para describir tablas estáticas que se usan para enumerar las funciones exportadas por un módulo o los atributos de datos de un nuevo tipo de objeto, y otro se usa para describir el valor de un número complejo. Estos serán discutidos junto con las funciones que los usan.

`Py_ssize_t`

A signed integral type such that `sizeof(Py_ssize_t) == sizeof(size_t)`. C99 doesn't define such a thing directly (`size_t` is an unsigned integral type). See [PEP 353](#) for details. `PY_SSIZE_T_MAX` is the largest positive value of type `Py_ssize_t`.

1.5 Excepciones

El programador de Python solo necesita lidiar con excepciones si se requiere un manejo específico de errores; las excepciones no manejadas se propagan automáticamente a la persona que llama, luego a la persona que llama, y así sucesivamente, hasta que llegan al intérprete de nivel superior, donde se informan al usuario acompañado de un seguimiento de pila (*stack traceback*).

Para los programadores de C, sin embargo, la comprobación de errores siempre tiene que ser explícita. Todas las funciones en la API Python/C pueden generar excepciones, a menos que se señale explícitamente en la documentación de una función. En general, cuando una función encuentra un error, establece una excepción, descarta cualquier referencia de objeto que posea y retorna un indicador de error. Si no se documenta lo contrario, este indicador es `NULL` o `-1`, dependiendo del tipo de retorno de la función. Algunas funciones retornan un resultado booleano verdadero/falso, con falso que indica un error. Muy pocas funciones no retornan ningún indicador de error explícito o tienen un valor de retorno ambiguo, y requieren pruebas explícitas de errores con `PyErr_Occurred()`. Estas excepciones siempre se documentan explícitamente.

El estado de excepción se mantiene en el almacenamiento por subproceso (esto es equivalente a usar el almacenamiento global en una aplicación sin subprocesos). Un subproceso puede estar en uno de dos estados: se ha producido una excepción o no. La función `PyErr_Occurred()` puede usarse para verificar esto: retorna una referencia prestada al objeto de tipo de excepción cuando se produce una excepción, y `NULL` de lo contrario. Hay una serie de funciones para establecer el estado de excepción: `PyErr_SetString()` es la función más común (aunque no la más general) para establecer el estado de excepción, y `PyErr_Clear()` borra la excepción estado.

El estado de excepción completo consta de tres objetos (todos los cuales pueden ser `NULL`): el tipo de excepción, el valor de excepción correspondiente y el rastreo. Estos tienen los mismos significados que el resultado de Python de `sys.exc_info()`; sin embargo, no son lo mismo: los objetos Python representan la última excepción manejada por una declaración de Python `try ... except`, mientras que el estado de excepción de nivel C solo existe mientras se está

pasando una excepción entre las funciones de C hasta que llega al bucle principal del intérprete de código de bytes (*bytecode*) de Python, que se encarga de transferirlo a `sys.exc_info()` y amigos.

Tenga en cuenta que a partir de Python 1.5, la forma preferida y segura de subprocesos para acceder al estado de excepción desde el código de Python es llamar a la función `sys.exc_info()`, que retorna el estado de excepción por subproceso para el código de Python. Además, la semántica de ambas formas de acceder al estado de excepción ha cambiado de modo que una función que detecta una excepción guardará y restaurará el estado de excepción de su hilo para preservar el estado de excepción de su llamador. Esto evita errores comunes en el código de manejo de excepciones causado por una función de aspecto inocente que sobrescribe la excepción que se maneja; También reduce la extensión de vida útil a menudo no deseada para los objetos a los que hacen referencia los marcos de pila en el rastreo.

Como principio general, una función que llama a otra función para realizar alguna tarea debe verificar si la función llamada generó una excepción y, de ser así, pasar el estado de excepción a quien la llama (*caller*). Debe descartar cualquier referencia de objeto que posea y retornar un indicador de error, pero *no* debe establecer otra excepción — que sobrescribirá la excepción que se acaba de generar y perderá información importante sobre la causa exacta del error.

Un ejemplo simple de detectar excepciones y pasarlas se muestra en el ejemplo `sum_sequence()` anterior. Sucede que este ejemplo no necesita limpiar ninguna referencia de propiedad cuando detecta un error. La siguiente función de ejemplo muestra algunos errores de limpieza. Primero, para recordar por qué le gusta Python, le mostramos el código Python equivalente:

```
def incr_item(dict, key):
    try:
        item = dict[key]
    except KeyError:
        item = 0
    dict[key] = item + 1
```

Aquí está el código C correspondiente, en todo su esplendor:

```
int
incr_item(PyObject *dict, PyObject *key)
{
    /* Objects all initialized to NULL for Py_XDECREF */
    PyObject *item = NULL, *const_one = NULL, *incremented_item = NULL;
    int rv = -1; /* Return value initialized to -1 (failure) */

    item = PyObject_GetItem(dict, key);
    if (item == NULL) {
        /* Handle KeyError only: */
        if (!PyErr_ExceptionMatches(PyExc_KeyError))
            goto error;

        /* Clear the error and use zero: */
        PyErr_Clear();
        item = PyLong_FromLong(0L);
        if (item == NULL)
            goto error;
    }
    const_one = PyLong_FromLong(1L);
    if (const_one == NULL)
        goto error;

    incremented_item = PyNumber_Add(item, const_one);
    if (incremented_item == NULL)
        goto error;

    if (PyObject_SetItem(dict, key, incremented_item) < 0)
```

(continué en la próxima página)

(proviene de la página anterior)

```

    goto error;
    rv = 0; /* Success */
    /* Continue with cleanup code */

error:
    /* Cleanup code, shared by success and failure path */

    /* Use Py_XDECREF() to ignore NULL references */
    Py_XDECREF(item);
    Py_XDECREF(const_one);
    Py_XDECREF(incremented_item);

    return rv; /* -1 for error, 0 for success */
}

```

Este ejemplo representa un uso aprobado de la declaración `goto` en C! Ilustra el uso de `PyErr_ExceptionMatches()` y `PyErr_Clear()` para manejar excepciones específicas, y el uso de `Py_XDECREF()` para eliminar referencias propias que pueden ser NULL (tenga en cuenta la 'X' en el nombre; `Py_DECREF()` se bloqueará cuando se enfrente con una referencia NULL). Es importante que las variables utilizadas para contener referencias propias se inicialicen en NULL para que esto funcione; Del mismo modo, el valor de retorno propuesto se inicializa a -1 (falla) y solo se establece en éxito después de que la última llamada realizada sea exitosa.

1.6 Integración de Python

La única tarea importante de la que solo tienen que preocuparse los integradores (a diferencia de los escritores de extensión) del intérprete de Python es la inicialización, y posiblemente la finalización, del intérprete de Python. La mayor parte de la funcionalidad del intérprete solo se puede usar después de que el intérprete se haya inicializado.

La función básica de inicialización es `Py_Initialize()`. Esto inicializa la tabla de módulos cargados y crea los módulos fundamentales `builtins`, `__main__`, y `sys`. También inicializa la ruta de búsqueda del módulo (`sys.path`).

`Py_Initialize()` no establece la «lista de argumentos de script» (`sys.argv`). Si el código de Python necesita esta variable que se ejecutará más adelante, debe establecerse explícitamente con una llamada a `PySys_SetArgvEx(argc, argv, updatepath)` después de la llamada a `Py_Initialize()`.

En la mayoría de los sistemas (en particular, en Unix y Windows, aunque los detalles son ligeramente diferentes), `Py_Initialize()` calcula la ruta de búsqueda del módulo basándose en su mejor estimación de la ubicación del ejecutable del intérprete de Python estándar, suponiendo que la biblioteca de Python se encuentra en una ubicación fija en relación con el ejecutable del intérprete de Python. En particular, busca un directorio llamado `lib/pythonX.Y` relativo al directorio padre donde se encuentra el ejecutable llamado `python` en la ruta de búsqueda del comando `shell` (la variable de entorno `PATH`).

Por ejemplo, si el ejecutable de Python se encuentra en `/usr/local/bin/python`, se supondrá que las bibliotecas están en `/usr/local/lib/pythonX.Y`. (De hecho, esta ruta particular también es la ubicación «alternativa», utilizada cuando no se encuentra un archivo ejecutable llamado `python` junto con `PATH`.) El usuario puede anular este comportamiento configurando la variable de entorno `PYTHONHOME`, o inserte directorios adicionales delante de la ruta estándar estableciendo `PYTHONPATH`.

La aplicación de integración puede dirigir la búsqueda llamando a `Py_SetProgramName(file)` antes llamando `Py_Initialize()`. Tenga en cuenta que `PYTHONHOME` todavía anula esto y `PYTHONPATH` todavía se inserta frente a la ruta estándar. Una aplicación que requiere un control total debe proporcionar su propia implementación de `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, y `Py_GetProgramFullPath()` (todo definido en `Modules/getpath.c`).

A veces, es deseable «no inicializar» Python. Por ejemplo, la aplicación puede querer comenzar de nuevo (hacer otra llamada a `Py_Initialize()`) o la aplicación simplemente se hace con el uso de Python y quiere liberar memoria asignada por Python. Esto se puede lograr llamando a `Py_FinalizeEx()`. La función `Py_IsInitialized()` retorna verdadero si Python se encuentra actualmente en el estado inicializado. Se proporciona más información sobre estas funciones en un capítulo posterior. Tenga en cuenta que `Py_FinalizeEx()` no libera toda la memoria asignada por el intérprete de Python, por ejemplo, la memoria asignada por los módulos de extensión actualmente no se puede liberar.

1.7 Depuración de compilaciones

Python se puede construir con varios macros para permitir verificaciones adicionales del intérprete y los módulos de extensión. Estas comprobaciones tienden a agregar una gran cantidad de sobrecarga al tiempo de ejecución, por lo que no están habilitadas de forma predeterminada.

Una lista completa de los diversos tipos de compilaciones de depuración se encuentra en el archivo `Misc/SpecialBuilds.txt` en la distribución fuente de Python. Hay compilaciones disponibles que admiten el rastreo de los conteos de referencia, la depuración del asignador de memoria o la creación de perfiles de bajo nivel del bucle principal del intérprete. Solo las compilaciones más utilizadas se describirán en el resto de esta sección.

Compilar el intérprete con el macro `Py_DEBUG` definido produce lo que generalmente se entiende por «una compilación de depuración» de Python. `Py_DEBUG` se habilita en la compilación de Unix agregando `--with-pydebug` al comando `./configure`. También está implícito en la presencia del macro no específico de Python `_DEBUG`. Cuando `Py_DEBUG` está habilitado en la compilación de Unix, la optimización del compilador está deshabilitada.

Además de la depuración del recuento de referencia que se describe a continuación, se realizan las siguientes verificaciones adicionales:

- Se agregan comprobaciones adicionales al asignador de objetos.
- Se agregan verificaciones adicionales al analizador y compilador.
- Las conversiones de tipos hacia abajo (*downcasting*) de tipos anchos a tipos estrechos se comprueban por pérdida de información.
- Se agregan varias aserciones al diccionario y se establecen implementaciones. Además, el objeto `set` adquiere un método `test_c_api()`.
- Las comprobaciones de cordura (*sanity checks*) de los argumentos de entrada se agregan a la creación del marco.
- El almacenamiento para *ints* se inicializa con un patrón no válido conocido para capturar referencias a dígitos no inicializados.
- El seguimiento de bajo nivel y la comprobación de excepciones adicionales se agregan a la máquina virtual en tiempo de ejecución.
- Se agregan verificaciones adicionales a la implementación de la arena de memoria.
- Se agrega depuración adicional al módulo de hilos.

Puede haber controles adicionales no mencionados aquí.

Definiendo `Py_TRACE_REFS` habilita el rastreo de referencias. Cuando se define, se mantiene una lista circular doblemente vinculada de objetos activos al agregar dos campos adicionales a cada `PyObject`. También se realiza un seguimiento de las asignaciones totales. Al salir, se imprimen todas las referencias existentes. (En modo interactivo, esto sucede después de cada declaración ejecutada por el intérprete). Implicado por `Py_DEBUG`.

Consulte `Misc/SpecialBuilds.txt` en la distribución fuente de Python para obtener información más detallada.

Interfaz binaria de aplicación estable

Tradicionalmente, la API en C de Python cambiará con cada lanzamiento. La mayoría de los cambios serán compatibles con la fuente, generalmente solo agregando API, en lugar de cambiar la API existente o eliminar la API (aunque algunas interfaces se eliminan después de ser desaprobadas primero).

Desafortunadamente, la compatibilidad API no se extiende a la compatibilidad binaria (el ABI). La razón es principalmente la evolución de las definiciones de estructura, donde la adición de un nuevo campo, o el cambio del tipo de campo, puede no romper la API, pero puede romper la ABI. Como consecuencia, los módulos de extensión deben volver a compilarse para cada versión de Python (aunque es posible una excepción en Unix cuando no se utiliza ninguna de las interfaces afectadas). Además, en Windows, los módulos de extensión se vinculan con un `pythonXY.dll` específico y deben recompilarse para vincularse con uno más nuevo.

Desde Python 3.2, se ha declarado un subconjunto de la API para garantizar un ABI estable. Los módulos de extensión que deseen utilizar esta API (llamada «API limitada») deben definir `PY_LIMITED_API`. Varios detalles del intérprete se ocultan del módulo de extensión; a cambio, se construye un módulo que funciona en cualquier versión 3.x ($x \geq 2$) sin recompilación.

En algunos casos, el ABI estable debe ampliarse con nuevas funciones. Los módulos de extensión que deseen utilizar estas nuevas API deben establecer `PY_LIMITED_API` en el valor `PY_VERSION_HEX` (ver [Versiones de API y ABI](#)) de la versión mínima de Python que desean admitir (por ejemplo, “0x03030000” para Python 3.3). Dichos módulos funcionarán en todas las versiones posteriores de Python, pero no se cargarán (debido a la falta de símbolos) en las versiones anteriores.

A partir de Python 3.2, el conjunto de funciones disponibles para la API limitada se documenta en [PEP 384](#). En la documentación de la API de C, los elementos de la API que no forman parte de la API limitada se marcan como «No forma parte de la API limitada».

La capa de muy alto nivel

Las funciones en este capítulo te permitirán ejecutar código fuente de Python desde un archivo o un búfer, pero no te permitirán interactuar de una manera detallada con el intérprete.

Varias de estas funciones aceptan un símbolo de inicio de la gramática como parámetro. Los símbolos de inicio disponibles son `Py_eval_input`, `Py_file_input`, y `Py_single_input`. Estos se describen siguiendo las funciones que los aceptan como parámetros.

Tenga en cuenta también que varias de estas funciones toman parámetros `FILE*`. Una cuestión particular que debe manejarse con cuidado es que la estructura `FILE` para diferentes bibliotecas en C puede ser diferente e incompatible. En Windows (al menos), es posible que las extensiones vinculadas dinámicamente utilicen diferentes bibliotecas, por lo que se debe tener cuidado de que los parámetros `FILE*` solo se pasen a estas funciones si es seguro que estaban creado por la misma biblioteca que está utilizando el tiempo de ejecución de Python.

int **Py_Main** (int *argc*, wchar_t ***argv*)

El programa principal para el intérprete estándar. Está disponible para programas que incorporan Python. Los parámetros *argc* y *argv* deben prepararse exactamente como los que se pasan a la función `main()` de un programa en C (convertido a `wchar_t` de acuerdo con la configuración regional del usuario). Es importante tener en cuenta que la lista de argumentos puede ser modificada (pero el contenido de las cadenas de caracteres señaladas por la lista de argumentos no lo es). El valor de retorno será 0 si el intérprete acaba normalmente (es decir, sin excepción), 1 si el intérprete acaba debido a una excepción, o 2 si la lista de parámetros no representa una línea de comando Python válida.

Tenga en cuenta que si se lanza un `SystemExit` no manejado, esta función no retornará 1, pero saldrá del proceso, siempre que `Py_InspectFlag` no esté configurado.

int **Py_BytesMain** (int *argc*, char ***argv*)

Similar a `Py_Main()` pero *argv* es un arreglo de cadenas de caracteres de bytes.

Nuevo en la versión 3.8.

int **PyRun_AnyFile** (FILE **fp*, const char **filename*)

Esta es una interfaz simplificada para `PyRun_AnyFileExFlags()` más abajo, dejando *closeit* establecido a 0 y *flags* establecido a NULL.

int **PyRun_AnyFileFlags** (FILE **fp*, const char **filename*, *PyCompilerFlags* **flags*)

Esta es una interfaz simplificada para `PyRun_AnyFileExFlags()` más abajo, dejando *closeit* establecido a 0.

int **PyRun_AnyFileEx** (FILE *fp, const char *filename, int closeit)

Esta es una interfaz simplificada para `PyRun_AnyFileExFlags()` más abajo, dejando `flags` establecido a NULL.

int **PyRun_AnyFileExFlags** (FILE *fp, const char *filename, int closeit, *PyCompilerFlags* *flags)

If `fp` refers to a file associated with an interactive device (console or terminal input or Unix pseudo-terminal), return the value of `PyRun_InteractiveLoop()`, otherwise return the result of `PyRun_SimpleFile()`. `filename` is decoded from the filesystem encoding (`sys.getfilesystemencoding()`). If `filename` is NULL, this function uses "???" as the filename. If `closeit` is true, the file is closed before `PyRun_SimpleFileExFlags()` returns.

int **PyRun_SimpleString** (const char *command)

Esta es una interfaz simplificada para `PyRun_SimpleStringFlags()` más abajo, dejando el argumento *PyCompilerFlags** establecido a NULL.

int **PyRun_SimpleStringFlags** (const char *command, *PyCompilerFlags* *flags)

Ejecuta el código fuente de Python desde `command` en el módulo `__main__` de acuerdo con el argumento `flags`. Si `__main__` aún no existe, se crea. Retorna 0 en caso de éxito o -1 si se produjo una excepción. Si hubo un error, no hay forma de obtener la información de excepción. Para el significado de `flags`, ver abajo.

Tenga en cuenta que si no se maneja de otro modo `SystemExit`, esta función no retornará -1, pero saldrá del proceso, siempre que `Py_InspectFlag` no esté configurado.

int **PyRun_SimpleFile** (FILE *fp, const char *filename)

Esta es una interfaz simplificada para `PyRun_SimpleStringFlags()` más abajo, dejando `closeit` establecido a 0 y `flags` establecido a NULL.

int **PyRun_SimpleFileEx** (FILE *fp, const char *filename, int closeit)

Esta es una interfaz simplificada para `PyRun_SimpleStringFlags()` más abajo, dejando `flags` establecido a NULL.

int **PyRun_SimpleFileExFlags** (FILE *fp, const char *filename, int closeit, *PyCompilerFlags* *flags)

Similar a `PyRun_SimpleStringFlags()`, pero el código fuente de Python se lee desde `fp` en lugar de una cadena de caracteres en memoria. `filename` debe ser el nombre del archivo, se decodifica a partir de la codificación del sistema de archivos (`sys.getfilesystemencoding()`). Si `closeit` es verdadero, el archivo se cierra antes de que `PyRun_SimpleFileExFlags` retorne.

Nota: En Windows, `fp` debe abrirse en modo binario (por ejemplo, `fopen(filename, "rb")`). De lo contrario, Python puede no manejar correctamente el archivo de script con la terminación de línea LF.

int **PyRun_InteractiveOne** (FILE *fp, const char *filename)

Esta es una interfaz simplificada para `PyRun_InteractiveOneFlags()` más abajo, dejando `flags` establecido a NULL.

int **PyRun_InteractiveOneFlags** (FILE *fp, const char *filename, *PyCompilerFlags* *flags)

Lee y ejecuta declaraciones de un archivo asociado con un dispositivo interactivo de acuerdo al argumento `flags`. Se le solicitará al usuario usando `sys.ps1` y `sys.ps2`. `filename` se decodifica a partir de la codificación del sistema de archivos (`sys.getfilesystemencoding()`).

Retorna 0 cuando la entrada se ejecuta con éxito, -1 si hubo una excepción, o un código de error del archivo `errcode.h` distribuido como parte de Python si hubo un error de análisis gramatical. (Tenga en cuenta que `errcode.h` no está incluido en `Python.h`, por lo que debe incluirse específicamente si es necesario).

int **PyRun_InteractiveLoop** (FILE *fp, const char *filename)

Esta es una interfaz simplificada para `PyRun_InteractiveLoopFlags()` más abajo, dejando `flags` establecido a NULL.

int PyRun_InteractiveLoopFlags (FILE *fp, const char *filename, PyCompilerFlags *flags)

Lee y ejecuta declaraciones de un archivo asociado con un dispositivo interactivo hasta llegar al EOF. Se le solicitará al usuario usando `sys.ps1` y `sys.ps2`. *filename* se decodifica a partir de la codificación del sistema de archivos (`sys.getfilesystemencoding()`). Retorna 0 en EOF o un número negativo en caso de falla.

int (*PyOS_InputHook) (void)

Se puede configurar para que apunte a una función con el prototipo `int func(void)`. Se llamará a la función cuando el indicador del intérprete de Python esté a punto de estar inactivo y espere la entrada del usuario desde el terminal. El valor de retorno es ignorado. Sobrescribiendo este enlace se puede utilizar para integrar la solicitud del intérprete con otros bucles de eventos, como se hace en `Modules/_tkinter.c` en el código fuente de Python.

char* (*PyOS_ReadlineFunctionPointer) (FILE *, FILE *, const char *)

Se puede configurar para que apunte a una función con el prototipo `char *func (FILE *stdin, FILE *stdout, char *prompt)`, sobrescribiendo la función predeterminada utilizada para leer una sola línea de entrada desde el intérprete. Se espera que la función genere la cadena de caracteres *prompt* si no es NULL, y luego lea una línea de entrada del archivo de entrada estándar proporcionado, retornando la cadena de caracteres resultante. Por ejemplo, el módulo `readline` establece este enlace para proporcionar funciones de edición de línea y finalización de tabulación.

El resultado debe ser una cadena de caracteres alocado por `PyMem_RawMalloc()` o `PyMem_RawRealloc()`, o NULL si ocurre un error.

Distinto en la versión 3.4: El resultado debe ser alocado por `PyMem_RawMalloc()` o `PyMem_RawRealloc()`, en vez de ser alocado por `PyMem_Malloc()` o `PyMem_Realloc()`.

struct _node* PyParser_SimpleParseString (const char *str, int start)

Esta es una interfaz simplificada para `PyParser_SimpleParseStringFlagsFilename()` más abajo, dejando *filename* establecido a NULL y *flags* establecido a 0.

Deprecated since version 3.9, will be removed in version 3.10.

struct _node* PyParser_SimpleParseStringFlags (const char *str, int start, int flags)

Esta es una interfaz simplificada para `PyParser_SimpleParseStringFlagsFilename()` más abajo, dejando *filename* establecido a NULL.

Deprecated since version 3.9, will be removed in version 3.10.

struct _node* PyParser_SimpleParseStringFlagsFilename (const char *str, const char *filename, int start, int flags)

Analiza gramaticalmente el código fuente de Python desde *str* usando el token de inicio *start* de acuerdo con el argumento *flags*. El resultado se puede usar para crear un objeto de código que se puede evaluar de manera eficiente. Esto es útil si un fragmento de código debe evaluarse muchas veces. *filename* se decodifica a partir de la codificación del sistema de archivos (`sys.getfilesystemencoding()`).

Deprecated since version 3.9, will be removed in version 3.10.

struct _node* PyParser_SimpleParseFile (FILE *fp, const char *filename, int start)

Esta es una interfaz simplificada para `PyParser_SimpleParseFileFlags()` más abajo, dejando *flags* establecido a 0.

Deprecated since version 3.9, will be removed in version 3.10.

struct _node* PyParser_SimpleParseFileFlags (FILE *fp, const char *filename, int start, int flags)

Similar a `PyParser_SimpleParseStringFlagsFilename()`, pero el código fuente de Python se lee desde *fp* en lugar de una cadena de caracteres en memoria.

Deprecated since version 3.9, will be removed in version 3.10.

PyObject* PyRun_String (const char *str, int start, PyObject *globals, PyObject *locals)

Return value: New reference. Esta es una interfaz simplificada para `PyRun_StringFlags()` más abajo, dejando *flags* establecido a NULL.

*PyObject** **PyRun_StringFlags** (const char *str, int start, *PyObject* *globals, *PyObject* *locals, *PyCompilerFlags* *flags)

Return value: New reference. Ejecuta el código fuente de Python desde *str* en el contexto especificado por los objetos *globals* y *locals* con los indicadores del compilador especificados por *flags*. *globals* debe ser un diccionario; *locals* puede ser cualquier objeto que implemente el protocolo de mapeo. El parámetro *start* especifica el token de inicio que se debe usar para analizar el código fuente.

Retorna el resultado de ejecutar el código como un objeto Python, o NULL” si se produjo una excepción.

*PyObject** **PyRun_File** (FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals)

Return value: New reference. Esta es una interfaz simplificada para *PyRun_FileExFlags()* más abajo, dejando *closeit* establecido a 0 y *flags* establecido a NULL.

*PyObject** **PyRun_FileEx** (FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, int closeit)

Return value: New reference. Esta es una interfaz simplificada para *PyRun_FileExFlags()* más abajo, dejando *flags* establecido a NULL.

*PyObject** **PyRun_FileFlags** (FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, *PyCompilerFlags* *flags)

Return value: New reference. Esta es una interfaz simplificada para *PyRun_FileExFlags()* más abajo, dejando *closeit* establecido a 0.

*PyObject** **PyRun_FileExFlags** (FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, int closeit, *PyCompilerFlags* *flags)

Return value: New reference. Similar a *PyRun_StringFlags()*, pero el código fuente de Python se lee desde *fp* en lugar de una cadena de caracteres en memoria. *filename* debe ser el nombre del archivo, se decodifica a partir de la codificación del sistema de archivos (*sys.getfilesystemencoding()*). Si *closeit* es verdadero, el archivo se cierra antes que *PyRun_FileExFlags()* retorne.

*PyObject** **Py_CompileString** (const char *str, const char *filename, int start)

Return value: New reference. Esta es una interfaz simplificada para *Py_CompileStringFlags()* más abajo, dejando *flags* establecido a NULL.

*PyObject** **Py_CompileStringFlags** (const char *str, const char *filename, int start, *PyCompilerFlags* *flags)

Return value: New reference. Esta es una interfaz simplificada para *Py_CompileStringExFlags()* más abajo, con *optimize* establecido a -1.

*PyObject** **Py_CompileStringObject** (const char *str, *PyObject* *filename, int start, *PyCompilerFlags* *flags, int optimize)

Return value: New reference. Analiza gramaticalmente y compila el código fuente de Python en *str*, retornando el objeto de código resultante. El token de inicio viene dado por *start*; esto se puede usar para restringir el código que se puede compilar y debe ser *Py_eval_input*, *Py_file_input*, o *Py_single_input*. El nombre de archivo especificado por *filename* se usa para construir el objeto de código y puede aparecer en *tracebacks* o mensajes de excepción *SyntaxError*. Esto retorna NULL” si el código no se puede analizar gramaticalmente o compilar.

El número entero *optimize* especifica el nivel de optimización del compilador; un valor de -1 selecciona el nivel de optimización del intérprete como se indica en las opciones -O. Los niveles explícitos son 0 (sin optimización; *__debug__* es verdadero), 1 (los *asserts* se eliminan, *__debug__* es falso) o 2 (los docstrings también se eliminan)

Nuevo en la versión 3.4.

*PyObject** **Py_CompileStringExFlags** (const char *str, const char *filename, int start, *PyCompilerFlags* *flags, int optimize)

Return value: New reference. Como *Py_CompileStringObject()*, pero *filename* es una cadena de caracteres de byte decodificado desde la codificación del sistema de archivos (*os.fsdecode()*).

Nuevo en la versión 3.2.

*PyObject** **PyEval_EvalCode** (*PyObject* *co, *PyObject* *globals, *PyObject* *locals)

Return value: New reference. Esta es una interfaz simplificada para `PyEval_EvalCodeEx()`, con solo el objeto de código y las variables globales y locales. Los otros argumentos están establecidos en NULL.

*PyObject** **PyEval_EvalCodeEx** (*PyObject* *co, *PyObject* *globals, *PyObject* *locals, *PyObject* *const *args, int argcount, *PyObject* *const *kws, int kwcount, *PyObject* *const *defs, int defcount, *PyObject* *kwdefs, *PyObject* *closure)

Return value: New reference. Evaluar un objeto de código precompilado, dado un entorno particular para su evaluación. Este entorno consta de un diccionario de variables globales, un objeto de mapeo de variables locales, arreglos de argumentos, palabras clave y valores predeterminados, un diccionario de valores predeterminados para argumentos *keyword-only* y una tupla de cierre de células.

PyFrameObject

La estructura en C de los objetos utilizados para describir objetos del marco. Los campos de este tipo están sujetos a cambios en cualquier momento.

*PyObject** **PyEval_EvalFrame** (*PyFrameObject* *f)

Return value: New reference. Evaluar un marco de ejecución. Esta es una interfaz simplificada para `PyEval_EvalFrameEx()`, para compatibilidad con versiones anteriores.

*PyObject** **PyEval_EvalFrameEx** (*PyFrameObject* *f, int throwflag)

Return value: New reference. Esta es la función principal sin barnizar de la interpretación de Python. El objeto de código asociado con el marco de ejecución del marco *f* se ejecuta, interpretando el código de bytes y ejecutando llamadas según sea necesario. El parámetro adicional *throwflag* se puede ignorar por lo general; si es verdadero, entonces se genera una excepción de inmediato; esto se usa para los métodos `throw()` de objetos generadores.

Distinto en la versión 3.4: Esta función ahora incluye una afirmación de depuración para ayudar a garantizar que no descarte silenciosamente una excepción activa.

int **PyEval_MergeCompilerFlags** (*PyCompilerFlags* *cf)

Esta función cambia los flags del marco de evaluación actual, y retorna verdad (*true*) en caso de éxito, falso (*false*) en caso de fallo.

int **Py_eval_input**

El símbolo de inicio de la gramática de Python para expresiones aisladas; para usar con `Py_CompileString()`.

int **Py_file_input**

El símbolo de inicio de la gramática de Python para secuencias de declaración leídas desde un archivo u otra fuente; para usar con `Py_CompileString()`. Este es el símbolo usado cuando se compile un código fuente en Python arbitrariamente largo.

int **Py_single_input**

El símbolo de inicio de la gramática de Python para una declaración única; para usar con `Py_CompileString()`. Este es el símbolo usado para el bucle interactivo del intérprete.

struct **PyCompilerFlags**

Esta es la estructura usada para contener los flags del compilador. En casos donde el código es sólo compilado, es pasado como `int flags`, y en casos donde el código es ejecutado, es pasado como `PyCompilerFlags *flags`. En este caso, `from __future__ import` puede modificar los *flags*.

Siempre y cuando `PyCompilerFlags *flags` es NULL, `cf_flags` es tratado como igual a 0, y cualquier modificación debido a `from __future__ import` es descartada.

int **cf_flags**

Flags del compilador.

int **cf_feature_version**

cf_feature_version es la versión menor de Python. Debe ser inicializado a `PY_MINOR_VERSION`.

El campo es ignorado por defecto, es usado si y solo si el flag `PyCF_ONLY_AST` está configurado en *cf_flags*.

Distinto en la versión 3.8: Agregado el campo *cf_feature_version*.

int **CO_FUTURE_DIVISION**

Este bit puede ser configurado en *flags* para causar que un operador de división / sea interpretado como una «división real» de acuerdo a **PEP 238**.

Conteo de referencias

Los macros de esta sección se utilizan para administrar conteos de referencia de objetos Python.

void **Py_INCREF** (*PyObject* **o*)

Incrementa el conteo de referencia para el objeto *o*. El objeto no debe ser NULL; si no está seguro de que no sea NULL, use *Py_XINCREF* ().

void **Py_XINCREF** (*PyObject* **o*)

Incrementa el conteo de referencia para el objeto *o*. El objeto puede ser NULL, en cuyo caso el macro no tiene efecto.

void **Py_DECREF** (*PyObject* **o*)

Disminuye el conteo de referencia para el objeto *o*. El objeto no debe ser NULL; si no está seguro de que no sea NULL, use *Py_XDECREF* (). Si el conteo de referencia llega a cero, se invoca la función de desasignación del tipo de objeto (que no debe ser NULL).

Advertencia: La función de desasignación puede hacer que se invoque un código arbitrario de Python (por ejemplo, cuando se desasigna una instancia de clase con un método `__del__()`). Si bien las excepciones en dicho código no se propagan, el código ejecutado tiene acceso libre a todas las variables globales de Python. Esto significa que cualquier objeto al que se pueda acceder desde una variable global debe estar en un estado coherente antes de invocar *Py_DECREF* (). Por ejemplo, el código para eliminar un objeto de una lista debe copiar una referencia al objeto eliminado en una variable temporal, actualizar la estructura de datos de la lista y luego llamar a *Py_DECREF* () para la variable temporal.

void **Py_XDECREF** (*PyObject* **o*)

Disminuye el conteo de referencia para el objeto *o*. El objeto puede ser NULL, en cuyo caso el macro no tiene efecto; de lo contrario, el efecto es el mismo que para *Py_DECREF* (), y se aplica la misma advertencia.

void **Py_CLEAR** (*PyObject* **o*)

Disminuye el conteo de referencia para el objeto *o*. El objeto puede ser NULL, en cuyo caso el macro no tiene efecto; de lo contrario, el efecto es el mismo que para *Py_DECREF* (), excepto que el argumento también se establece en NULL. La advertencia para *Py_DECREF* () no se aplica con respecto al objeto pasado porque el macro usa cuidadosamente una variable temporal y establece el argumento en NULL antes de disminuir su conteo de referencia.

Es una buena idea usar este macro siempre que disminuya el conteo de referencia de un objeto que pueda atravesarse durante la recolección de basura.

Las siguientes funciones son para la incorporación dinámica de Python en tiempo de ejecución: `Py_IncRef(PyObject *o)`, `Py_DecRef(PyObject *o)`. Simplemente son versiones de funciones exportadas de `Py_XINCREF()` y `Py_XDECREF()`, respectivamente.

Las siguientes funciones o macros son solo para uso dentro del núcleo del intérprete: `_Py_Dealloc()`, `_Py_ForgetReference()`, `_Py_NewReference()`, así como la variable global `_Py_RefTotal`.

Manejo de excepciones

Las funciones descritas en este capítulo le permitirán manejar y aumentar las excepciones de Python. Es importante comprender algunos de los conceptos básicos del manejo de excepciones de Python. Funciona de manera similar a la variable POSIX `errno`: hay un indicador global (por subproceso) del último error que ocurrió. La mayoría de las funciones de C API no borran esto en caso de éxito, pero lo configuran para indicar la causa del error en caso de falla. La mayoría de las funciones de C API también retornan un indicador de error, generalmente `NULL` si se supone que retornan un puntero, o `-1` si retornan un número entero (excepción: las funciones `PyArg_*()` retornan `1` para el éxito y `0` para el fracaso).

Concretamente, el indicador de error consta de tres punteros de objeto: el tipo de excepción, el valor de la excepción y el objeto de rastreo. Cualquiera de esos punteros puede ser `NULL` si no está configurado (aunque algunas combinaciones están prohibidas, por ejemplo, no puede tener un rastreo no `NULL` si el tipo de excepción es `NULL`).

Cuando una función debe fallar porque alguna función que llamó falló, generalmente no establece el indicador de error; la función que llamó ya lo configuró. Es responsable de manejar el error y borrar la excepción o regresar después de limpiar cualquier recurso que tenga (como referencias de objetos o asignaciones de memoria); debería *no* continuar normalmente si no está preparado para manejar el error. Si regresa debido a un error, es importante indicarle a la persona que llama que se ha establecido un error. Si el error no se maneja o se propaga cuidadosamente, es posible que las llamadas adicionales a la API de Python/C no se comporten como se espera y pueden fallar de manera misteriosa.

Nota: El indicador de error es **no** el resultado de `sys.exc_info()`. El primero corresponde a una excepción que aún no se detecta (y, por lo tanto, todavía se está propagando), mientras que el segundo retorna una excepción después de que se detecta (y, por lo tanto, ha dejado de propagarse).

5.1 Impresión y limpieza

void **PyErr_Clear** ()

Borra el indicador de error. Si el indicador de error no está configurado, no hay efecto.

void **PyErr_PrintEx** (int *set_sys_last_vars*)

Imprime un rastreo estándar en `sys.stderr` y borra el indicador de error. **A menos que** el error sea un Salida del sistema, en ese caso no se imprime ningún rastreo y el proceso de Python se cerrará con el código de error especificado por la instancia de Salida del sistema.

Llame a esta función **solo** cuando el indicador de error está configurado. De lo contrario, provocará un error fatal!

Si *set_sys_last_vars* no es cero, las variables `sys.last_type`, `sys.last_value` y `sys.last_traceback` se establecerán en el tipo, valor y rastreo de la excepción impresa, respectivamente.

void **PyErr_Print** ()

Alias para `PyErr_PrintEx(1)`.

void **PyErr_WriteUnraisable** (*PyObject* **obj*)

Llama `sys.unraisablehook()` utilizando la excepción actual y el argumento *obj*.

Esta función de utilidad imprime un mensaje de advertencia en `sys.stderr` cuando se ha establecido una excepción, pero es imposible que el intérprete la active. Se usa, por ejemplo, cuando ocurre una excepción en un método `__del__()`.

La función se llama con un solo argumento *obj* que identifica el contexto en el que ocurrió la excepción que no se evalúa. Si es posible, la repr *obj* se imprimirá en el mensaje de advertencia.

Se debe establecer una excepción al llamar a esta función.

5.2 Lanzando excepciones

Estas funciones lo ayudan a configurar el indicador de error del hilo actual. Por conveniencia, algunas de estas funciones siempre retornarán un puntero NULL para usar en una declaración `return`.

void **PyErr_SetString** (*PyObject* **type*, const char **message*)

This is the most common way to set the error indicator. The first argument specifies the exception type; it is normally one of the standard exceptions, e.g. `PyExc_RuntimeError`. You need not increment its reference count. The second argument is an error message; it is decoded from 'utf-8'.

void **PyErr_SetObject** (*PyObject* **type*, *PyObject* **value*)

Esta función es similar a `PyErr_SetString()` pero le permite especificar un objeto Python arbitrario para el «valor» de la excepción.

*PyObject** **PyErr_Format** (*PyObject* **exception*, const char **format*, ...)

Return value: Always NULL. Esta función establece el indicador de error y retorna NULL. *exception* debe ser una clase de excepción Python. El *format* y los parámetros posteriores ayudan a formatear el mensaje de error; tienen el mismo significado y valores que en `PyUnicode_FromFormat()`. *format* es una cadena de caracteres codificada en ASCII.

*PyObject** **PyErr_FormatV** (*PyObject* **exception*, const char **format*, va_list *vargs*)

Return value: Always NULL. Igual que `PyErr_Format()`, pero tomando un argumento *va_list* en lugar de un número variable de argumentos.

Nuevo en la versión 3.5.

void **PyErr_SetNone** (*PyObject* **type*)

Esta es una abreviatura de `PyErr_SetObject(type, Py_None)`.

int PyErr_BadArgument ()

Esta es una abreviatura de `PyErr_SetString(PyExc_TypeError, message)`, donde *message* indica que se invocó una operación incorporada con un argumento ilegal. Es principalmente para uso interno.

PyObject* PyErr_NoMemory ()

Return value: Always NULL. Esta es una abreviatura de `PyErr_SetNone(PyExc_MemoryError)`; retorna NULL para que una función de asignación de objetos pueda escribir `return PyErr_NoMemory()`; cuando se queda sin memoria.

PyObject* PyErr_SetFromErrno (PyObject *type)

Return value: Always NULL. Esta es una función conveniente para generar una excepción cuando una función de biblioteca C ha retornado un error y establece la variable C `errno`. Construye un objeto tupla cuyo primer elemento es el valor entero `errno` y cuyo segundo elemento es el mensaje de error correspondiente (obtenido de `strerror()`), y luego llama a `PyErr_SetObject(type, objeto)`. En Unix, cuando el valor `errno` es `EINTR`, que indica una llamada interrumpida del sistema, esto llama `PyErr_CheckSignals()`, y si eso establece el indicador de error, lo deja configurado a ese. La función siempre retorna NULL, por lo que una función envolvente alrededor de una llamada del sistema puede escribir `return PyErr_SetFromErrno(type)`; cuando la llamada del sistema retorna un error.

PyObject* PyErr_SetFromErrnoWithFilenameObject (PyObject *type, PyObject *filenameObject)

Return value: Always NULL. Similar a `PyErr_SetFromErrno()`, con el comportamiento adicional de que si *filenameObject* *no es "NULL", se pasa al constructor de *type* como tercer parámetro. En el caso de la excepción `OSError`, se utiliza para definir el atributo `filename` de la instancia de excepción.

PyObject* PyErr_SetFromErrnoWithFilenameObjects (PyObject *type, PyObject *filenameObject, PyObject *filenameObject2)

Return value: Always NULL. Similar a `PyErr_SetFromErrnoWithFilenameObject()`, pero toma un segundo objeto de nombre de archivo, para generar errores cuando falla una función que toma dos nombres de archivo.

Nuevo en la versión 3.4.

PyObject* PyErr_SetFromErrnoWithFilename (PyObject *type, const char *filename)

Return value: Always NULL. Similar a `PyErr_SetFromErrnoWithFilenameObject()`, pero el nombre del archivo se da como una cadena de caracteres de C. *filename* se decodifica a partir de la codificación del sistema de archivos (`os.fsdecode()`).

PyObject* PyErr_SetFromWindowsError (int ierr)

Return value: Always NULL. Esta es una función conveniente para subir `WindowsError`. Si se llama con *ierr* de 0, el código de error retornado por una llamada a `GetLastError()` se usa en su lugar. Llama a la función `Win32 FormatMessage()` para recuperar la descripción de Windows del código de error proporcionado por *ierr* o `GetLastError()`, luego construye un objeto de tupla cuyo primer elemento es el *ierr* valor y cuyo segundo elemento es el mensaje de error correspondiente (obtenido de `FormatMessage()`), y luego llama a `PyErr_SetObject(PyExc_WindowsError, objeto)`. Esta función siempre retorna NULL.

Disponibilidad: Windows.

PyObject* PyErr_SetExcFromWindowsError (PyObject *type, int ierr)

Return value: Always NULL. Similar a `PyErr_SetFromWindowsError()`, con un parámetro adicional que especifica el tipo de excepción que se generará.

Disponibilidad: Windows.

PyObject* PyErr_SetFromWindowsErrorWithFilename (int ierr, const char *filename)

Return value: Always NULL. Similar a `PyErr_SetFromWindowsErrorWithFilenameObject()`, pero el nombre del archivo se da como una cadena de caracteres de C. *filename* se decodifica a partir de la codificación del sistema de archivos (`os.fsdecode()`).

Disponibilidad: Windows.

*PyObject** **PyErr_SetExcFromWindowsErrWithFilenameObject** (*PyObject* *type, int ierr, *PyObject* *filename)

Return value: Always *NULL*. Similar a `PyErr_SetFromWindowsErrWithFilenameObject()`, con un parámetro adicional que especifica el tipo de excepción que se generará.

Disponibilidad: Windows.

*PyObject** **PyErr_SetExcFromWindowsErrWithFilenameObjects** (*PyObject* *type, int ierr, *PyObject* *filename, *PyObject* *filename2)

Return value: Always *NULL*. Similar a `PyErr_SetExcFromWindowsErrWithFilenameObject()`, pero acepta un segundo objeto de nombre de archivo.

Disponibilidad: Windows.

Nuevo en la versión 3.4.

*PyObject** **PyErr_SetExcFromWindowsErrWithFilename** (*PyObject* *type, int ierr, const char *filename)

Return value: Always *NULL*. Similar a `PyErr_SetFromWindowsErrWithFilename()`, con un parámetro adicional que especifica el tipo de excepción que se generará.

Disponibilidad: Windows.

*PyObject** **PyErr_SetImportError** (*PyObject* *msg, *PyObject* *name, *PyObject* *path)

Return value: Always *NULL*. Esta es una función conveniente para subir `ImportError`. *msg* se establecerá como la cadena de mensaje de la excepción. *name* y *path*, que pueden ser *NULL*, se establecerán como atributos respectivos `name` y `path` de `ImportError`.

Nuevo en la versión 3.3.

*PyObject** **PyErr_SetImportErrorSubclass** (*PyObject* *exception, *PyObject* *msg, *PyObject* *name, *PyObject* *path)

Return value: Always *NULL*. Al igual que `PyErr_SetImportError()` pero esta función permite especificar una subclase de `ImportError` para aumentar.

Nuevo en la versión 3.6.

void **PyErr_SyntaxLocationObject** (*PyObject* *filename, int lineno, int col_offset)

Establece información de archivo, línea y desplazamiento para la excepción actual. Si la excepción actual no es un `SyntaxError`, establece atributos adicionales, lo que hace que el sub sistema de impresión de excepciones piense que la excepción es `SyntaxError`.

Nuevo en la versión 3.4.

void **PyErr_SyntaxLocationEx** (const char *filename, int lineno, int col_offset)

Como `PyErr_SyntaxLocationObject()`, pero *filename* es una cadena de bytes decodificada a partir de la codificación del sistema de archivos (`os.fsdecode()`).

Nuevo en la versión 3.2.

void **PyErr_SyntaxLocation** (const char *filename, int lineno)

Like `PyErr_SyntaxLocationEx()`, but the *col_offset* parameter is omitted.

void **PyErr_BadInternalCall** ()

Esta es una abreviatura de `PyErr_SetString(PyExc_SystemError, message)`, donde *message* indica que se invocó una operación interna (por ejemplo, una función de Python/C API) con un argumento ilegal. Es principalmente para uso interno.

5.3 Emitir advertencias

Use estas funciones para emitir advertencias desde el código C. Reflejan funciones similares exportadas por el módulo Python `warnings`. Normalmente imprimen un mensaje de advertencia a `sys.stderr`; sin embargo, también es posible que el usuario haya especificado que las advertencias se conviertan en errores, y en ese caso generarán una excepción. También es posible que las funciones generen una excepción debido a un problema con la maquinaria de advertencia. El valor de retorno es 0 si no se genera una excepción, o -1 si se genera una excepción. (No es posible determinar si realmente se imprime un mensaje de advertencia, ni cuál es el motivo de la excepción; esto es intencional). Si se produce una excepción, la persona que llama debe hacer su manejo normal de excepciones (por ejemplo, referencias propiedad de `Py_DECREF()` y retornan un valor de error).

int **PyErr_WarnEx** (*PyObject* *category, const char *message, *Py_ssize_t* stack_level)

Emite un mensaje de advertencia. El argumento *category* es una categoría de advertencia (ver más abajo) o NULL; el argumento *message* es una cadena de caracteres codificada en UTF-8. *stack_level* es un número positivo que proporciona una cantidad de marcos de pila; la advertencia se emitirá desde la línea de código que se está ejecutando actualmente en ese marco de pila. Un *stack_level* de 1 es la función que llama `PyErr_WarnEx()`, 2 es la función por encima de eso, y así sucesivamente.

Las categorías de advertencia deben ser subclases de `PyExc_Warning`; `PyExc_Warning` es una subclase de `PyExc_Exception`; la categoría de advertencia predeterminada es `PyExc_RuntimeWarning`. Las categorías de advertencia estándar de Python están disponibles como variables globales cuyos nombres se enumeran en *Categorías de advertencia estándar*.

Para obtener información sobre el control de advertencia, consulte la documentación del módulo `warnings` y la opción `-W` en la documentación de la línea de comandos. No hay API de C para el control de advertencia.

int **PyErr_WarnExplicitObject** (*PyObject* *category, *PyObject* *message, *PyObject* *filename, int lineno, *PyObject* *module, *PyObject* *registry)

Issue a warning message with explicit control over all warning attributes. This is a straightforward wrapper around the Python function `warnings.warn_explicit()`; see there for more information. The *module* and *registry* arguments may be set to NULL to get the default effect described there.

Nuevo en la versión 3.4.

int **PyErr_WarnExplicit** (*PyObject* *category, const char *message, const char *filename, int lineno, const char *module, *PyObject* *registry)

Similar a `PyErr_WarnExplicitObject()` excepto que *message* y *module* son cadenas codificadas UTF-8, y *filename* se decodifica de la codificación del sistema de archivos (`os.fsdecode()`).

int **PyErr_WarnFormat** (*PyObject* *category, *Py_ssize_t* stack_level, const char *format, ...)

Función similar a `PyErr_WarnEx()`, pero usa `PyUnicode_FromFormat()` para formatear el mensaje de advertencia. *format* es una cadena de caracteres codificada en ASCII.

Nuevo en la versión 3.2.

int **PyErr_ResourceWarning** (*PyObject* *source, *Py_ssize_t* stack_level, const char *format, ...)

Función similar a `PyErr_WarnFormat()`, pero *category* es `ResourceWarning` y pasa *source* a `warnings.WarningMessage()`.

Nuevo en la versión 3.6.

5.4 Consultando el indicador de error

*PyObject** **PyErr_Occurred**()

Return value: Borrowed reference. Prueba si el indicador de error está configurado. Si se establece, retorna la excepción *type* (el primer argumento de la última llamada a una de las funciones `PyErr_Set*`() o `PyErr_Restore`()). Si no está configurado, retorna NULL. No posee una referencia al valor de retorno, por lo que no necesita usar `Py_DECREF`().

La persona que llama debe retener el GIL.

Nota: No compare el valor de retorno con una excepción específica; use `PyErr_ExceptionMatches`() en su lugar, como se muestra a continuación. (La comparación podría fallar fácilmente ya que la excepción puede ser una instancia en lugar de una clase, en el caso de una excepción de clase, o puede ser una subclase de la excepción esperada).

int **PyErr_ExceptionMatches**(*PyObject *exc*)

Equivalente a `PyErr_GivenExceptionMatches(PyErr_Occurred(), exc)`. Esto solo debería llamarse cuando se establece una excepción; se producirá una infracción de acceso a la memoria si no se ha producido ninguna excepción.

int **PyErr_GivenExceptionMatches**(*PyObject *given, PyObject *exc*)

Retorna verdadero si la excepción *dada* coincide con el tipo de excepción en *exc*. Si *exc* es un objeto de clase, esto también retorna verdadero cuando *dado* es una instancia de una subclase. Si *exc* es una tupla, se busca una coincidencia en todos los tipos de excepción en la tupla (y recursivamente en sub tuplas).

void **PyErr_Fetch**(*PyObject **ptype, PyObject **pvalue, PyObject **ptraceback*)

Recupere el indicador de error en tres variables cuyas direcciones se pasan. Si el indicador de error no está configurado, configure las tres variables en NULL. Si está configurado, se borrará y usted tendrá una referencia a cada objeto recuperado. El objeto de valor y rastreo puede ser NULL incluso cuando el objeto de tipo no lo es.

Nota: Normalmente, esta función solo la usa el código que necesita capturar excepciones o el código que necesita guardar y restaurar el indicador de error temporalmente, por ejemplo:

```
{
    PyObject *type, *value, *traceback;
    PyErr_Fetch(&type, &value, &traceback);

    /* ... code that might produce other errors ... */

    PyErr_Restore(type, value, traceback);
}
```

void **PyErr_Restore**(*PyObject *type, PyObject *value, PyObject *traceback*)

Establece el indicador de error de los tres objetos. Si el indicador de error ya está configurado, se borra primero. Si los objetos son NULL, el indicador de error se borra. No pase un tipo NULL y un valor o rastreo no NULL. El tipo de excepción debería ser una clase. No pase un tipo o valor de excepción no válido. (Violar estas reglas causará problemas sutiles más adelante). Esta llamada quita una referencia a cada objeto: debe tener una referencia a cada objeto antes de la llamada y después de la llamada ya no posee estas referencias. (Si no comprende esto, no use esta función. Se lo advertí).

Nota: Normalmente, esta función solo la usa el código que necesita guardar y restaurar el indicador de error temporalmente. Use `PyErr_Fetch`() para guardar el indicador de error actual.

void **PyErr_NormalizeException** (*PyObject **exc*, *PyObject **val*, *PyObject **tb*)

Bajo ciertas circunstancias, los valores retornados por *PyErr_Fetch()* a continuación pueden ser «no normalizados», lo que significa que **exc* es un objeto de clase pero **val* no es una instancia de la misma clase. Esta función se puede utilizar para crear instancias de la clase en ese caso. Si los valores ya están normalizados, no pasa nada. La normalización retrasada se implementa para mejorar el rendimiento.

Nota: Esta función *no* establece implícitamente el atributo `__traceback__` en el valor de excepción. Si se desea establecer el rastreo de manera adecuada, se necesita el siguiente fragmento adicional:

```
if (tb != NULL) {
    PyException_SetTraceback(val, tb);
}
```

void **PyErr_GetExcInfo** (*PyObject **ptype*, *PyObject **pvalue*, *PyObject **ptraceback*)

Recupere la información de excepción, como se conoce de `sys.exc_info()`. Esto se refiere a una excepción que *ya fue capturada*, no a una excepción que se planteó recientemente. Retorna nuevas referencias para los tres objetos, cualquiera de los cuales puede ser NULL. No modifica el estado de información de excepción.

Nota: Esta función normalmente no es utilizada por el código que quiere manejar excepciones. En cambio, se puede usar cuando el código necesita guardar y restaurar el estado de excepción temporalmente. Use *PyErr_SetExcInfo()* para restaurar o borrar el estado de excepción.

Nuevo en la versión 3.3.

void **PyErr_SetExcInfo** (*PyObject *type*, *PyObject *value*, *PyObject *traceback*)

Establezca la información de excepción, como se conoce de `sys.exc_info()`. Esto se refiere a una excepción que *ya fue capturada*, no a una excepción que se planteó recientemente. Esta función roba las referencias de los argumentos. Para borrar el estado de excepción, pase NULL para los tres argumentos. Para ver las reglas generales sobre los tres argumentos, consulte *PyErr_Restore()*.

Nota: Esta función normalmente no es utilizada por el código que quiere manejar excepciones. En cambio, se puede usar cuando el código necesita guardar y restaurar el estado de excepción temporalmente. Use *PyErr_GetExcInfo()* para leer el estado de excepción.

Nuevo en la versión 3.3.

5.5 Manejo de señal

int **PyErr_CheckSignals** ()

Esta función interactúa con el manejo de la señal de Python. Comprueba si se ha enviado una señal a los procesos y, de ser así, invoca el controlador de señal correspondiente. Si el módulo `signal` es compatible, esto puede invocar un controlador de señal escrito en Python. En todos los casos, el efecto predeterminado para SIGINT es aumentar la excepción `KeyboardInterrupt`. Si se produce una excepción, se establece el indicador de error y la función retorna -1; de lo contrario, la función retorna 0. El indicador de error puede o no borrarse si se configuró previamente.

void **PyErr_SetInterrupt** ()

Simule el efecto de la llegada de una señal SIGINT. La próxima vez se llama *PyErr_CheckSignals()*, se llamará al manejador de señal de Python para SIGINT.

Si `SIGINT` no es manejado por Python (se configuró en `signal.SIG_DFL` o `signal.SIG_IGN`), esta función no hace nada.

int PySignal_SetWakeupFd (int fd)

Esta función de utilidad especifica un descriptor de archivo en el que el número de señal se escribe como un solo byte cada vez que se recibe una señal. *fd* debe ser sin bloqueo. retorna el descriptor de archivo anterior.

El valor `-1` desactiva la función; Este es el estado inicial. Esto es equivalente a `signal.set_wakeup_fd()` en Python, pero sin verificación de errores. *fd* debe ser un descriptor de archivo válido. La función solo debe llamarse desde el hilo principal.

Distinto en la versión 3.5: En Windows, la función ahora también admite controladores de socket.

5.6 Clases de Excepción

*PyObject** **PyErr_NewException** (const char *name, *PyObject* *base, *PyObject* *dict)

Return value: New reference. Esta función de utilidad crea y retorna una nueva clase de excepción. El argumento *name* debe ser el nombre de la nueva excepción, una cadena de caracteres en C de la forma `module.classname`. Los argumentos *base* y *dict* son normalmente `NULL`. Esto crea un objeto de clase derivado de `Exception` (accesible en C como `PyExc_Exception`).

El atributo `__module__` de la nueva clase se establece en la primera parte (hasta el último punto) del argumento *name*, y el nombre de la clase se establece en la última parte (después del último punto). El argumento *base* se puede usar para especificar clases base alternativas; puede ser solo una clase o una tupla de clases. El argumento *dict* se puede usar para especificar un diccionario de variables de clase y métodos.

*PyObject** **PyErr_NewExceptionWithDoc** (const char *name, const char *doc, *PyObject* *base, *PyObject* *dict)

Return value: New reference. Igual que `PyErr_NewException()`, excepto que la nueva clase de excepción puede recibir fácilmente una cadena de documentación: si *doc* no es `NULL`, se utilizará como la cadena de documentación para la clase de excepción.

Nuevo en la versión 3.2.

5.7 Objetos Excepción

*PyObject** **PyException_GetTraceback** (*PyObject* *ex)

Return value: New reference. Retorna el rastreo asociado con la excepción como una nueva referencia, accesible desde Python a través de `__traceback__`. Si no hay un rastreo asociado, esto retorna `NULL`.

int PyException_SetTraceback (*PyObject* *ex, *PyObject* *tb)

Establezca el rastreo asociado con la excepción a *tb*. Use `Py_None` para borrarlo.

*PyObject** **PyException_GetContext** (*PyObject* *ex)

Return value: New reference. Retorna el contexto (otra instancia de excepción durante cuyo manejo *ex* se generó) asociado con la excepción como una nueva referencia, accesible desde Python a través de `__context__`. Si no hay un contexto asociado, esto retorna `NULL`.

void PyException_SetContext (*PyObject* *ex, *PyObject* *ctx)

Establece el contexto asociado con la excepción a *ctx*. Use `NULL` para borrarlo. No hay verificación de tipo para asegurarse de que *ctx* es una instancia de excepción. Esto roba una referencia a *ctx*.

*PyObject** **PyException_GetCause** (*PyObject* *ex)

Return value: New reference. Retorna la causa (ya sea una instancia de excepción, o `None`, establecida por `raise ... from ...`) asociada con la excepción como una nueva referencia, como accesible desde Python a través de `__cause__`.

void **PyException_SetCause** (*PyObject* *ex, *PyObject* *cause)

Establece la causa asociada con la excepción a *cause*. Use NULL para borrarlo. No hay verificación de tipo para asegurarse de que *cause* sea una instancia de excepción o None. Esto roba una referencia a *cause*.

`__suppress_context__` es implícitamente establecido en True por esta función.

5.8 Objetos Unicode de Excepción

Las siguientes funciones se utilizan para crear y modificar excepciones Unicode de C.

*PyObject** **PyUnicodeDecodeError_Create** (const char *encoding, const char *object, *Py_ssize_t* length, *Py_ssize_t* start, *Py_ssize_t* end, const char *reason)

Return value: New reference. Crea un objeto `UnicodeDecodeError` con los atributos *encoding*, *object*, *length*, *start*, *end* y *reason*. *encoding* y *reason* son cadenas codificadas UTF-8.

*PyObject** **PyUnicodeEncodeError_Create** (const char *encoding, const *Py_UNICODE* *object, *Py_ssize_t* length, *Py_ssize_t* start, *Py_ssize_t* end, const char *reason)

Return value: New reference. Crea un objeto `UnicodeEncodeError` con los atributos *encoding*, *object*, *length*, *start*, *end* y *reason*. *encoding* y *reason* son cadenas codificadas UTF-8.

Obsoleto desde la versión 3.3: 3.11

`Py_UNICODE` está obsoleto desde Python 3.3. Migre por favor a `PyObject_CallFunction (PyExc_UnicodeEncodeError, "sOnns", ...)`.

*PyObject** **PyUnicodeTranslateError_Create** (const *Py_UNICODE* *object, *Py_ssize_t* length, *Py_ssize_t* start, *Py_ssize_t* end, const char *reason)

Return value: New reference. Crea un objeto `UnicodeTranslateError` con los atributos *encoding*, *object*, *length*, *start*, *end* y *reason*. *encoding* y *reason* son cadenas codificadas UTF-8.

Obsoleto desde la versión 3.3: 3.11

`Py_UNICODE` está obsoleto desde Python 3.3. Migre por favor a `PyObject_CallFunction (PyExc_UnicodeTranslateError, "sOnns", ...)`.

*PyObject** **PyUnicodeDecodeError_GetEncoding** (*PyObject* *exc)

*PyObject** **PyUnicodeEncodeError_GetEncoding** (*PyObject* *exc)

Return value: New reference. Retorna el atributo *encoding* del objeto de excepción dado.

*PyObject** **PyUnicodeDecodeError_GetObject** (*PyObject* *exc)

*PyObject** **PyUnicodeEncodeError_GetObject** (*PyObject* *exc)

*PyObject** **PyUnicodeTranslateError_GetObject** (*PyObject* *exc)

Return value: New reference. Retorna el atributo *object* del objeto de excepción dado.

int **PyUnicodeDecodeError_GetStart** (*PyObject* *exc, *Py_ssize_t* *start)

int **PyUnicodeEncodeError_GetStart** (*PyObject* *exc, *Py_ssize_t* *start)

int **PyUnicodeTranslateError_GetStart** (*PyObject* *exc, *Py_ssize_t* *start)

Obtiene el atributo *start* del objeto de excepción dado y lo coloca en **start*. *start* no debe ser NULL. retorna 0 en caso de éxito, -1 en caso de error.

int **PyUnicodeDecodeError_SetStart** (*PyObject* *exc, *Py_ssize_t* start)

int **PyUnicodeEncodeError_SetStart** (*PyObject* *exc, *Py_ssize_t* start)

int **PyUnicodeTranslateError_SetStart** (*PyObject* *exc, *Py_ssize_t* start)

Establece el atributo *start* del objeto de excepción dado en *start*. Retorna 0 en caso de éxito, -1 en caso de error.

int **PyUnicodeDecodeError_GetEnd** (*PyObject* *exc, *Py_ssize_t* *end)

int **PyUnicodeEncodeError_GetEnd** (*PyObject* *exc, *Py_ssize_t* *end)

int **PyUnicodeTranslateError_GetEnd** (*PyObject* *exc, *Py_ssize_t* *end)

Obtiene el atributo *end* del objeto de excepción dado y lo coloca en *end. end no debe ser NULL. retorna 0 en caso de éxito, -1 en caso de error.

int **PyUnicodeDecodeError_SetEnd** (*PyObject* *exc, *Py_ssize_t* end)

int **PyUnicodeEncodeError_SetEnd** (*PyObject* *exc, *Py_ssize_t* end)

int **PyUnicodeTranslateError_SetEnd** (*PyObject* *exc, *Py_ssize_t* end)

Establece el atributo *end* del objeto de excepción dado en end. Retorna 0 en caso de éxito, -1 en caso de error.

*PyObject** **PyUnicodeDecodeError_GetReason** (*PyObject* *exc)

*PyObject** **PyUnicodeEncodeError_GetReason** (*PyObject* *exc)

*PyObject** **PyUnicodeTranslateError_GetReason** (*PyObject* *exc)

Return value: New reference. Retorna el atributo *reason* del objeto de excepción dado.

int **PyUnicodeDecodeError_SetReason** (*PyObject* *exc, const char *reason)

int **PyUnicodeEncodeError_SetReason** (*PyObject* *exc, const char *reason)

int **PyUnicodeTranslateError_SetReason** (*PyObject* *exc, const char *reason)

Establece el atributo *reason* del objeto de excepción dado en reason. Retorna 0 en caso de éxito, -1 en caso de error.

5.9 Control de Recursión

Estas dos funciones proporcionan una forma de realizar llamadas recursivas seguras en el nivel C, tanto en el núcleo como en los módulos de extensión. Son necesarios si el código recursivo no invoca necesariamente el código Python (que rastrea su profundidad de recursión automáticamente). Tampoco son necesarios para las implementaciones de *tp_call* porque *call protocol* se encarga del manejo de la recursividad.

int **Py_EnterRecursiveCall** (const char *where)

Marca un punto donde una llamada recursiva de nivel C está a punto de realizarse.

Si `USE_STACKCHECK` está definido, esta función verifica si la pila del SO se desbordó usando `PyOS_CheckStack()`. En este caso, establece un `MemoryError` y retorna un valor distinto de cero.

La función verifica si se alcanza el límite de recursión. Si este es el caso, se establece a `RecursionError` y se retorna un valor distinto de cero. De lo contrario, se retorna cero.

where debería ser una cadena de caracteres codificada en UTF-8 como "en la comprobación de instancia" para concatenarse con el mensaje `RecursionError` causado por el límite de profundidad de recursión.

Distinto en la versión 3.9: Esta función ahora también está disponible en la API limitada.

void **Py_LeaveRecursiveCall** (void)

Termina una `Py_EnterRecursiveCall()`. Se debe llamar una vez por cada invocación exitosa de `Py_EnterRecursiveCall()`.

Distinto en la versión 3.9: Esta función ahora también está disponible en la API limitada.

La implementación adecuada de `tp_repr` para los tipos de contenedor requiere un manejo de recursión especial. Además de proteger la pila, `tp_repr` también necesita rastrear objetos para evitar ciclos. Las siguientes dos funciones facilitan esta funcionalidad. Efectivamente, estos son los C equivalentes a `reprlib.recursive_repr()`.

int **Py_ReprEnter** (*PyObject* *object)

Llamado al comienzo de la implementación `tp_repr` para detectar ciclos.

Si el objeto ya ha sido procesado, la función retorna un entero positivo. En ese caso, la implementación `tp_repr` debería retornar un objeto de cadena que indique un ciclo. Como ejemplos, los objetos `dict` retornan `{...}` y los objetos `list` retornan `[...]`.

La función retornará un entero negativo si se alcanza el límite de recursión. En ese caso, la implementación `tp_repr` normalmente debería retornar NULL.

De lo contrario, la función retorna cero y la implementación `tp_repr` puede continuar normalmente.

void **Py_ReprLeave** (*PyObject *object*)

Termina a `Py_ReprEnter()`. Se debe llamar una vez por cada invocación de `Py_ReprEnter()` que retorna cero.

5.10 Excepciones Estándar

Todas las excepciones estándar de Python están disponibles como variables globales cuyos nombres son `PyExc_` seguidos del nombre de excepción de Python. Estos tienen el tipo `PyObject*`; todos son objetos de clase. Para completar, aquí están todas las variables:

Nombre en C	Nombre en Python	Notas
<code>PyExc_BaseException</code>	<code>BaseException</code>	1
<code>PyExc_Exception</code>	<code>Exception</code>	1
<code>PyExc_ArithmeticError</code>	<code>ArithmeticError</code>	1
<code>PyExc_AssertionError</code>	<code>AssertionError</code>	
<code>PyExc_AttributeError</code>	<code>AttributeError</code>	
<code>PyExc_BlockingIOError</code>	<code>BlockingIOError</code>	
<code>PyExc_BrokenPipeError</code>	<code>BrokenPipeError</code>	
<code>PyExc_BufferError</code>	<code>BufferError</code>	
<code>PyExc_ChildProcessError</code>	<code>ChildProcessError</code>	
<code>PyExc_ConnectionAbortedError</code>	<code>ConnectionAbortedError</code>	
<code>PyExc_ConnectionError</code>	<code>ConnectionError</code>	
<code>PyExc_ConnectionRefusedError</code>	<code>ConnectionRefusedError</code>	
<code>PyExc_ConnectionResetError</code>	<code>ConnectionResetError</code>	
<code>PyExc_EOFError</code>	<code>EOFError</code>	
<code>PyExc_FileExistsError</code>	<code>FileExistsError</code>	
<code>PyExc_FileNotFoundError</code>	<code>FileNotFoundError</code>	
<code>PyExc_FloatingPointError</code>	<code>FloatingPointError</code>	
<code>PyExc_GeneratorExit</code>	<code>GeneratorExit</code>	
<code>PyExc_ImportError</code>	<code>ImportError</code>	
<code>PyExc_IndentationError</code>	<code>IndentationError</code>	
<code>PyExc_IndexError</code>	<code>IndexError</code>	
<code>PyExc_InterruptedError</code>	<code>InterruptedError</code>	
<code>PyExc_IsADirectoryError</code>	<code>IsADirectoryError</code>	
<code>PyExc_KeyError</code>	<code>KeyError</code>	
<code>PyExc_KeyboardInterrupt</code>	<code>KeyboardInterrupt</code>	
<code>PyExc_LookupError</code>	<code>LookupError</code>	1
<code>PyExc_MemoryError</code>	<code>MemoryError</code>	
<code>PyExc_ModuleNotFoundError</code>	<code>ModuleNotFoundError</code>	
<code>PyExc_NameError</code>	<code>NameError</code>	
<code>PyExc_NotADirectoryError</code>	<code>NotADirectoryError</code>	
<code>PyExc_NotImplementedError</code>	<code>NotImplementedError</code>	
<code>PyExc_OSError</code>	<code>OSError</code>	1
<code>PyExc_OverflowError</code>	<code>OverflowError</code>	
<code>PyExc_PermissionError</code>	<code>PermissionError</code>	
<code>PyExc_ProcessLookupError</code>	<code>ProcessLookupError</code>	

Continúa en la página siguiente

Tabla 1 – proviene de la página anterior

Nombre en C	Nombre en Python	Notas
PyExc_RecursionError	RecursionError	
PyExc_ReferenceError	ReferenceError	
PyExc_RuntimeError	RuntimeError	
PyExc_StopAsyncIteration	StopAsyncIteration	
PyExc_StopIteration	StopIteration	
PyExc_SyntaxError	SyntaxError	
PyExc_SystemError	SystemError	
PyExc_SystemExit	SystemExit	
PyExc_TabError	TabError	
PyExc_TimeoutError	TimeoutError	
PyExc_TypeError	TypeError	
PyExc_UnboundLocalError	UnboundLocalError	
PyExc_UnicodeDecodeError	UnicodeDecodeError	
PyExc_UnicodeEncodeError	UnicodeEncodeError	
PyExc_UnicodeError	UnicodeError	
PyExc_UnicodeTranslateError	UnicodeTranslateError	
PyExc_ValueError	ValueError	
PyExc_ZeroDivisionError	ZeroDivisionError	

Nuevo en la versión 3.3: PyExc_BlockingIOError, PyExc_BrokenPipeError, PyExc_ChildProcessError, PyExc_ConnectionError, PyExc_ConnectionAbortedError, PyExc_ConnectionRefusedError, PyExc_ConnectionResetError, PyExc_FileExistsError, PyExc_FileNotFoundError, PyExc_InterruptedError, PyExc_IsADirectoryError, PyExc_NotADirectoryError, PyExc_PermissionError, PyExc_ProcessLookupError y PyExc_TimeoutError fueron introducidos luego de [PEP 3151](#).

Nuevo en la versión 3.5: PyExc_StopAsyncIteration y PyExc_RecursionError.

Nuevo en la versión 3.6: PyExc_ModuleNotFoundError.

Estos son alias de compatibilidad para PyExc_OSError:

Nombre en C	Notas
PyExc_EnvironmentError	
PyExc_IOError	
PyExc_WindowsError	²

Distinto en la versión 3.3: Estos alias solían ser tipos de excepción separados.

Notas:

¹ Esta es una clase base para otras excepciones estándar.

² Solo se define en Windows; protege el código que usa esto probando que la macro del preprocesador `MS_WINDOWS` está definida.

5.11 Categorías de advertencia estándar

Todas las categorías de advertencia estándar de Python están disponibles como variables globales cuyos nombres son `PyExc_` seguidos del nombre de excepción de Python. Estos tienen el tipo *PyObject**; todos son objetos de clase. Para completar, aquí están todas las variables:

Nombre en C	Nombre en Python	Notas
<code>PyExc_Warning</code>	<code>Warning</code>	³
<code>PyExc_BytesWarning</code>	<code>BytesWarning</code>	
<code>PyExc_DeprecationWarning</code>	<code>DeprecationWarning</code>	
<code>PyExc_FutureWarning</code>	<code>FutureWarning</code>	
<code>PyExc_ImportWarning</code>	<code>ImportWarning</code>	
<code>PyExc_PendingDeprecationWarning</code>	<code>PendingDeprecationWarning</code>	
<code>PyExc_ResourceWarning</code>	<code>ResourceWarning</code>	
<code>PyExc_RuntimeWarning</code>	<code>RuntimeWarning</code>	
<code>PyExc_SyntaxWarning</code>	<code>SyntaxWarning</code>	
<code>PyExc_UnicodeWarning</code>	<code>UnicodeWarning</code>	
<code>PyExc_UserWarning</code>	<code>UserWarning</code>	

Nuevo en la versión 3.2: `PyExc_ResourceWarning`.

Notas:

³ Esta es una clase base para otras categorías de advertencia estándar.

Las funciones de este capítulo realizan varias tareas de utilidad, que van desde ayudar a que el código C sea más portátil en todas las plataformas, usar módulos Python desde C y analizar argumentos de funciones y construir valores Python a partir de valores C.

6.1 Utilidades del sistema operativo

*PyObject** **PyOS_FSPath** (*PyObject* *path)

Return value: *New reference.* Retorna la representación del sistema de archivos para *path*. Si el objeto es *str* o *bytes*, entonces su conteo de referencias se incrementa. Si el objeto implementa la interfaz *os.PathLike*, entonces *__fspath__()* se retorna siempre que sea un objeto *str* o *bytes*. De lo contrario *TypeError* se lanza y se retorna *NULL*.

Nuevo en la versión 3.6.

int **Py_FdIsInteractive** (*FILE* *fp, *const char* *filename)

Retorna verdadero (distinto de cero) si el archivo de E/S (*I/O*) estándar *fp* con nombre *filename* se considera interactivo. Este es el caso de los archivos para los que *isatty(fileno(fp))* es verdadero. Si el indicador global *Py_InteractiveFlag* es verdadero, esta función también retorna verdadero si el puntero *filename* es *NULL* o si el nombre es igual a una de las cadenas de caracteres '*<stdin>*' o '*???*'.

void **PyOS_BeforeFork** ()

Función para preparar algún estado interno antes de una bifurcación de proceso (*process fork*). Esto debería llamarse antes de llamar a *fork()* o cualquier función similar que clone el proceso actual. Solo disponible en sistemas donde *fork()* está definido.

Advertencia: La llamada C *fork()* solo debe hacerse desde *hilo «principal»* (del intérprete *«principal»*). Lo mismo es cierto para *PyOS_BeforeFork()*.

Nuevo en la versión 3.7.

void **PyOS_AfterFork_Parent** ()

Función para actualizar algún estado interno después de una bifurcación de proceso. Se debe invocar desde el proceso principal después de llamar a `fork()` o cualquier función similar que clone el proceso actual, independientemente de si la clonación del proceso fue exitosa. Solo disponible en sistemas donde `fork()` está definido.

Advertencia: La llamada C `fork()` solo debe hacerse desde *hilo «principal»* (del intérprete *«principal»*). Lo mismo es cierto para `PyOS_AfterFork_Parent()`.

Nuevo en la versión 3.7.

void **PyOS_AfterFork_Child** ()

Función para actualizar el estado del intérprete interno después de una bifurcación de proceso (*process fork*). Debe llamarse desde el proceso secundario después de llamar a `fork()`, o cualquier función similar que clone el proceso actual, si existe alguna posibilidad de que el proceso vuelva a llamar al intérprete de Python. Solo disponible en sistemas donde `fork()` está definido.

Advertencia: La llamada C `fork()` solo debe hacerse desde *hilo «principal»* (del intérprete *«principal»*). Lo mismo es cierto para `PyOS_AfterFork_Child()`.

Nuevo en la versión 3.7.

Ver también:

`os.register_at_fork()` permite registrar funciones personalizadas de Python a las que puede llamar `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` y `PyOS_AfterFork_Child()`.

void **PyOS_AfterFork** ()

Función para actualizar algún estado interno después de una bifurcación de proceso (*process fork*); Esto debería llamarse en el nuevo proceso si el intérprete de Python continuará siendo utilizado. Si se carga un nuevo ejecutable en el nuevo proceso, no es necesario llamar a esta función.

Obsoleto desde la versión 3.7: Esta función es reemplazada por `PyOS_AfterFork_Child()`.

int **PyOS_CheckStack** ()

Retorna verdadero cuando el intérprete se queda sin espacio de pila (*stack space*). Esta es una verificación confiable, pero solo está disponible cuando `USE_STACKCHECK` está definido (actualmente en Windows usando el compilador *Microsoft Visual C++*). `USE_STACKCHECK` se definirá automáticamente; nunca debe cambiar la definición en su propio código.

`PyOS_sighandler_t` **PyOS_getsig** (int *i*)

Retorna el controlador de señal actual para la señal *i*. Esta es una pequeña envoltura alrededor de `sigaction()` o `signal()`. ¡No llame a esas funciones directamente! `PyOS_sighandler_t` es un alias *typedef* para `void (*) (int)`.

`PyOS_sighandler_t` **PyOS_setsig** (int *i*, `PyOS_sighandler_t` *h*)

Configura el controlador de señal para la señal *i* como *h*; retorna el antiguo manejador de señal. Esta es una pequeña envoltura alrededor de `sigaction()` o `signal()`. ¡No llame a esas funciones directamente! `PyOS_sighandler_t` es un alias *typedef* para `void (*) (int)`.

wchar_t* **Py_DecodeLocale** (const char* *arg*, size_t **size*)

Decodifica una cadena de bytes de la codificación de configuración regional con controlador de error de subroga-teescape: los bytes no codificables se decodifican como caracteres en el rango U+DC80..U+DCFF. Si una secuencia de bytes se puede decodificar como un carácter sustituto, escape los bytes usando el controlador de error de escape sustituto en lugar de decodificarlos.

Codificación, de máxima prioridad a menor prioridad:

- “UTF-8” en macOS, Android y VxWorks;
- UTF-8 en Windows si `Py_LegacyWindowsFSEncodingFlag` es cero;
- “UTF-8” si el modo Python UTF-8 está habilitado;
- ASCII si la configuración regional `LC_CTYPE` es "C", `nl_langinfo (CODESET)` retorna la codificación ASCII (o un alias) y las funciones `mbstowcs()` y `wcstombs()` utilizan la codificación ISO-8859-1.
- la codificación de la configuración regional actual.

Retorna un puntero a una cadena de caracteres anchos recientemente asignada, use `PyMem_RawFree()` para liberar la memoria. Si el tamaño no es NULL, escribe el número de caracteres anchos excluyendo el carácter nulo en `*size`

Retorna NULL en caso de error de decodificación o error de asignación de memoria. Si `size` no es NULL, `*size` se establece en `(size_t) -1` en caso de error de memoria o en `(size_t) -2` en caso de error de decodificación.

Los errores de decodificación nunca deberían ocurrir, a menos que haya un error en la biblioteca C.

Utilice la función `Py_EncodeLocale()` para codificar la cadena de caracteres en una cadena de bytes.

Ver también:

Las funciones `PyUnicode_DecodeFSDefaultAndSize()` y `PyUnicode_DecodeLocaleAndSize()`.

Nuevo en la versión 3.5.

Distinto en la versión 3.7: La función ahora usa la codificación UTF-8 en el modo UTF-8.

Distinto en la versión 3.8: La función ahora usa la codificación UTF-8 en Windows si `Py_LegacyWindowsFSEncodingFlag` es cero;

char* **Py_EncodeLocale** (const wchar_t *text, size_t *error_pos)

Codifica una cadena de caracteres anchos en la codificación de configuración regional con controlador de error de surrogateescape: los caracteres sustitutos en el rango U+DC80..U+DCFF se convierten en bytes 0x80..0xFF.

Codificación, de máxima prioridad a menor prioridad:

- “UTF-8” en macOS, Android y VxWorks;
- UTF-8 en Windows si `Py_LegacyWindowsFSEncodingFlag` es cero;
- “UTF-8” si el modo Python UTF-8 está habilitado;
- ASCII si la configuración regional `LC_CTYPE` es "C", `nl_langinfo (CODESET)` retorna la codificación ASCII (o un alias) y las funciones `mbstowcs()` y `wcstombs()` utilizan la codificación ISO-8859-1.
- la codificación de la configuración regional actual.

La función utiliza la codificación UTF-8 en el modo Python UTF-8.

Return a pointer to a newly allocated byte string, use `PyMem_Free()` to free the memory. Return NULL on encoding error or memory allocation error.

Si `error_pos` no es NULL, `*error_pos` se establece en `(size_t) -1` en caso de éxito, o se establece en el índice del carácter no válido en el error de codificación.

Use la función `Py_DecodeLocale()` para decodificar la cadena de bytes en una cadena de caracteres anchos.

Ver también:

Las funciones `PyUnicode_EncodeFSDefault()` y `PyUnicode_EncodeLocale()`.

Nuevo en la versión 3.5.

Distinto en la versión 3.7: La función ahora usa la codificación UTF-8 en el modo UTF-8.

Distinto en la versión 3.8: The function now uses the UTF-8 encoding on Windows if `Py_LegacyWindowsFSEncodingFlag` is zero.

6.2 Funciones del Sistema

Estas son funciones de utilidad que hacen que la funcionalidad del módulo `sys` sea accesible para el código C. Todos funcionan con el diccionario del módulo `sys` del subprocesso actual del intérprete, que está contenido en la estructura interna del estado del subprocesso.

PyObject **PySys_GetObject** (const char *name)

Return value: Borrowed reference. Retorna el objeto `name` del módulo `sys` o `NULL` si no existe, sin establecer una excepción.

int **PySys_SetObject** (const char *name, *PyObject* *v)

Establece `name` en el módulo `sys` en `v` a menos que `v` sea `NULL`, en cuyo caso `name` se elimina del módulo `sys`. Retorna 0 en caso de éxito, -1 en caso de error.

void **PySys_ResetWarnOptions** ()

Restablece `sys.warnoptions` a una lista vacía. Esta función puede llamarse antes de `Py_Initialize()`.

void **PySys_AddWarnOption** (const wchar_t *s)

Agrega `s` a `sys.warnoptions`. Esta función debe llamarse antes de `Py_Initialize()` para afectar la lista de filtros de advertencias.

void **PySys_AddWarnOptionUnicode** (*PyObject* *unicode)

Agrega `unicode` a `sys.warnoptions`.

Nota: esta función no se puede utilizar actualmente desde fuera de la implementación de CPython, ya que debe llamarse antes de la importación implícita de `warnings` en `Py_Initialize()` para que sea efectiva, pero no se puede llamar hasta que se haya inicializado suficiente tiempo de ejecución para permitir la creación de objetos Unicode.

void **PySys_SetPath** (const wchar_t *path)

Establece `sys.path` en un objeto lista de rutas que se encuentra en `path`, que debería ser una lista de rutas separadas con el delimitador de ruta de búsqueda de la plataforma (`:` en Unix, `;` en Windows)

void **PySys_WriteStdout** (const char *format, ...)

Escribe la cadena de caracteres de salida descrita por `format` en `sys.stdout`. No se lanzan excepciones, incluso si se produce el truncamiento (ver más abajo).

`format` debe limitar el tamaño total de la cadena de caracteres de salida formateada a 1000 bytes o menos; después de 1000 bytes, la cadena de caracteres de salida se trunca. En particular, esto significa que no deben existir formatos «%s» sin restricciones; estos deben limitarse usando «%.<N>s» donde <N> es un número decimal calculado de modo que <N> más el tamaño máximo de otro texto formateado no exceda los 1000 bytes. También tenga cuidado con «%f», que puede imprimir cientos de dígitos para números muy grandes.

Si ocurre un problema, o `sys.stdout` no está configurado, el mensaje formateado se escribe en el real (nivel C) `stdout`.

void **PySys_WriteStderr** (const char *format, ...)

Como `PySys_WriteStdout()`, pero escribe a `sys.stderr` o `stderr` en su lugar.

void **PySys_FormatStdout** (const char *format, ...)

Función similar a `PySys_WriteStdout()` pero formatea el mensaje usando `PyUnicode_FromFormatV()` y no trunca el mensaje a una longitud arbitraria.

Nuevo en la versión 3.2.

void **PySys_FormatStderr** (const char **format*, ...)

Como `PySys_FormatStdout()`, pero escribe a `sys.stderr` o `stderr` en su lugar.

Nuevo en la versión 3.2.

void **PySys_AddXOption** (const wchar_t **s*)

Analiza (*parse*) *s* como un conjunto de opciones `-X` y los agrega a la asignación de opciones actual tal como lo retorna `PySys_GetXOptions()`. Esta función puede llamarse antes de `Py_Initialize()`.

Nuevo en la versión 3.2.

PyObject ***PySys_GetXOptions** ()

Return value: Borrowed reference. Retorna el diccionario actual de opciones `-X`, de manera similar a `sys._xoptions`. En caso de error, se retorna NULL y se establece una excepción.

Nuevo en la versión 3.2.

int **PySys_Audit** (const char **event*, const char **format*, ...)

Lanza un evento de auditoría con cualquier gancho activo. Retorna cero para el éxito y no cero con una excepción establecida en caso de error.

Si se han agregado ganchos, *format* y otros argumentos se utilizarán para construir una tupla para pasar. Además de N, están disponibles los mismos caracteres de formato que los utilizados en `Py_BuildValue()`. Si el valor generado no es una tupla, se agregará a una tupla de un solo elemento. (La opción de formato N consume una referencia, pero dado que no hay forma de saber si se consumirán argumentos para esta función, su uso puede causar fugas de referencia).

Note that # format characters should always be treated as `Py_ssize_t`, regardless of whether `PY_SSIZE_T_CLEAN` was defined.

`sys.audit()` realiza la misma función del código Python.

Nuevo en la versión 3.8.

Distinto en la versión 3.8.2: Require `Py_ssize_t` for # format characters. Previously, an unavoidable deprecation warning was raised.

int **PySys_AddAuditHook** (Py_AuditHookFunction *hook*, void **userData*)

Append the callable *hook* to the list of active auditing hooks. Return zero on success and non-zero on failure. If the runtime has been initialized, also set an error on failure. Hooks added through this API are called for all interpreters created by the runtime.

El puntero *userData* se pasa a la función gancho. Dado que las funciones de enlace pueden llamarse desde diferentes tiempos de ejecución, este puntero no debe referirse directamente al estado de Python.

Es seguro llamar a esta función antes de `Py_Initialize()`. Cuando se llama después de la inicialización del tiempo de ejecución, se notifican los enlaces de auditoría existentes y pueden anular silenciosamente la operación al generar un error subclassificado de *Excepción* (otros errores no se silenciarán).

La función gancho (*hook*) es de tipo `int (*)(const char *event, PyObject *args, void *userData)`, donde *args* está garantizado como un `PyTupleObject`. La función gancho siempre se llama con el GIL en poder del intérprete de Python que lanzó el evento.

Ver **PEP 578** para una descripción detallada de la auditoría. Las funciones en el tiempo de ejecución y la biblioteca estándar que generan eventos se enumeran en table de eventos de auditoría. Los detalles se encuentran en la documentación de cada función.

Lanza un evento de auditoría `sys.addaudithook` sin argumentos.

Nuevo en la versión 3.8.

6.3 Control de procesos

void **Py_FatalError** (const char *message)

Imprime un mensaje de error fatal y elimina el proceso. No se realiza limpieza. Esta función solo debe invocarse cuando se detecta una condición que haría peligroso continuar usando el intérprete de Python; por ejemplo, cuando la administración del objeto parece estar dañada. En Unix, se llama a la función de biblioteca C estándar `abort()` que intentará producir un archivo `core`.

La función `Py_FatalError()` se reemplaza con una macro que registra automáticamente el nombre de la función actual, a menos que se defina la macro `Py_LIMITED_API`.

Distinto en la versión 3.9: Registra el nombre de la función automáticamente.

void **Py_Exit** (int status)

Sale del proceso actual. Esto llama `Py_FinalizeEx()` y luego llama a la función estándar de la biblioteca C `exit(status)`. Si `Py_FinalizeEx()` indica un error, el estado de salida se establece en 120.

Distinto en la versión 3.6: Los errores de finalización ya no se ignoran.

int **Py_AtExit** (void (*func)())

Registra una función de limpieza a la que llamará `Py_FinalizeEx()`. Se llamará a la función de limpieza sin argumentos y no debería retornar ningún valor. Como máximo se pueden registrar 32 funciones de limpieza. Cuando el registro es exitoso, `Py_AtExit()` retorna 0; en caso de error, retorna -1. La última función de limpieza registrada se llama primero. Cada función de limpieza se llamará como máximo una vez. Dado que la finalización interna de Python se habrá completado antes de la función de limpieza, `func` no debería llamar a las API de Python.

6.4 Importando Módulos

*PyObject** **PyImport_ImportModule** (const char *name)

Return value: New reference. Esta es una interfaz simplificada para `PyImport_ImportModuleEx()` a continuación, dejando los argumentos *globals* y *locals* establecidos en NULL y *level* establecidos en 0. Cuando el argumento *name* contiene un punto (cuando especifica un submódulo de un paquete), el argumento *fromlist* se establece en la lista `['*']` para que el valor de retorno sea el módulo con nombre en lugar del paquete de nivel superior que lo contiene como lo haría de lo contrario sea el caso. (Desafortunadamente, esto tiene un efecto secundario adicional cuando *name* de hecho especifica un subpaquete en lugar de un submódulo: los submódulos especificados en la variable `__all__` del paquete están cargados). Retorna una nueva referencia al módulo importado, o NULL con una excepción establecida en caso de error. Una importación fallida de un módulo no deja el módulo en `sys.modules`.

Esta función siempre usa importaciones absolutas.

*PyObject** **PyImport_ImportModuleNoBlock** (const char *name)

Return value: New reference. Esta función es un alias obsoleto de `PyImport_ImportModule()`.

Distinto en la versión 3.3: Esta función solía fallar inmediatamente cuando el bloqueo de importación era retenido por otro hilo. Sin embargo, en Python 3.3, el esquema de bloqueo cambió a bloqueos por módulo para la mayoría de los propósitos, por lo que el comportamiento especial de esta función ya no es necesario.

*PyObject** **PyImport_ImportModuleEx** (const char *name, *PyObject* *globals, *PyObject* *locals, *PyObject* *fromlist)

Return value: New reference. Importa un módulo. Esto se describe mejor haciendo referencia a la función Python incorporada `__import__()`.

El valor de retorno es una nueva referencia al módulo importado o paquete de nivel superior, o NULL con una excepción establecida en caso de error. Al igual que para `__import__()`, el valor de retorno cuando se solicitó

un submódulo de un paquete normalmente es el paquete de nivel superior, a menos que se proporcione un *fromlist* no vacío.

Las importaciones que fallan eliminan objetos de módulo incompletos, como con `PyImport_ImportModule()`.

PyObject* PyImport_ImportModuleLevelObject (PyObject *name, PyObject *globals, PyObject *locals, PyObject *fromlist, int level)

Return value: New reference. Importa un módulo. Esto se describe mejor haciendo referencia a la función Python incorporada `__import__()`, ya que la función estándar `__import__()` llama a esta función directamente.

El valor de retorno es una nueva referencia al módulo importado o paquete de nivel superior, o NULL con una excepción establecida en caso de error. Al igual que para `__import__()`, el valor de retorno cuando se solicitó un submódulo de un paquete normalmente es el paquete de nivel superior, a menos que se proporcione un *fromlist* no vacío.

Nuevo en la versión 3.3.

PyObject* PyImport_ImportModuleLevel (const char *name, PyObject *globals, PyObject *locals, PyObject *fromlist, int level)

Return value: New reference. Similar a `PyImport_ImportModuleLevelObject()`, pero el nombre es una cadena de caracteres codificada UTF-8 en lugar de un objeto Unicode.

Distinto en la versión 3.3: Los valores negativos para *level* ya no se aceptan.

PyObject* PyImport_Import (PyObject *name)

Return value: New reference. Esta es una interfaz de nivel superior que llama a la «función de enlace de importación» actual (con un nivel explícito de 0, que significa importación absoluta). Invoca la función `__import__()` de las `__builtins__` de los globales (*globals*) actuales. Esto significa que la importación se realiza utilizando los ganchos de importación instalados en el entorno actual.

Esta función siempre usa importaciones absolutas.

PyObject* PyImport_ReloadModule (PyObject *m)

Return value: New reference. Recarga un módulo. Retorna una nueva referencia al módulo recargado, o NULL con una excepción establecida en caso de error (el módulo todavía existe en este caso).

PyObject* PyImport_AddModuleObject (PyObject *name)

Return value: Borrowed reference. Retorna el objeto módulo correspondiente a un nombre de módulo. El argumento *name* puede tener la forma `package.module`. Primero revise el diccionario de módulos si hay uno allí, y si no, crea uno nuevo y lo agrega al diccionario de módulos. Retorna NULL con una excepción establecida en caso de error.

Nota: Esta función no carga ni importa el módulo; si el módulo no estaba cargado, obtendrá un objeto de módulo vacío. Utilice `PyImport_ImportModule()` o una de sus variantes para importar un módulo. Las estructuras de paquete implicadas por un nombre punteado para *name* no se crean si aún no están presentes.

Nuevo en la versión 3.3.

PyObject* PyImport_AddModule (const char *name)

Return value: Borrowed reference. Similar a `PyImport_AddModuleObject()`, pero el nombre es una cadena de caracteres codificada UTF-8 en lugar de un objeto Unicode.

PyObject* PyImport_ExecCodeModule (const char *name, PyObject *co)

Return value: New reference. Dado un nombre de módulo (posiblemente de la forma `package.module`) y un objeto código leído desde un archivo de *bytecode* de Python u obtenido de la función incorporada `compile()`, carga el módulo. Retorna una nueva referencia al objeto módulo, o NULL con una excepción establecida si se produjo un error. *name* se elimina de `sys.modules` en casos de error, incluso si *name* ya estaba en `sys.modules` en

la entrada a `PyImport_ExecCodeModule()`. Dejar módulos inicializados de forma incompleta en `sys.modules` es peligroso, ya que las importaciones de dichos módulos no tienen forma de saber que el objeto del módulo es un estado desconocido (y probablemente dañado con respecto a las intenciones del autor del módulo).

Los módulos `__spec__` y `__loader__` se establecerán, si no se han configurado ya, con los valores apropiados. El cargador de la especificación se establecerá en el módulo `__loader__` (si está configurado) y en una instancia de `SourceFileLoader` de lo contrario.

El atributo del módulo `__file__` se establecerá en el objeto código `co_filename`. Si corresponde, también se establecerá `__cached__`.

Esta función volverá a cargar el módulo si ya se importó. Consulte `PyImport_ReloadModule()` para conocer la forma prevista de volver a cargar un módulo.

Si `name` apunta a un nombre punteado de la forma `package.module`, cualquier estructura de paquete que no se haya creado aún no se creará.

Ver también `PyImport_ExecCodeModuleEx()` y `PyImport_ExecCodeModuleWithPathnames()`.

*PyObject** **PyImport_ExecCodeModuleEx** (const char *name, PyObject *co, const char *pathname)
Return value: New reference. Como `PyImport_ExecCodeModule()`, pero el atributo `__file__` del objeto del módulo se establece en `pathname` si no es NULL.

Ver también `PyImport_ExecCodeModuleWithPathnames()`.

*PyObject** **PyImport_ExecCodeModuleObject** (PyObject *name, PyObject *co, PyObject *pathname, PyObject *cpathname)
Return value: New reference. Como `PyImport_ExecCodeModuleEx()`, pero el atributo `__cached__` del objeto módulo se establece en `cpathname` si no es NULL. De las tres funciones, esta es la recomendada para usar.

Nuevo en la versión 3.3.

*PyObject** **PyImport_ExecCodeModuleWithPathnames** (const char *name, PyObject *co, const char *pathname, const char *cpathname)
Return value: New reference. Como `PyImport_ExecCodeModuleObject()`, pero `name`, `pathname` y `cpathname` son cadenas de caracteres codificadas UTF-8. También se intenta averiguar cuál debe ser el valor de `pathname` de `cpathname` si el primero se establece en NULL.

Nuevo en la versión 3.2.

Distinto en la versión 3.3: Utiliza `imp.source_from_cache()` para calcular la ruta de origen si solo se proporciona la ruta del *bytecode*.

long **PyImport_GetMagicNumber** ()

Retorna el número mágico para los archivos de *bytecode* de Python (también conocido como archivos `.pyc`). El número mágico debe estar presente en los primeros cuatro bytes del archivo de código de bytes, en orden de bytes *little-endian*. Retorna `-1` en caso de error.

Distinto en la versión 3.3: Retorna un valor de `-1` en caso de error.

const char * **PyImport_GetMagicTag** ()

Retorna la cadena de caracteres de etiqueta mágica para nombres de archivo de código de bytes Python en formato **PEP 3147**. Tenga en cuenta que el valor en `sys.implementation.cache_tag` es autoritario y debe usarse en lugar de esta función.

Nuevo en la versión 3.2.

*PyObject** **PyImport_GetModuleDict** ()

Return value: Borrowed reference. Retorna el diccionario utilizado para la administración del módulo (también conocido como `sys.modules`). Tenga en cuenta que esta es una variable por intérprete.

*PyObject** **PyImport_GetModule** (*PyObject* *name)

Return value: New reference. Retorna el módulo ya importado con el nombre dado. Si el módulo aún no se ha importado, retorna NULL pero no establece un error. Retorna NULL y establece un error si falla la búsqueda.

Nuevo en la versión 3.7.

*PyObject** **PyImport_GetImporter** (*PyObject* *path)

Return value: New reference. Retorna un objeto buscador para un elemento *path* `sys.path/pkg.__path__`, posiblemente obteniéndolo del diccionario `sys.path_importer_cache`. Si aún no estaba en caché, atraviesa `sys.path_hooks` hasta que se encuentre un gancho (*hook*) que pueda manejar el elemento de ruta. Retorna None si ningún gancho podría; esto le dice a la persona que llama que *path based finder* no pudo encontrar un buscador para este elemento de ruta. Guarda en el resultado (caché) en `sys.path_importer_cache`. Retorna una nueva referencia al objeto del buscador.

int **PyImport_ImportFrozenModuleObject** (*PyObject* *name)

Return value: New reference. Carga un módulo congelado llamado *name*. Retorna 1 para el éxito, 0 si no se encuentra el módulo y -1 con una excepción establecida si falla la inicialización. Para acceder al módulo importado con una carga exitosa, use `PyImport_ImportModule()`. (Tenga en cuenta el nombre inapropiado — esta función volvería a cargar el módulo si ya se importó).

Nuevo en la versión 3.3.

Distinto en la versión 3.4: El atributo `__file__` ya no está establecido en el módulo.

int **PyImport_ImportFrozenModule** (const char *name)

Similar a `PyImport_ImportFrozenModuleObject()`, pero el nombre es una cadena de caracteres codificada UTF-8 en lugar de un objeto Unicode.

struct **_frozen**

Esta es la definición del tipo de estructura para los descriptores de módulos congelados, según lo generado con la herramienta **freeze** (ver `Tools/freeze` en la distribución de código fuente de Python). Su definición, que se encuentra en `Include/import.h`, es:

```
struct _frozen {
    const char *name;
    const unsigned char *code;
    int size;
};
```

const struct *_frozen** **PyImport_FrozenModules**

Este puntero se inicializa para apuntar a un arreglo de registros `struct _frozen`, terminado por uno cuyos miembros son todos NULL o cero. Cuando se importa un módulo congelado, se busca en esta tabla. El código de terceros podría jugar trucos con esto para proporcionar una colección de módulos congelados creada dinámicamente.

int **PyImport_AppendInittab** (const char *name, *PyObject** (*initfunc)(void))

Agrega un solo módulo a la tabla existente de módulos incorporados. Este es un contenedor conveniente `PyImport_ExtendInittab()`, que retorna -1 si la tabla no se puede extender. El nuevo módulo se puede importar con el nombre *name*, y utiliza la función *initfunc* como la función de inicialización llamada en el primer intento de importación. Esto debería llamarse antes de `Py_Initialize()`.

struct **_inittab**

Estructura que describe una sola entrada en la lista de módulos incorporados. Cada una de estas estructuras proporciona el nombre y la función de inicialización de un módulo incorporado en el intérprete. El nombre es una cadena de caracteres codificada ASCII. Los programas que incorporan Python pueden usar una matriz de estas estructuras junto con `PyImport_ExtendInittab()` para proporcionar módulos integrados adicionales. La estructura se define en `Include/import.h` como:

```
struct _inittab {
    const char *name;           /* ASCII encoded string */
    PyObject* (*initfunc) (void);
};
```

int **PyImport_ExtendInittab** (struct *_inittab* *newtab)

Add a collection of modules to the table of built-in modules. The *newtab* array must end with a sentinel entry which contains NULL for the *name* field; failure to provide the sentinel value can result in a memory fault. Returns 0 on success or -1 if insufficient memory could be allocated to extend the internal table. In the event of failure, no modules are added to the internal table. This must be called before *Py_Initialize()*.

If Python is initialized multiple times, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* must be called before each Python initialization.

6.5 Soporte de empaquetado (*marshalling*) de datos

Estas rutinas permiten que el código C funcione con objetos serializados utilizando el mismo formato de datos que el módulo *marshal*. Hay funciones para escribir datos en el formato de serialización y funciones adicionales que se pueden usar para volver a leer los datos. Los archivos utilizados para almacenar datos ordenados deben abrirse en modo binario.

Los valores numéricos se almacenan con el byte menos significativo primero.

El módulo admite dos versiones del formato de datos: la versión 0 es la versión histórica, la versión 1 comparte cadenas de caracteres internas en el archivo y al desempaquetar (*unmarshalling*). La versión 2 usa un formato binario para números de punto flotante. *Py_MARSHAL_VERSION* indica el formato de archivo actual (actualmente 2).

void **PyMarshal_WriteLongToFile** (long *value*, FILE **file*, int *version*)

Empaqueta (*marshal*) un entero *long*, *value*, a un archivo *file*. Esto solo escribirá los 32 bits menos significativos de *value*; independientemente del tamaño del tipo nativo *long*. *version* indica el formato del archivo.

This function can fail, in which case it sets the error indicator. Use *PyErr_Occurred()* to check for that.

void **PyMarshal_WriteObjectToFile** (*PyObject* **value*, FILE **file*, int *version*)

Empaqueta (*marshal*) un objeto Python, *value*, a un archivo *file*. *version* indica el formato del archivo.

This function can fail, in which case it sets the error indicator. Use *PyErr_Occurred()* to check for that.

*PyObject** **PyMarshal_WriteObjectToString** (*PyObject* **value*, int *version*)

Return value: *New reference*. Retorna un objeto de bytes que contiene la representación empaquetada (*marshalled*) de *value*. *version* indica el formato del archivo.

Las siguientes funciones permiten volver a leer los valores empaquetados (*marshalled*).

long **PyMarshal_ReadLongFromFile** (FILE **file*)

Retorna un C *long* del flujo de datos en un *FILE** abierto para lectura. Solo se puede leer un valor de 32 bits con esta función, independientemente del tamaño nativo de *long*.

En caso de error, establece la excepción apropiada (*EOFError*) y retorna -1.

int **PyMarshal_ReadShortFromFile** (FILE **file*)

Retorna un C *short* desde el flujo de datos en un *FILE** abierto para lectura. Solo se puede leer un valor de 16 bits con esta función, independientemente del tamaño nativo de *short*.

En caso de error, establece la excepción apropiada (*EOFError*) y retorna -1.

*PyObject** **PyMarshal_ReadObjectFromFile** (FILE **file*)

Return value: *New reference*. Retorna un objeto Python del flujo de datos en un *FILE** abierto para lectura.

En caso de error, establece la excepción apropiada (*EOFError*, *ValueError* o *TypeError*) y retorna NULL.

*PyObject** **PyMarshal_ReadLastObjectFromFile** (FILE *file)

Return value: New reference. Retorna un objeto Python del flujo de datos en un FILE* abierto para lectura. A diferencia de *PyMarshal_ReadObjectFromFile()*, esta función asume que no se leerán más objetos del archivo, lo que le permite cargar agresivamente los datos del archivo en la memoria para que la deserialización pueda operar desde los datos en la memoria en lugar de leer un byte a la vez desde el archivo. Solo use esta variante si está seguro de que no leerá nada más del archivo.

En caso de error, establece la excepción apropiada (EOFError, ValueError o TypeError) y retorna NULL.

*PyObject** **PyMarshal_ReadObjectFromString** (const char *data, Py_ssize_t len)

Return value: New reference. Retorna un objeto Python del flujo de datos en un búfer de bytes que contiene *len* bytes a los que apunta *data*.

En caso de error, establece la excepción apropiada (EOFError, ValueError o TypeError) y retorna NULL.

6.6 Analizando argumentos y construyendo valores

Estas funciones son útiles al crear sus propias funciones y métodos de extensiones. Información y ejemplos adicionales están disponibles en *extending-index*.

Las tres primeras de estas funciones descritas, *PyArg_ParseTuple()*, *PyArg_ParseTupleAndKeywords()*, y *PyArg_Parse()*, todas usan *cadenas de caracteres de formato* que se utilizan para contarle a la función sobre los argumentos esperados. Las cadenas de caracteres de formato utilizan la misma sintaxis para cada una de estas funciones.

6.6.1 Analizando argumentos

Una cadena de formato consta de cero o más «unidades de formato.» Una unidad de formato describe un objeto Python; por lo general es un solo carácter o una secuencia de unidades formato entre paréntesis. Con unas pocas excepciones, una unidad de formato que no es una secuencia entre paréntesis normalmente corresponde a un único argumento de dirección de estas funciones. En la siguiente descripción, la forma citada es la unidad de formato; la entrada en paréntesis (redondos) es el tipo de objeto Python que coincida con la unidad de formato; y la entrada entre corchetes [cuadrados] es el tipo de la variable(s) C cuya dirección debe ser pasada.

Cadena de caracteres y búferes

Estos formatos permiten acceder a un objeto como un bloque contiguo de memoria. Usted no tiene que proporcionar almacenamiento en bruto para el Unicode o área de bytes retornada.

En general, cuando un formato establece un puntero a un búfer, el búfer es gestionado por el objeto de Python correspondiente, y el búfer comparte la vida útil de este objeto. Usted no tendrá que liberar cualquier memoria usted mismo. Las únicas excepciones son *es*, *es#*, *et* y *et#*.

Sin embargo, cuando una estructura *Py_buffer* se llena, la memoria intermedia subyacente está bloqueada de manera que la persona que llama puede posteriormente utilizar la memoria intermedia incluso dentro de un bloque *Py_BEGIN_ALLOW_THREADS* sin el riesgo de que los datos mutables sean redimensionados o destruidos. Como resultado, **usted tiene que llamar** *PyBuffer_Release()* después de haber terminado de procesar los datos (o en caso de aborto temprano).

A menos que se indique lo contrario, los búferes no son terminados en NULL (*NUL-terminated*).

Algunos formatos requieren *bytes-like object* de sólo lectura, y establecen un puntero en lugar de una estructura de búfer. Trabajan comprobando que el campo del objeto *PyBufferProcs.bf_releasebuffer* es NULL, que no permite objetos mutables como *bytearray*.

Nota: Para todas las variantes de formato de # (s#, y#, etc.), el tipo del argumento *length* (int o *Py_ssize_t*) es controlado por la definición de la macro `PY_SSIZE_T_CLEAN` antes de incluir `Python.h`. Si se ha definido la macro, *length* es un *Py_ssize_t* en lugar de un `int`. Este comportamiento va a cambiar en futuras versiones de Python para soportar únicamente *Py_ssize_t* y dejar el soporte de `int`. Es mejor definir siempre `PY_SSIZE_T_CLEAN`.

s (str) [const char *] Convierte un objeto Unicode a un puntero C a una cadena de caracteres. Un puntero a una cadena de caracteres existente se almacena en la variable puntero del carácter cuya dirección se pasa. La cadena de caracteres en C es terminada en `NULL`. La cadena de caracteres de Python no debe contener puntos de código incrustados nulos; si lo hace, se lanza una excepción `ValueError`. Los objetos Unicode se convierten en cadenas de caracteres de C utilizando codificación 'utf-8'. Si esta conversión fallase lanza un `UnicodeError`.

Nota: Este formato no acepta *objetos de tipo bytes*. Si desea aceptar los caminos del sistema de archivos y convertirlos en cadenas de caracteres C, es preferible utilizar el formato `O&` con `PyUnicode_FSConverter()` como convertidor.

Distinto en la versión 3.5: Anteriormente, `TypeError` se lanzó cuando se encontraron puntos de código nulos incrustados en la cadena de caracteres de Python.

s* (str o bytes-like object) [Py_buffer] Este formato acepta objetos Unicode, así como objetos de tipo bytes. Llena una estructura *Py_buffer* proporcionada por la persona que llama. En este caso la cadena de caracteres de C resultante puede contener bytes NUL embebidos. Los objetos Unicode se convierten en cadenas de caracteres C utilizando codificación 'utf-8'.

s# (str, bytes-like object de sólo lectura) [const char *, int o Py_ssize_t] Como *s**, excepto que no acepta los objetos mutables. El resultado se almacena en dos variables de C, la primera un puntero a una cadena de caracteres C, el segundo es su longitud. La cadena de caracteres puede contener caracteres nulos incrustados. Los objetos Unicode se convierten en cadenas de caracteres C utilizando codificación 'utf-8'.

z (str o None) [const char *] Como *s*, pero el objeto Python también puede ser `None`, en cuyo caso el puntero C se establece en `NULL`.

z* (str, bytes-like object o None) [Py_buffer] Como *s**, pero el objeto Python también puede ser `None`, en cuyo caso el miembro de `buf` de la estructura *Py_buffer* se establece en `NULL`.

z# (str, bytes-like object de sólo lectura o None) [const char *, int o Py_ssize_t] Como *s#*, pero el objeto Python también puede ser `None`, en cuyo caso el puntero C se establece en `NULL`.

y (bytes-like object de sólo lectura) [const char *] Este formato convierte un objeto de tipo bytes a un puntero C a una cadena de caracteres; no acepta objetos Unicode. El búfer de bytes no debe contener bytes nulos incrustados; si lo hace, se lanza una excepción `ValueError`.

Distinto en la versión 3.5: Anteriormente, `TypeError` se lanzó cuando bytes nulos incrustados se encontraron en el buffer de bytes.

y* (bytes-like object) [Py_buffer] Esta variante de *s** no acepta objetos Unicode, solamente los objetos de tipo bytes. Esta es la forma recomendada para aceptar datos binarios.

y# (bytes-like object de sólo lectura) [const char *, int o Py_ssize_t] Esta variante en *s#* no acepta objetos Unicode, solo objetos similares a bytes.

S (bytes) [PyBytesObject *] Requiere que el objeto Python es un objeto `bytes`, sin intentar ninguna conversión. Lanza `TypeError` si el objeto no es un objeto `bytes`. La variable C también puede ser declarado como *PyObject**.

Y (bytearray) [PyByteArrayObject *] Requiere que el objeto Python es un objeto `bytearray`, sin intentar ninguna conversión. Lanza `TypeError` si el objeto no es un objeto `bytearray`. La variable C también puede ser declarada como *PyObject**.

u (str) [const Py_UNICODE *] Convierte un objeto Unicode de Python a un puntero a un búfer C NUL terminado de caracteres Unicode. Debe pasar la dirección de una variable de puntero `Py_UNICODE`, que se llena con el puntero a un búfer Unicode existente. Tenga en cuenta que el ancho de un carácter `Py_UNICODE` depende de las opciones de compilación (que es 16 o 32 bits). La cadena de Python no debe contener puntos de código incrustados nulos; si lo hace, se lanza una excepción `ValueError`.

Distinto en la versión 3.5: Anteriormente, `TypeError` se lanzó cuando se encontraron puntos de código nulos incrustados en la cadena de caracteres de Python.

Deprecated since version 3.3, will be removed in version 3.12: Parte de la API de viejo estilo `Py_UNICODE`; favor migrar al uso de `PyUnicode_AsWideCharString()`.

u# (str) [const Py_UNICODE *, int o Py_ssize_t] Esta variante en `u` almacena en dos variables de C, el primero un puntero a un búfer de datos Unicode, el segundo de su longitud. Esta variante permite puntos de código nulos.

Deprecated since version 3.3, will be removed in version 3.12: Parte de la API de viejo estilo `Py_UNICODE`; favor migrar al uso de `PyUnicode_AsWideCharString()`.

z (str o None) [const Py_UNICODE *] Como `u`, pero el objeto Python también puede ser `None`, en cuyo caso el puntero `Py_UNICODE` se establece en `NULL`.

Deprecated since version 3.3, will be removed in version 3.12: Parte de la API de viejo estilo `Py_UNICODE`; favor migrar al uso de `PyUnicode_AsWideCharString()`.

z# (str o None) [const Py_UNICODE *, int o Py_ssize_t] Al igual que `u#`, pero el objeto Python también puede ser `None`, en cuyo caso el puntero `Py_UNICODE` se establece en `NULL`.

Deprecated since version 3.3, will be removed in version 3.12: Parte de la API de viejo estilo `Py_UNICODE`; favor migrar al uso de `PyUnicode_AsWideCharString()`.

U (str) [PyObject *] Requiere que el objeto Python es un objeto Unicode, sin intentar ninguna conversión. Lanza `TypeError` si el objeto no es un objeto Unicode. La variable C también puede ser declarada como `PyObject *`.

w* (bytes-like object de lectura y escritura) [Py_buffer] Este formato acepta cualquier objeto que implemente la interfaz del búfer de lectura-escritura. Llena la estructura `Py_buffer` proporcionada por quien llama. El búfer puede contener bytes nulos incrustados. Quien llama tiene que llamar `PyBuffer_Release()` cuando termina con el búfer.

es (str) [const char *encoding, char **buffer] Esta variante en `s` se usa para codificar Unicode en un búfer de caracteres. Solo funciona para datos codificados sin bytes NUL integrados.

Este formato requiere dos argumentos. El primero solo se usa como entrada, y debe ser `const char *` que apunta al nombre de una codificación como una cadena de caracteres terminada en NUL, o `NULL`, en cuyo caso se utiliza la codificación `'utf-8'`. Se lanza una excepción si Python no conoce la codificación con nombre. El segundo argumento debe ser `char **`; el valor del puntero al que hace referencia se establecerá en un búfer con el contenido del texto del argumento. El texto se codificará en la codificación especificada por el primer argumento.

`PyArg_ParseTuple()` asignará un búfer del tamaño necesitado, copiará los datos codificados en este búfer y ajustará `*buffer` para referenciar el nuevo almacenamiento asignado. Quien llama es responsable para llamar `PyMem_Free()` para liberar el búfer asignado después de su uso.

et (str, bytes o bytearray) [const char *encoding, char **buffer] Igual que `es`, excepto que los objetos de cadena de caracteres de bytes se pasan sin recodificarlos. En cambio, la implementación supone que el objeto de cadena de caracteres de bytes utiliza la codificación que se pasa como parámetro.

es# (str) [const char *encoding, char **buffer, int o Py_ssize_t *buffer_length] Esta variante en `s#` se usa para codificar Unicode en un búfer de caracteres. A diferencia del formato `es`, esta variante permite datos de entrada que contienen caracteres NUL.

Requiere tres argumentos. El primero solo se usa como entrada, y debe ser `const char *` que apunta al nombre de una codificación como una cadena de caracteres terminada en NUL, o `NULL`, en cuyo caso se utiliza la codificación `'utf-8'`. Se lanza una excepción si Python no conoce la codificación con nombre. El

segundo argumento debe ser `char**`; El valor del puntero al que hace referencia se establecerá en un búfer con el contenido del texto del argumento. El texto se codificará en la codificación especificada por el primer argumento. El tercer argumento debe ser un puntero a un entero; el número entero referenciado se establecerá en el número de bytes en el búfer de salida.

Hay dos modos de operación:

Si `*buffer` señala un puntero `NULL`, la función asignará un búfer del tamaño necesario, copiará los datos codificados en este búfer y configurará `*buffer` para hacer referencia al almacenamiento recién asignado. Quien llama es responsable de llamar a `PyMem_Free()` para liberar el búfer asignado después del uso.

Si `*buffer` apunta a un puntero no `NULL` (un búfer ya asignado), `PyArg_ParseTuple()` usará esta ubicación como el búfer e interpretará el valor inicial de `*buffer_length` como el tamaño del búfer. Luego copiará los datos codificados en el búfer y los terminará en `NUL`. Si el búfer no es lo suficientemente grande, se establecerá a `ValueError`.

En ambos casos, `*buffer_length` se establece a la longitud de los datos codificados sin el byte `NUL` final.

et# (str, bytes o bytearray) [const char *encoding, char **buffer, int o `Py_ssize_t` *buffer_length]

Igual que `es#`, excepto que los objetos de cadena de caracteres de bytes se pasan sin recodificarlos. En cambio, la implementación supone que el objeto de cadena de caracteres de bytes utiliza la codificación que se pasa como parámetro.

Números

b (int) [unsigned char] Convierte un entero de Python no negativo en un pequeño `int` sin signo, almacenado en un `unsigned char` de C.

B (int) [unsigned char] Convierte un entero de Python en un pequeño `int` sin comprobación de desbordamiento, almacenado en un `unsigned char` de C.

h (int) [short int] Convierte un entero de Python a un `short int` de C.

H (int) [unsigned short int] Convierte un entero de Python a un `unsigned short int` de C, sin verificación de desbordamiento.

i (int) [int] Convierte un entero Python a un `int` de C plano.

I (int) [unsigned int] Convierte un entero de Python a un `unsigned int` de C, sin verificación de desbordamiento.

l (int) [long int] Convierte un entero Python a un `long int` de C.

k (int) [unsigned long] Convierte un entero de Python a un `unsigned long` de C, sin verificación de desbordamiento.

L (int) [long long] Convierte un entero de Python a un `long long` de C.

K (int) [unsigned long long] Convierte un entero de Python a un `unsigned long long` de C, sin verificación de desbordamiento.

n (int) [Py_ssize_t] Convierte un entero de Python a un `Py_ssize_t` de C.

c (bytes o bytearray de largo 1) [char] Convierte un byte de Python, representado como un objeto `bytes` o `bytearray` de longitud 1, a un `char` de C.

Distinto en la versión 3.3: Permite objetos `bytearray`.

C (str de largo 1) [int] Convierte un carácter Python, representado como un objeto `str` de longitud 1, a un tipo `int` de C.

f (float) [float] Convierte un número de punto flotante de Python a un `float` de C.

d (float) [double] Convierte un número de punto flotante de Python a un `double` de C.

D (complex) [Py_complex] Convierte un número complejo de Python en una estructura *Py_complex* de C.

Otros objetos

O (object) [PyObject *] Almacena un objeto Python (sin ninguna conversión) en un puntero de objeto C. El programa C recibe así el objeto real que se pasó. El recuento de referencia del objeto no aumenta. El puntero almacenado no es NULL.

O! (object) [PyObject *, PyObject *] Almacena un objeto Python en un puntero de objeto C. Esto es similar a O, pero toma dos argumentos C: el primero es la dirección de un objeto de tipo Python, el segundo es la dirección de la variable C (de tipo *PyObject **) en el que se almacena el puntero del objeto. Si el objeto Python no tiene el tipo requerido, se lanza *TypeError*.

O& (object) [converter, anything] Convierte un objeto Python en una variable C a través de una función *converter*. Esto requiere dos argumentos: el primero es una función, el segundo es la dirección de una variable C (de tipo arbitrario), convertida a *void **. La función *converter* a su vez se llama de la siguiente manera:

```
status = converter(object, address);
```

donde *object* es el objeto de Python a convertir y *address* es el argumento *void** que se pasó a la función *PyArg_Parse*()*. El *status* retornado debe ser 1 para una conversión exitosa y 0 si la conversión ha fallado. Cuando la conversión falla, la función *converter* debería generar una excepción y dejar el contenido de *address* sin modificar.

Si el *converter* retorna *Py_CLEANUP_SUPPORTED*, se puede llamar por segunda vez si el análisis del argumento finalmente falla, dando al convertidor la oportunidad de liberar cualquier memoria que ya haya asignado. En esta segunda llamada, el parámetro *object* será NULL; *address* tendrá el mismo valor que en la llamada original.

Distinto en la versión 3.1: *Py_CLEANUP_SUPPORTED* fue agregada.

p (bool) [int] Prueba el valor pasado por verdad (un booleano predicado *p*) y convierte el resultado a su valor entero C verdadero/falso entero equivalente. Establece *int* en 1 si la expresión era verdadera y 0 si era falsa. Esto acepta cualquier valor válido de Python. Consulte *truth* para obtener más información sobre cómo Python prueba los valores por verdad.

Nuevo en la versión 3.3.

(items) (tuple) [matching-items] El objeto debe ser una secuencia de Python cuya longitud es el número de unidades de formato en *items*. Los argumentos C deben corresponder a las unidades de formato individuales en *items*. Las unidades de formato para secuencias pueden estar anidadas.

Es posible pasar enteros «largos» (enteros cuyo valor excede el de la plataforma *LONG_MAX*), sin embargo, no se realiza una verificación de rango adecuada — los bits más significativos se truncan silenciosamente cuando el campo receptor es demasiado pequeño para recibir el valor (en realidad, la semántica se hereda de las descargas en C — su kilometraje puede variar).

Algunos otros caracteres tienen un significado en una cadena de formato. Esto puede no ocurrir dentro de paréntesis anidados. Son:

- | Indica que los argumentos restantes en la lista de argumentos de Python son opcionales. Las variables C correspondientes a argumentos opcionales deben inicializarse a su valor predeterminado — cuando no se especifica un argumento opcional, *PyArg_ParseTuple()* no toca el contenido de las variables C correspondientes.

- \$ *PyArg_ParseTupleAndKeywords()* solamente: indica que los argumentos restantes en la lista de argumentos de Python son solo palabras clave. Actualmente, todos los argumentos de solo palabras clave también deben ser argumentos opcionales, por lo que | siempre debe especificarse antes de \$ en la cadena de formato.

Nuevo en la versión 3.3.

- : La lista de unidades de formato termina aquí; la cadena después de los dos puntos se usa como el nombre de la función en los mensajes de error (el «valor asociado» de la excepción que `PyArg_ParseTuple()` lanza).
- ; La lista de unidades de formato termina aquí; la cadena después del punto y coma se usa como mensaje de error *en lugar de* del mensaje de error predeterminado. : y ; se excluyen mutuamente.

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

Los argumentos adicionales pasados a estas funciones deben ser direcciones de variables cuyo tipo está determinado por la cadena de formato; Estos se utilizan para almacenar valores de la tupla de entrada. Hay algunos casos, como se describe en la lista de unidades de formato anterior, donde estos parámetros se utilizan como valores de entrada; deben coincidir con lo especificado para la unidad de formato correspondiente en ese caso.

Para que la conversión tenga éxito, el objeto *arg* debe coincidir con el formato y el formato debe estar agotado. En caso de éxito, las funciones `PyArg_Parse*()` retornan verdadero; de lo contrario, retornan falso y generan una excepción apropiada. Cuando las funciones `PyArg_Parse*()` fallan debido a un error de conversión en una de las unidades de formato, las variables en las direcciones correspondientes y las siguientes unidades de formato quedan intactas.

Funciones API

int **PyArg_ParseTuple** (*PyObject* *args, const char *format, ...)

Analiza los parámetros de una función que solo toma parámetros posicionales en variables locales. Retorna verdadero en el éxito; en caso de fallo, retorna falso y genera la excepción apropiada.

int **PyArg_VaParse** (*PyObject* *args, const char *format, va_list vars)

Idéntico a `PyArg_ParseTuple()`, excepto que acepta una *va_list* en lugar de un número variable de argumentos.

int **PyArg_ParseTupleAndKeywords** (*PyObject* *args, *PyObject* *kw, const char *format, char *keywords[], ...)

Analiza los parámetros de una función que toma parámetros posicionales y de palabras clave en variables locales. El argumento *keywords* es un arreglo terminado en NULL de nombres de parámetros de palabras clave. Los nombres vacíos denotan *parámetros solo posicionales*. Retorna verdadero cuando hay éxito; en caso de fallo, retorna falso y genera la excepción apropiada.

Distinto en la versión 3.6: Soporte agregado para *sólo parámetros posicionales*.

int **PyArg_VaParseTupleAndKeywords** (*PyObject* *args, *PyObject* *kw, const char *format, char *keywords[], va_list vars)

Idéntico a `PyArg_ParseTupleAndKeywords()`, excepto que acepta una *va_list* en lugar de un número variable de argumentos.

int **PyArg_ValidateKeywordArguments** (*PyObject* *)

Asegúrese de que las claves en el diccionario de argumentos de palabras clave son cadenas. Esto solo es necesario si `PyArg_ParseTupleAndKeywords()` no se utiliza, ya que este último ya hace esta comprobación.

Nuevo en la versión 3.2.

int **PyArg_Parse** (*PyObject* *args, const char *format, ...)

Función utilizada para deconstruir las listas de argumentos de las funciones de «estilo antiguo» — estas son funciones que usan el método de análisis de parámetros `METH_OLDARGS`, que se ha eliminado en Python 3. No se recomienda su uso en el análisis de parámetros en código nuevo, y la mayoría del código en el intérprete estándar se ha modificado para que ya no se use para ese propósito. Sin embargo, sigue siendo una forma conveniente de descomponer otras tuplas, y puede continuar usándose para ese propósito.

int **PyArg_UnpackTuple** (*PyObject* *args, const char *name, *Py_ssize_t* min, *Py_ssize_t* max, ...)

Una forma más simple de recuperación de parámetros que no utiliza una cadena de formato para especificar los tipos de argumentos. Las funciones que utilizan este método para recuperar sus parámetros deben declararse como `METH_VARARGS` en las tablas de funciones o métodos. La tupla que contiene los parámetros reales debe pasarse

como *args*; en realidad debe ser una tupla. La longitud de la tupla debe ser al menos *min* y no más de *max*; *min* y *max* pueden ser iguales. Se deben pasar argumentos adicionales a la función, cada uno de los cuales debe ser un puntero a una variable *PyObject**; estos se completarán con los valores de *args*; contendrán referencias prestadas. Las variables que corresponden a parámetros opcionales no dados por *args* no se completarán; estos deben ser inicializados por quien llama. Esta función retorna verdadero en caso de éxito y falso si *args* no es una tupla o contiene el número incorrecto de elementos; se establecerá una excepción si hubo una falla.

Este es un ejemplo del uso de esta función, tomado de las fuentes del módulo auxiliar *_weakref* para referencias débiles:

```
static PyObject *
weakref_ref(PyObject *self, PyObject *args)
{
    PyObject *object;
    PyObject *callback = NULL;
    PyObject *result = NULL;

    if (PyArg_UnpackTuple(args, "ref", 1, 2, &object, &callback)) {
        result = PyWeakref_NewRef(object, callback);
    }
    return result;
}
```

La llamada a *PyArg_UnpackTuple()* en este ejemplo es completamente equivalente a esta llamada a *PyArg_ParseTuple()*:

```
PyArg_ParseTuple(args, "O|O:ref", &object, &callback)
```

6.6.2 Construyendo valores

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

Return value: *New reference.* Crea un nuevo valor basado en una cadena de formato similar a los aceptados por la familia de funciones *PyArg_Parse**() y una secuencia de valores. Retorna el valor o NULL en caso de error; se generará una excepción si se retorna NULL.

Py_BuildValue() no siempre genera 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, paréntesis la cadena de formato.

Cuando los búfer de memoria se pasan como parámetros para suministrar datos para construir objetos, como para los formatos *s* y *s#*, los datos requeridos se copian. Las memorias intermedias proporcionadas por quien llama nunca son referenciadas por los objetos creados por *Py_BuildValue()*. En otras palabras, si su código invoca *malloc()* y pasa la memoria asignada a *Py_BuildValue()*, su código es responsable de llamar a *free()* para esa memoria una vez retorna *Py_BuildValue()*.

En la siguiente descripción, la cadena de caracteres entre comillas, *así*, es la unidad de formato; la entrada entre paréntesis (redondos) es el tipo de objeto Python que retornará la unidad de formato; y la entrada entre corchetes [cuadrados] es el tipo de los valores C que se pasarán.

Los caracteres espacio, tabulación, dos puntos y coma se ignoran en las cadenas de formato (pero no dentro de las unidades de formato como *s#*). Esto se puede usar para hacer que las cadenas de formato largo sean un poco más legibles.

s (str o None) [const char*] Convierte una cadena de caracteres C terminada en nulo en un objeto Python *str* usando la codificación 'utf-8'. Si el puntero de la cadena de caracteres C es NULL, se usa None.

- s# (str o None) [const char *, int o `Py_ssize_t`]** Convierte una cadena de caracteres de C y su longitud en un objeto Python `str` utilizando la codificación 'utf-8'. Si el puntero de la cadena de caracteres de C es NULL, la longitud se ignora y se retorna None.
- y (bytes) [const char *]** Esto convierte una cadena de caracteres de C en un objeto Python `bytes`. Si el puntero de la cadena de caracteres de C es NULL, se retorna None.
- y# (bytes) [const char *, int o `Py_ssize_t`]** Esto convierte una cadena de caracteres de C y sus longitudes en un objeto Python. Si el puntero de la cadena de caracteres de C es NULL, se retorna None.
- z (str o None) [const char *]** Igual que `s`.
- z# (str o None) [const char *, int o `Py_ssize_t`]** Igual que `s#`.
- u (str) [const wchar_t *]** Convierte un búfer `wchar_t` de datos Unicode (UTF-16 o UCS-4) en un objeto Python Unicode. Si el puntero del búfer Unicode es NULL, se retorna None.
- u# (str) [const wchar_t *, int o `Py_ssize_t`]** Convierte un búfer de datos Unicode (UTF-16 o UCS-4) y su longitud en un objeto Python Unicode. Si el puntero del búfer Unicode es NULL, la longitud se ignora y se retorna None.
- U (str o None) [const char *]** Igual que `s`.
- U# (str o None) [const char *, int o `Py_ssize_t`]** Igual que `s#`.
- i (int) [int]** Convierte un `int` plano de C a un objeto entero de Python.
- b (int) [char]** Convierte un `char` plano de C a un objeto entero de Python.
- h (int) [short int]** Convierte un `short int` plano de C a un objeto entero de Python.
- l (int) [long int]** Convierte un `long int` de C en un objeto entero de Python.
- B (int) [unsigned char]** Convierte un `unsigned char` de C a un entero de Python.
- H (int) [unsigned short int]** Convierte un `unsigned short int` de C a un entero de Python.
- I (int) [unsigned int]** Convierte un `unsigned int` de C a un entero de Python.
- k (int) [unsigned long]** Convierte un `unsigned long` de C a un entero de Python.
- L (int) [long long]** Convierte un `long long` de C en un objeto entero de Python.
- K (int) [unsigned long long]** Convierte un `unsigned long long` de C a un entero de Python.
- n (int) [`Py_ssize_t`]** Convierte un `Py_ssize_t` de C a un entero de Python.
- c (bytes de largo 1) [char]** Convierte un `int` de C representando un byte a un objeto `bytes` de Python de largo 1.
- C (str de largo 1) [int]** Convierte un `int` de C representando un carácter a un objeto `str` de Python de largo 1.
- d (float) [double]** Convierte un `double` de C a un número de punto flotante de Python.
- f (float) [float]** Convierte un `float` de C a un número de punto flotante de Python.
- D (complex) [`Py_complex` *]** Convierte una estructura `Py_complex` de C en un número complejo de Python.
- O (object) [`PyObject` *]** Pasa un objeto Python sin tocarlo (excepto por su recuento de referencia, que se incrementa en uno). Si el objeto pasado es un puntero NULL, se supone que esto fue causado porque la llamada que produjo el argumento encontró un error y estableció una excepción. Por lo tanto, `Py_BuildValue()` retornará NULL pero no lanzará una excepción. Si aún no se ha producido ninguna excepción, se establece `SystemError`.
- S (object) [`PyObject` *]** Igual que `O`.

N (object) [PyObject*] Igual que **O**, excepto que no incrementa el recuento de referencia en el objeto. Útil cuando el objeto se crea mediante una llamada a un constructor de objetos en la lista de argumentos.

O& (object) [converter, anything] Convierte *anything* a un objeto Python a través de una función *converter*. La función se llama con *anything* (que debería ser compatible con `void*`) como argumento y debería retornar un «nuevo» objeto de Python, o `NULL` si se produjo un error.

(items) (tuple) [matching-items] Convierta una secuencia de valores C en una tupla de Python con el mismo número de elementos.

[items] (list) [matching-items] Convierte una secuencia de valores C en una lista de Python con el mismo número de elementos.

{items} (dict) [matching-items] Convierte una secuencia de valores C en un diccionario Python. Cada par de valores C consecutivos agrega un elemento al diccionario, que sirve como clave y valor, respectivamente.

Si hay un error en la cadena de formato, se establece la excepción `SystemError` y se retorna `NULL`.

*PyObject** **Py_VaBuildValue** (const char *format, va_list args)

Return value: New reference. Idéntico a `Py_BuildValue()`, excepto que acepta una *va_list* en lugar de un número variable de argumentos.

6.7 Conversión y formato de cadenas de caracteres

Funciones para conversión de números y salida de cadena de caracteres formateadas.

int **PyOS_snprintf** (char *str, size_t size, const char *format, ...)

Salida de no más de *size* bytes a *str* según la cadena de caracteres de formato *format* y los argumentos adicionales. Consulte la página de manual de Unix `snprintf(3)`.

int **PyOS_vsnprintf** (char *str, size_t size, const char *format, va_list va)

Salida de no más de *size* bytes a *str* según la cadena de caracteres de formato *format* y la lista de argumentos variables *va*. Página de manual de Unix `vsnprintf(3)`.

`PyOS_snprintf()` y `PyOS_vsnprintf()` envuelven las funciones estándar de la biblioteca C `snprintf()` y `vsnprintf()`. Su propósito es garantizar un comportamiento consistente en casos de esquina (*corner cases*), que las funciones del Estándar C no hacen.

Las envolturas aseguran que `str[size-1]` sea siempre `'\0'` al retornar. Nunca se escriben más de *size* bytes (incluido el `'\0'` del final) en *str*. Ambas funciones requieren que `str != NULL`, `size > 0` y `format != NULL`.

Si la plataforma no tiene `vsnprintf()` y el tamaño del búfer necesario para evitar el truncamiento excede *size* en más de 512 bytes, Python aborta con a `Py_FatalError()`.

El valor de retorno (*rv*) para estas funciones debe interpretarse de la siguiente manera:

- Cuando `0 <= rv < size`, la conversión de salida fue exitosa y los caracteres *rv* se escribieron en *str* (excluyendo el byte `'\0'` final en `str[rv]`).
- Cuando `rv >= size`, la conversión de salida se truncó y se habría necesitado un búfer con `rv + 1` bytes para tener éxito. `str[size-1]` es `'\0'` en este caso.
- Cuando `rv < 0`, «sucedió algo malo». `str[size-1]` es `'\0'` en este caso también, pero el resto de *str* no está definido. La causa exacta del error depende de la plataforma subyacente.

Las siguientes funciones proporcionan cadenas de caracteres independientes de la configuración regional para numerar las conversiones.

double **PyOS_string_to_double** (const char *s, char **endptr, PyObject *overflow_exception)

Convierte una cadena de caracteres *s* en un `double`, generando una excepción de Python en caso de falla. El conjunto de cadenas de caracteres aceptadas corresponde al conjunto de cadenas aceptadas por el constructor de

Python `float()`, excepto que `s` no debe tener espacios en blanco iniciales o finales. La conversión es independiente de la configuración regional actual.

Si `endptr` es `NULL`, convierte toda la cadena de caracteres. Lanza `ValueError` y retorna `-1.0` si la cadena de caracteres no es una representación válida de un número de punto flotante.

Si `endptr` no es `NULL`, convierte la mayor cantidad posible de la cadena de caracteres y configura `*endptr` para que apunte al primer carácter no convertido. Si ningún segmento inicial de la cadena de caracteres es la representación válida de un número de punto flotante, configura `*endptr` para que apunte al comienzo de la cadena de caracteres, lanza `ValueError` y retorna `-1.0`.

Si `s` representa un valor que es demasiado grande para almacenar en un flotante (por ejemplo, `"1e500"` es una cadena de caracteres de este tipo en muchas plataformas), entonces si `overflow_exception` es `NULL` retorna `Py_HUGE_VAL` (con un signo apropiado) y no establece ninguna excepción. De lo contrario, `overflow_exception` debe apuntar a un objeto excepción de Python; lanza esa excepción y retorna `-1.0`. En ambos casos, configura `*endptr` para que apunte al primer carácter después del valor convertido.

Si se produce algún otro error durante la conversión (por ejemplo, un error de falta de memoria), establece la excepción Python adecuada y retorna `-1.0`.

Nuevo en la versión 3.1.

char* **PyOS_double_to_string** (double *val*, char *format_code*, int *precision*, int *flags*, int **ptype*)

Convierte un `double val` en una cadena de caracteres usando *format_code*, *precision* y *flags* suministrados.

format_code debe ser uno de `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` or `'r'`. Para `'r'`, la *precision* suministrada debe ser 0 y se ignora. El código de formato `'r'` especifica el formato estándar `repr()`.

flags puede ser cero o más de los valores `Py_DTST_SIGN`, `Py_DTST_ADD_DOT_0`, o `Py_DTST_ALT`, unidos por *or* (*or-ed*) juntos:

- `Py_DTST_SIGN` significa preceder siempre a la cadena de caracteres retornada con un carácter de signo, incluso si *val* no es negativo.
- `Py_DTST_ADD_DOT_0` significa asegurarse de que la cadena de caracteres retornada no se verá como un número entero.
- `Py_DTST_ALT` significa aplicar reglas de formato «alternativas». Consulte la documentación del especificador `PyOS_snprintf()` `"#"` para obtener más detalles.

Si *ptype* no es `NULL`, el valor al que apunta se establecerá en uno de `Py_DTST_FINITE`, `Py_DTST_INFINITE` o `Py_DTST_NAN`, lo que significa que *val* es un número finito, un número infinito o no es un número, respectivamente.

El valor de retorno es un puntero a *buffer* con la cadena de caracteres convertida o `NULL` si la conversión falla. La persona que llama es responsable de liberar la cadena de caracteres retornada llamando a `PyMem_Free()`.

Nuevo en la versión 3.1.

int **PyOS_stricmp** (const char **s1*, const char **s2*)

Comparación no sensible a mayúsculas y minúsculas en cadenas de caracteres. La función se comporta casi de manera idéntica a `strcmp()`, excepto que ignora el caso.

int **PyOS_strnicmp** (const char **s1*, const char **s2*, *Py_ssize_t* *size*)

Comparación no sensible a mayúsculas y minúsculas en cadenas de caracteres. La función se comporta casi de manera idéntica a `strncmp()`, excepto que ignora el caso.

6.8 Reflexión

*PyObject** **PyEval_GetBuiltins** (void)

Return value: Borrowed reference. Retorna un diccionario de las construcciones en el marco de ejecución actual, o el intérprete del estado del hilo si no se está ejecutando ningún marco actualmente.

*PyObject** **PyEval_GetLocals** (void)

Return value: Borrowed reference. Retorna un diccionario de las variables locales en el marco de ejecución actual, o NULL si actualmente no se está ejecutando ningún marco.

*PyObject** **PyEval_GetGlobals** (void)

Return value: Borrowed reference. Retorna un diccionario de las variables globales en el marco de ejecución actual, o NULL si actualmente no se está ejecutando ningún marco.

*PyFrameObject** **PyEval_GetFrame** (void)

Return value: Borrowed reference. Retorna el marco del estado del hilo actual, que es NULL si actualmente no se está ejecutando ningún marco.

Vea también *PyThreadState_GetFrame()*.

*PyFrameObject** **PyFrame_GetBack** (*PyFrameObject* *frame)

Obtiene el *frame* siguiente marco (*frame*) exterior.

Devuelve una referencia fuerte o NULL si *frame* no tiene un marco exterior.

frame no debe ser NULL.

Nuevo en la versión 3.9.

*PyCodeObject** **PyFrame_GetCode** (*PyFrameObject* *frame)

Obtiene el código *frame*.

Retorna una referencia fuerte.

frame no debe ser NULL. El resultado (código del marco) no puede ser NULL.

Nuevo en la versión 3.9.

int **PyFrame_GetLineNumber** (*PyFrameObject* *frame)

Retorna el número de línea que *frame* está ejecutando actualmente.

frame no debe ser NULL.

const char* **PyEval_GetFuncName** (*PyObject* *func)

Retorna el nombre de *func* si es una función, clase u objeto de instancia; de lo contrario, el nombre del tipo *funcs*.

const char* **PyEval_GetFuncDesc** (*PyObject* *func)

Retorna una cadena de caracteres de descripción, según el tipo de *func*. Los valores de retorno incluyen «()» para funciones y métodos, «constructor», «instancia» y «objeto». Concatenado con el resultado de *PyEval_GetFuncName()*, el resultado será una descripción de *func*.

6.9 Registro de códec y funciones de soporte

int **PyCodec_Register** (*PyObject* **search_function*)

Registra una nueva función de búsqueda de códec.

Como efecto secundario, intenta cargar el paquete `encodings`, si aún no lo ha hecho, para asegurarse de que siempre esté primero en la lista de funciones de búsqueda.

int **PyCodec_KnownEncoding** (const char **encoding*)

Retorna 1 o 0 dependiendo de si hay un códec registrado para el *encoding* dado. Esta función siempre finaliza con éxito.

*PyObject** **PyCodec_Encode** (*PyObject* **object*, const char **encoding*, const char **errors*)

Return value: New reference. API de codificación genérica basada en códec.

object se pasa a través de la función de codificador encontrada por el *encoding* dado usando el método de manejo de errores definido por *errors*. *errors* pueden ser NULL para usar el método predeterminado definido para el códec. Lanza un `LookupError` si no se puede encontrar el codificador.

*PyObject** **PyCodec_Decode** (*PyObject* **object*, const char **encoding*, const char **errors*)

Return value: New reference. API de decodificación basada en códec genérico.

object se pasa a través de la función de decodificador encontrada por el *encoding* dado usando el método de manejo de errores definido por *errors*. *errors* puede ser NULL para usar el método predeterminado definido para el códec. Lanza un `LookupError` si no se puede encontrar el codificador.

6.9.1 API de búsqueda de códec

En las siguientes funciones, la cadena de caracteres *encoding* se busca convertida a todos los caracteres en minúscula, lo que hace que las codificaciones se busquen a través de este mecanismo sin distinción entre mayúsculas y minúsculas. Si no se encuentra ningún códec, se establece un `KeyError` y se retorna NULL.

*PyObject** **PyCodec_Encoder** (const char **encoding*)

Return value: New reference. Obtiene una función de codificador para el *encoding* dado.

*PyObject** **PyCodec_Decoder** (const char **encoding*)

Return value: New reference. Obtiene una función de decodificador para el *encoding* dado.

*PyObject** **PyCodec_IncrementalEncoder** (const char **encoding*, const char **errors*)

Return value: New reference. Obtiene un objeto `IncrementalEncoder` para el *encoding* dado.

*PyObject** **PyCodec_IncrementalDecoder** (const char **encoding*, const char **errors*)

Return value: New reference. Obtiene un objeto `IncrementalDecoder` para el *encoding* dado.

*PyObject** **PyCodec_StreamReader** (const char **encoding*, *PyObject* **stream*, const char **errors*)

Return value: New reference. Obtiene una función de fábrica `StreamReader` para el *encoding* dado.

*PyObject** **PyCodec_StreamWriter** (const char **encoding*, *PyObject* **stream*, const char **errors*)

Return value: New reference. Obtiene una función de fábrica `StreamWriter` por el *encoding* dado.

6.9.2 API de registro para controladores de errores de codificación Unicode

int PyCodec_RegisterError (const char *name, PyObject *error)

Registra la función de devolución de llamada de manejo de errores *error* bajo el nombre *name* dado. Esta función de devolución de llamada será llamada por un códec cuando encuentre caracteres no codificables / bytes no codificables y *name* se especifica como parámetro de error en la llamada a la función de codificación / decodificación.

La devolución de llamada obtiene un único argumento, una instancia de `UnicodeEncodeError`, `UnicodeDecodeError` o `UnicodeTranslateError` que contiene información sobre la secuencia problemática de caracteres o bytes y su desplazamiento en la cadena original (consulte *Objetos Unicode de Excepción* para funciones para extraer esta información). La devolución de llamada debe generar la excepción dada o retornar una tupla de dos elementos que contiene el reemplazo de la secuencia problemática, y un número entero que proporciona el desplazamiento en la cadena original en la que se debe reanudar la codificación / decodificación.

Retorna 0 en caso de éxito, -1 en caso de error.

PyObject* PyCodec_LookupError (const char *name)

Return value: New reference. Busca la función de devolución de llamada de manejo de errores registrada con *name*. Como caso especial se puede pasar NULL, en cuyo caso se retornará la devolución de llamada de manejo de errores para «estricto».

PyObject* PyCodec_StrictErrors (PyObject *exc)

Return value: Always NULL. Lanza *exc* como una excepción.

PyObject* PyCodec_IgnoreErrors (PyObject *exc)

Return value: New reference. Ignora el error Unicode, omitiendo la entrada defectuosa.

PyObject* PyCodec_ReplaceErrors (PyObject *exc)

Return value: New reference. Reemplaza el error de codificación Unicode con ? o U+FFFD.

PyObject* PyCodec_XMLCharRefReplaceErrors (PyObject *exc)

Return value: New reference. Reemplaza el error de codificación Unicode con referencias de caracteres XML.

PyObject* PyCodec_BackslashReplaceErrors (PyObject *exc)

Return value: New reference. Reemplaza el error de codificación Unicode con escapes de barra invertida (\x, \u y \U).

PyObject* PyCodec_NameReplaceErrors (PyObject *exc)

Return value: New reference. Reemplaza el error de codificación Unicode con escapes \N{...}.

Nuevo en la versión 3.5.

Capa de objetos abstractos

Las funciones de este capítulo interactúan con los objetos de Python independientemente de su tipo, o con amplias clases de tipos de objetos (por ejemplo, todos los tipos numéricos o todos los tipos de secuencia). Cuando se usan en tipos de objetos para los que no se aplican, generarán una excepción de Python.

No es posible utilizar estas funciones en objetos que no se inicializan correctamente, como un objeto de lista que ha sido creado por `PyList_New()`, pero cuyos elementos no se han establecido en algunos valores no-“NULL” aún.

7.1 Protocolo de objeto

*PyObject** **Py_NotImplemented**

El singleton `NotImplemented`, se usa para indicar que una operación no está implementada para la combinación de tipos dada.

Py_RETURN_NOTIMPLEMENTED

Maneja adecuadamente el retorno *Py_NotImplemented* desde una función C (es decir, incrementa el recuento de referencias de *NotImplemented* y lo retorna).

int PyObject_Print (*PyObject* *o, FILE *fp, int flags)

Imprime un objeto *o*, en el archivo *fp*. Retorna -1 en caso de error. El argumento de las banderas se usa para habilitar ciertas opciones de impresión. La única opción actualmente admitida es `Py_PRINT_RAW`; si se proporciona, se escribe `str()` del objeto en lugar de `repr()`.

int PyObject_HasAttr (*PyObject* *o, *PyObject* *attr_name)

Retorna 1 si *o* tiene el atributo *attr_name*, y 0 en caso contrario. Esto es equivalente a la expresión de Python `hasattr(o, attr_name)`. Esta función siempre finaliza exitosamente.

Tenga en cuenta que las excepciones que se producen al llamar a los métodos a `__getattr__()` y `__getattribute__()` se suprimirán. Para obtener informe de errores, utilice *PyObject_GetAttr()* alternativamente.

int PyObject_HasAttrString (*PyObject* *o, const char *attr_name)

Retorna 1 si *o* tiene el atributo *attr_name*, y 0 en caso contrario. Esto es equivalente a la expresión de Python `hasattr(o, attr_name)`. Esta función siempre finaliza exitosamente.

Tenga en cuenta que las excepciones que se producen al llamar a `__getattr__()` y `__getattribute__()` y al crear un objeto de cadena temporal se suprimirán. Para obtener informes de errores, utilice `PyObject_GetAttrString()` en su lugar.

PyObject* PyObject_GetAttr (PyObject *o, PyObject *attr_name)

Return value: New reference. Recupera un atributo llamado `attr_name` del objeto `o`. Retorna el valor del atributo en caso de éxito o NULL en caso de error. Este es el equivalente de la expresión de Python `o.attr_name`.

PyObject* PyObject_GetAttrString (PyObject *o, const char *attr_name)

Return value: New reference. Recupera un atributo llamado `attr_name` del objeto `o`. Retorna el valor del atributo en caso de éxito o NULL en caso de error. Este es el equivalente de la expresión de Python `o.attr_name`.

PyObject* PyObject_GenericGetAttr (PyObject *o, PyObject *name)

Return value: New reference. Función *getter* de atributo genérico que debe colocarse en la ranura `tp_getattro` de un objeto tipo. Busca un descriptor en el diccionario de clases en el MRO del objeto, así como un atributo en el objeto `__dict__` (si está presente). Como se describe en `descriptors`, los descriptors de datos tienen preferencia sobre los atributos de instancia, mientras que los descriptors que no son de datos no lo hacen. De lo contrario, se genera un `AttributeError`.

int PyObject_SetAttr (PyObject *o, PyObject *attr_name, PyObject *v)

Establece el valor del atributo llamado `attr_name`, para el objeto `o`, en el valor `v`. Genera una excepción y retorna `-1` en caso de falla; retorna `0` en caso de éxito. Este es el equivalente de la declaración de Python `o.attr_name = v`.

If `v` is NULL, the attribute is deleted. This behaviour is deprecated in favour of using `PyObject_DelAttr()`, but there are currently no plans to remove it.

int PyObject_SetAttrString (PyObject *o, const char *attr_name, PyObject *v)

Establece el valor del atributo llamado `attr_name`, para el objeto `o`, en el valor `v`. Genera una excepción y retorna `-1` en caso de falla; retorna `0` en caso de éxito. Este es el equivalente de la declaración de Python `o.attr_name = v`.

If `v` is NULL, the attribute is deleted, but this feature is deprecated in favour of using `PyObject_DelAttrString()`.

int PyObject_GenericSetAttr (PyObject *o, PyObject *name, PyObject *value)

Establecimiento de atributo genérico y función de eliminación que está destinada a colocarse en la ranura de un objeto tipo `tp_setattro`. Busca un descriptor de datos en el diccionario de clases en el MRO del objeto y, si se encuentra, tiene preferencia sobre la configuración o eliminación del atributo en el diccionario de instancias. De lo contrario, el atributo se establece o elimina en el objeto `__dict__` (si está presente). En caso de éxito, se retorna `0`; de lo contrario, se genera un `AttributeError` y se retorna `-1`.

int PyObject_DelAttr (PyObject *o, PyObject *attr_name)

Elimina el atributo llamado `attr_name`, para el objeto `o`. Retorna `-1` en caso de falla. Este es el equivalente de la declaración de Python `del o.attr_name`.

int PyObject_DelAttrString (PyObject *o, const char *attr_name)

Elimina el atributo llamado `attr_name`, para el objeto `o`. Retorna `-1` en caso de falla. Este es el equivalente de la declaración de Python `del o.attr_name`.

PyObject* PyObject_GenericGetDict (PyObject *o, void *context)

Return value: New reference. Una implementación genérica para obtener un descriptor `__dict__`. Crea el diccionario si es necesario.

Nuevo en la versión 3.3.

int PyObject_GenericSetDict (PyObject *o, PyObject *value, void *context)

Una implementación genérica para el creador de un descriptor `__dict__`. Esta implementación no permite que se elimine el diccionario.

Nuevo en la versión 3.3.

*PyObject** **PyObject_RichCompare** (*PyObject* *o1, *PyObject* *o2, int opid)

Return value: New reference. Compara los valores de *o1* y *o2* utilizando la operación especificada por *opid*, que debe ser uno de los siguientes `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, o `Py_GE`, correspondiente a `<`, `<=`, `==`, `!=`, `>` o `>=` respectivamente. Este es el equivalente de la expresión de Python `o1 op o2`, donde `op` es el operador correspondiente a *opid*. Retorna el valor de la comparación en caso de éxito o `NULL` en caso de error.

int **PyObject_RichCompareBool** (*PyObject* *o1, *PyObject* *o2, int opid)

Compara los valores de *o1* y *o2* utilizando la operación especificada por *opid*, que debe ser uno de los siguientes `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, o `Py_GE`, correspondiente a `<`, `<=`, `==`, `!=`, `>` o `>=` respectivamente. Retorna `-1` en caso de error, `0` si el resultado es falso, `1` en caso contrario. Este es el equivalente de la expresión de Python `o1 op o2`, donde `op` es el operador correspondiente a *opid*.

Nota: Si *o1* y *o2* son el mismo objeto, `PyObject_RichCompareBool()` siempre retornará `1` para `Py_EQ` y `0` para `Py_NE`.

*PyObject** **PyObject_Repr** (*PyObject* *o)

Return value: New reference. Calcula una representación de cadena de caracteres del objeto *o*. Retorna la representación de cadena de caracteres en caso de éxito, `NULL` en caso de error. Este es el equivalente de la expresión de Python `repr(o)`. Llamado por la función incorporada `repr()`.

Distinto en la versión 3.4: Esta función ahora incluye una afirmación de depuración para ayudar a garantizar que no descarte silenciosamente una excepción activa.

*PyObject** **PyObject_ASCII** (*PyObject* *o)

Return value: New reference. Como `PyObject_Repr()`, calcula una representación de cadena de caracteres del objeto *o*, pero escapa los caracteres no ASCII en la cadena de caracteres retornada por `PyObject_Repr()` con `\x`, `\u` o `\U` escapa. Esto genera una cadena de caracteres similar a la que retorna `PyObject_Repr()` en Python 2. Llamado por la función incorporada `ascii()`.

*PyObject** **PyObject_Str** (*PyObject* *o)

Return value: New reference. Calcula una representación de cadena de caracteres del objeto *o*. Retorna la representación de cadena de caracteres en caso de éxito, `NULL` en caso de error. Llamado por la función incorporada `str()` y, por lo tanto, por la función `print()`.

Distinto en la versión 3.4: Esta función ahora incluye una afirmación de depuración para ayudar a garantizar que no descarte silenciosamente una excepción activa.

*PyObject** **PyObject_Bytes** (*PyObject* *o)

Return value: New reference. Calcula una representación de bytes del objeto *o*. `NULL` se retorna en caso de error y un objeto de bytes en caso de éxito. Esto es equivalente a la expresión de Python `bytes(o)`, cuando *o* no es un número entero. A diferencia de `bytes(o)`, se lanza un `TypeError` cuando *o* es un entero en lugar de un objeto de bytes con inicialización cero.

int **PyObject_IsSubclass** (*PyObject* *derived, *PyObject* *cls)

Retorna `1` si la clase *derived* es idéntica o derivada de la clase *cls*; de lo contrario, retorna `0`. En caso de error, retorna `-1`.

Si *cls* es una tupla, la verificación se realizará con cada entrada en *cls*. El resultado será `1` cuando al menos una de las verificaciones retorne `1`, de lo contrario será `0`.

Si *cls* tiene un método `__subclasscheck__()`, se llamará para determinar el estado de la subclase como se describe en [PEP 3119](#). De lo contrario, *derived* es una subclase de *cls* si es una subclase directa o indirecta, es decir, contenida en `cls.__mro__`.

Normalmente, solo los objetos clase, es decir, las instancias de `type` o una clase derivada, se consideran clases. Sin embargo, los objetos pueden anular esto al tener un atributo `__bases__` (que debe ser una tupla de clases base).

int PyObject_IsInstance (*PyObject* *inst, *PyObject* *cls)

Retorna 1 si *inst* es una instancia de la clase *cls* o una subclase de *cls*, o 0 si no. En caso de error, retorna -1 y establece una excepción.

Si *cls* es una tupla, la verificación se realizará con cada entrada en *cls*. El resultado será 1 cuando al menos una de las verificaciones retorne 1, de lo contrario será 0.

Si *cls* tiene un método `__instancecheck__()`, se llamará para determinar el estado de la subclase como se describe en [PEP 3119](#). De lo contrario, *inst* es una instancia de *cls* si su clase es una subclase de *cls*.

Una instancia *inst* puede anular lo que se considera su clase al tener un atributo `__class__`.

Un objeto *cls* puede anular si se considera una clase y cuáles son sus clases base, al tener un atributo `__bases__` (que debe ser una tupla de clases base).

Py_hash_t PyObject_Hash (*PyObject* *o)

Calcula y retorna el valor hash de un objeto *o*. En caso de fallo, retorna -1. Este es el equivalente de la expresión de Python `hash(o)`.

Distinto en la versión 3.2: The return type is now `Py_hash_t`. This is a signed integer the same size as `Py_ssize_t`.

Py_hash_t PyObject_HashNotImplemented (*PyObject* *o)

Establece un `TypeError` indicando que `type(o)` no es *hashable* y retorna -1. Esta función recibe un tratamiento especial cuando se almacena en una ranura `tp_hash`, lo que permite que un tipo indique explícitamente al intérprete que no es *hashable*.

int PyObject_IsTrue (*PyObject* *o)

Retorna 1 si el objeto *o* se considera verdadero y 0 en caso contrario. Esto es equivalente a la expresión de Python `not not o`. En caso de error, retorna -1.

int PyObject_Not (*PyObject* *o)

Retorna 0 si el objeto *o* se considera verdadero, y 1 de lo contrario. Esto es equivalente a la expresión de Python `not o`. En caso de error, retorna -1.

*PyObject** **PyObject_Type** (*PyObject* *o)

Return value: New reference. When *o* is non-NULL, returns a type object corresponding to the object type of object *o*. On failure, raises `SystemError` and returns NULL. This is equivalent to the Python expression `type(o)`. This function increments the reference count of the return value. There's really no reason to use this function instead of the `Py_TYPE()` function, which returns a pointer of type `PyTypeObject*`, except when the incremented reference count is needed.

int PyObject_TypeCheck (*PyObject* *o, *PyTypeObject* *type)

Retorna verdadero si el objeto *o* es de tipo *type* o un subtipo de *type*. Ambos parámetros no deben ser NULL.

Py_ssize_t **PyObject_Size** (*PyObject* *o)

Py_ssize_t **PyObject_Length** (*PyObject* *o)

Retorna la longitud del objeto *o*. Si el objeto *o* proporciona los protocolos de secuencia y mapeo, se retorna la longitud de la secuencia. En caso de error, se retorna -1. Este es el equivalente a la expresión de Python `len(o)`.

Py_ssize_t **PyObject_LengthHint** (*PyObject* *o, *Py_ssize_t* defaultvalue)

Return an estimated length for the object *o*. First try to return its actual length, then an estimate using `__length_hint__()`, and finally return the default value. On error return -1. This is the equivalent to the Python expression `operator.length_hint(o, defaultvalue)`.

Nuevo en la versión 3.4.

*PyObject** **PyObject_GetItem** (*PyObject* *o, *PyObject* *key)

Return value: New reference. Retorna el elemento de *o* correspondiente a la clave *key* del objeto o NULL en caso de error. Este es el equivalente de la expresión de Python `o[key]`.

`int PyObject_SetItem(PyObject *o, PyObject *key, PyObject *v)`

Asigna el objeto *key* al valor *v*. Genera una excepción y retorna `-1` en caso de error; retorna `0` en caso de éxito. Este es el equivalente de la declaración de Python `o[key] = v`. Esta función *no* roba una referencia a *v*.

`int PyObject_DelItem(PyObject *o, PyObject *key)`

Elimina la asignación para el objeto *key* del objeto *o*. Retorna `-1` en caso de falla. Esto es equivalente a la declaración de Python `del o[key]`.

*PyObject** `PyObject_Dir(PyObject *o)`

Return value: *New reference*. Esto es equivalente a la expresión de Python `dir(o)`, que retorna una lista (posiblemente vacía) de cadenas de caracteres apropiadas para el argumento del objeto, o `NULL` si hubo un error. Si el argumento es `NULL`, es como el Python `dir()`, que retorna los nombres de los locales actuales; en este caso, si no hay un marco de ejecución activo, se retorna `NULL` pero `PyErr_Occurred()` retornará falso.

*PyObject** `PyObject_GetIter(PyObject *o)`

Return value: *New reference*. Esto es equivalente a la expresión de Python `iter(o)`. Retorna un nuevo iterador para el argumento del objeto, o el propio objeto si el objeto ya es un iterador. Lanza `TypeError` y retorna `NULL` si el objeto no puede iterarse.

7.2 Protocolo de llamada

CPython admite dos protocolos de llamada diferentes: *tp_call* y *vectorcall*.

7.2.1 El protocolo *tp_call*

Las instancias de clases que establecen *tp_call* son invocables. La firma del slot es:

```
PyObject *tp_call(PyObject *callable, PyObject *args, PyObject *kwargs);
```

Se realiza una llamada usando una tupla para los argumentos posicionales y un dict para los argumentos de palabras clave, de manera similar a `callable(*args, **kwargs)` en el código Python. *args* debe ser no `NULL` (use una tupla vacía si no hay argumentos) pero *kwargs* puede ser `NULL` si no hay argumentos de palabra clave.

Esta convención no solo es utilizada por *tp_call*: *tp_new* y *tp_init* también pasan argumentos de esta manera.

To call an object, use `PyObject_Call()` or another *call API*.

7.2.2 El protocolo *vectorcall*

Nuevo en la versión 3.9.

El protocolo *vectorcall* se introdujo en **PEP 590** como un protocolo adicional para hacer que las llamadas sean más eficientes.

Como regla general, CPython preferirá el *vectorcall* para llamadas internas si el invocable lo admite. Sin embargo, esta no es una regla estricta. Además, algunas extensiones de terceros usan *tp_call* directamente (en lugar de usar `PyObject_Call()`). Por lo tanto, una clase que admita *vectorcall* también debe implementar *tp_call*. Además, el invocable debe comportarse de la misma manera independientemente del protocolo que se utilice. La forma recomendada de lograr esto es configurando *tp_call* en `PyVectorcall_Call()`. Vale la pena repetirlo:

Advertencia: Una clase que admita *vectorcall* **debe** también implementar *tp_call* con la misma semántica.

Una clase no debería implementar `vectorcall` si eso fuera más lento que `tp_call`. Por ejemplo, si el destinatario de la llamada necesita convertir los argumentos a una tupla `args` y un dict `kwargs` de todos modos, entonces no tiene sentido implementar `vectorcall`.

Las clases pueden implementar el protocolo `vectorcall` habilitando el indicador `Py_TPFLAGS_HAVE_VECTORCALL` y la configuración `tp_vectorcall_offset` al desplazamiento dentro de la estructura del objeto donde aparece un `vectorcallfunc`. Este es un puntero a una función con la siguiente firma:

```
PyObject* (*vectorcallfunc) (PyObject* callable, PyObject* const *args, size_t nargsf, PyObject* kwnames)
```

- `callable` es el objeto siendo invocado.
- `args` es un arreglo en C que consta de los argumentos posicionales seguidos por el valores de los argumentos de la palabra clave. Puede ser `NULL` si no hay argumentos.
- `nargsf` es el número de argumentos posicionales más posiblemente el flag `PY_VECTORCALL_ARGUMENTS_OFFSET`. Para obtener el número real de argumentos posicionales de `nargsf`, use `PyVectorcall_NARGS()`.
- `kwnames` es una tupla que contiene los nombres de los argumentos de la palabra clave; en otras palabras, las claves del diccionario `kwargs`. Estos nombres deben ser cadenas (instancias de `str` o una subclase) y deben ser únicos. Si no hay argumentos de palabras clave, entonces `kwnames` puede ser `NULL`.

PY_VECTORCALL_ARGUMENTS_OFFSET

Si este flag se establece en un argumento `vectorcall` `nargsf`, el destinatario de la llamada puede cambiar temporalmente `args[-1]`. En otras palabras, `args` apunta al argumento 1 (no 0) en el vector asignado. El destinatario de la llamada debe restaurar el valor de `args[-1]` antes de regresar.

Para `PyObject_VectorcallMethod()`, este flag significa en cambio que `args[0]` puede cambiarse.

Siempre que puedan hacerlo de forma económica (sin asignación adicional), se anima a las personas que llaman a utilizar `PY_VECTORCALL_ARGUMENTS_OFFSET`. Si lo hace, permitirá que las personas que llaman, como los métodos enlazados, realicen sus llamadas posteriores (que incluyen un argumento *self* antepuesto) de manera muy eficiente.

Para llamar a un objeto que implementa `vectorcall`, use una función *call API* como con cualquier otro invocable. `PyObject_Vectorcall()` normalmente será más eficiente.

Nota: En CPython 3.8, la API de `vectorcall` y las funciones relacionadas estaban disponibles provisionalmente bajo nombres con un guión bajo inicial: `_PyObject_Vectorcall`, `_Py_TPFLAGS_HAVE_VECTORCALL`, `_PyObject_VectorcallMethod`, `_PyVectorcall_Function`, `_PyObject_CallMethodNoArgs`, `_PyObject_CallMethodOneArg`. Además, `PyObject_VectorcallDict` estaba disponible como `_PyObject_FastCallDict`. Los nombres antiguos todavía se definen como alias de los nuevos nombres no subrayados.

Control de recursión

Cuando se usa `tp_call`, los destinatarios no necesitan preocuparse por *recursividad*: CPython usa `Py_EnterRecursiveCall()` y `Py_LeaveRecursiveCall()` para llamadas realizadas usando `tp_call`.

Por eficiencia, este no es el caso de las llamadas realizadas mediante `vectorcall`: el destinatario de la llamada debe utilizar `Py_EnterRecursiveCall` y `Py_LeaveRecursiveCall` si es necesario.

API de soporte para vectorcall

Py_ssize_t **PyVectorcall_NARGS** (*size_t nargsf*)

Dado un argumento vectorcall *nargsf*, retorna el número real de argumentos. Actualmente equivalente a:

```
(Py_ssize_t) (nargsf & ~PY_VECTORCALL_ARGUMENTS_OFFSET)
```

Sin embargo, la función `PyVectorcall_NARGS` debe usarse para permitir futuras extensiones.

Esta función no es parte de la *API limitada*.

Nuevo en la versión 3.8.

vectorcallfunc **PyVectorcall_Function** (*PyObject *op*)

Si *op* no admite el protocolo vectorcall (ya sea porque el tipo no lo hace o porque la instancia específica no lo hace), retorna `NULL`. De lo contrario, retorna el puntero de la función vectorcall almacenado en *op*. Esta función nunca lanza una excepción.

Esto es principalmente útil para verificar si *op* admite vectorcall, lo cual se puede hacer marcando `PyVectorcall_Function(op) != NULL`.

Esta función no es parte de la *API limitada*.

Nuevo en la versión 3.8.

*PyObject** **PyVectorcall_Call** (*PyObject *callable*, *PyObject *tuple*, *PyObject *dict*)

Llama a la *vectorcallfunc* de *callable* con argumentos posicionales y de palabras clave dados en una tupla y dict, respectivamente.

Esta es una función especializada, destinada a colocarse en el slot `tp_call` o usarse en una implementación de `tp_call`. No comprueba el flag `Py_TPFLAGS_HAVE_VECTORCALL` y no vuelve a `tp_call`.

Esta función no es parte de la *API limitada*.

Nuevo en la versión 3.8.

7.2.3 API para invocar objetos

Various functions are available for calling a Python object. Each converts its arguments to a convention supported by the called object – either `tp_call` or vectorcall. In order to do as little conversion as possible, pick one that best fits the format of data you have available.

La siguiente tabla resume las funciones disponibles; consulte la documentación individual para obtener más detalles.

Función	invocable	args	kwargs
<code>PyObject_Call()</code>	<code>PyObject *</code>	tupla	dict/ <code>NULL</code>
<code>PyObject_CallNoArgs()</code>	<code>PyObject *</code>	—	—
<code>PyObject_CallOneArg()</code>	<code>PyObject *</code>	1 objeto	—
<code>PyObject_CallObject()</code>	<code>PyObject *</code>	tupla/ <code>NULL</code>	—
<code>PyObject_CallFunction()</code>	<code>PyObject *</code>	formato	—
<code>PyObject_CallMethod()</code>	<code>obj + char*</code>	formato	—
<code>PyObject_CallFunctionObjArgs()</code>	<code>PyObject *</code>	variadica	—
<code>PyObject_CallMethodObjArgs()</code>	<code>obj + nombre</code>	variadica	—
<code>PyObject_CallMethodNoArgs()</code>	<code>obj + nombre</code>	—	—
<code>PyObject_CallMethodOneArg()</code>	<code>obj + nombre</code>	1 objeto	—
<code>PyObject_Vectorcall()</code>	<code>PyObject *</code>	vectorcall	vectorcall
<code>PyObject_VectorcallDict()</code>	<code>PyObject *</code>	vectorcall	dict/ <code>NULL</code>
<code>PyObject_VectorcallMethod()</code>	<code>arg + nombre</code>	vectorcall	vectorcall

*PyObject** **PyObject_Call** (*PyObject* *callable, *PyObject* *args, *PyObject* *kwargs)

Return value: New reference. Llama a un objeto de Python invocable *callable*, con argumentos dados por la tupla *args*, y argumentos con nombre dados por el diccionario *kwargs*.

args no debe ser *NULL*; use una tupla vacía si no se necesitan argumentos. Si no se necesitan argumentos con nombre, *kwargs* puede ser *NULL*.

Retorna el resultado de la llamada en caso de éxito o genera una excepción y retorna *NULL* en caso de error.

Este es el equivalente de la expresión de Python: `callable(*args, **kwargs)`.

*PyObject** **PyObject_CallNoArgs** (*PyObject* *callable)

Llama a un objeto de Python invocable *callable* sin ningún argumento. Es la forma más eficiente de llamar a un objeto Python invocable sin ningún argumento.

Retorna el resultado de la llamada en caso de éxito o genera una excepción y retorna *NULL* en caso de error.

Nuevo en la versión 3.9.

*PyObject** **PyObject_CallOneArg** (*PyObject* *callable, *PyObject* *arg)

Llama a un objeto de Python invocable *callable* con exactamente 1 argumento posicional *arg* y sin argumentos de palabra clave.

Retorna el resultado de la llamada en caso de éxito o genera una excepción y retorna *NULL* en caso de error.

Esta función no es parte de la *API limitada*.

Nuevo en la versión 3.9.

*PyObject** **PyObject_CallObject** (*PyObject* *callable, *PyObject* *args)

Return value: New reference. Llama a un objeto de Python invocable *callable*, con argumentos dados por la tupla *args*. Si no se necesitan argumentos, entonces *args* puede ser *NULL*.

Retorna el resultado de la llamada en caso de éxito o genera una excepción y retorna *NULL* en caso de error.

Este es el equivalente de la expresión de Python: `callable(*args)`.

*PyObject** **PyObject_CallFunction** (*PyObject* *callable, const char *format, ...)

Return value: New reference. Llama a un objeto de Python invocable *callable*, con un número variable de argumentos C. Los argumentos de C se describen usando una cadena de caracteres de formato de estilo *Py_BuildValue()*. El formato puede ser *NULL*, lo que indica que no se proporcionan argumentos.

Retorna el resultado de la llamada en caso de éxito o genera una excepción y retorna *NULL* en caso de error.

Este es el equivalente de la expresión de Python: `callable(*args)`.

Tenga en cuenta que si solo pasa *PyObject* *args, *PyObject_CallFunctionObjArgs()* es una alternativa más rápida.

Distinto en la versión 3.4: El tipo de *format* se cambió desde `char *`.

*PyObject** **PyObject_CallMethod** (*PyObject* *obj, const char *name, const char *format, ...)

Return value: New reference. Llama al método llamado *name* del objeto *obj* con un número variable de argumentos en C. Los argumentos de C se describen mediante una cadena de formato *Py_BuildValue()* que debería producir una tupla.

El formato puede ser *NULL*, lo que indica que no se proporcionan argumentos.

Retorna el resultado de la llamada en caso de éxito o genera una excepción y retorna *NULL* en caso de error.

Este es el equivalente de la expresión de Python: `obj.name(arg1, arg2, ...)`.

Tenga en cuenta que si solo pasa *PyObject* *args, *PyObject_CallMethodObjArgs()* es una alternativa más rápida.

Distinto en la versión 3.4: Los tipos de *name* y *format* se cambiaron desde `char *`.

*PyObject** **PyObject_CallFunctionObjArgs** (*PyObject* *callable, ...)

Return value: *New reference.* Llama a un objeto de Python invocable *callable*, con un número variable de argumentos *PyObject* *. Los argumentos se proporcionan como un número variable de parámetros seguidos de *NULL*.

Retorna el resultado de la llamada en caso de éxito o genera una excepción y retorna *NULL* en caso de error.

Este es el equivalente de la expresión de Python: `callable(arg1, arg2, ...)`.

*PyObject** **PyObject_CallMethodObjArgs** (*PyObject* *obj, *PyObject* *name, ...)

Return value: *New reference.* Llama a un método del objeto de Python *obj*, donde el nombre del método se proporciona como un objeto de cadena de caracteres de Python en *name*. Se llama con un número variable de argumentos *PyObject* *. Los argumentos se proporcionan como un número variable de parámetros seguidos de *NULL*.

Retorna el resultado de la llamada en caso de éxito o genera una excepción y retorna *NULL* en caso de error.

*PyObject** **PyObject_CallMethodNoArgs** (*PyObject* *obj, *PyObject* *name)

Llama a un método del objeto de Python *obj* sin argumentos, donde el nombre del método se da como un objeto de cadena de caracteres de Python en *name*.

Retorna el resultado de la llamada en caso de éxito o genera una excepción y retorna *NULL* en caso de error.

Esta función no es parte de la *API limitada*.

Nuevo en la versión 3.9.

*PyObject** **PyObject_CallMethodOneArg** (*PyObject* *obj, *PyObject* *name, *PyObject* *arg)

Llama a un método del objeto de Python *obj* con un único argumento posicional *arg*, donde el nombre del método se proporciona como un objeto de cadena de caracteres de Python en *name*.

Retorna el resultado de la llamada en caso de éxito o genera una excepción y retorna *NULL* en caso de error.

Esta función no es parte de la *API limitada*.

Nuevo en la versión 3.9.

*PyObject** **PyObject_Vectorcall** (*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kw-names)

Llama a un objeto de Python invocable *callable*. Los argumentos son los mismos que para *vectorcallfunc*. Si *callable* admite *vectorcall*, esto llama directamente a la función *vectorcall* almacenada en *callable*.

Retorna el resultado de la llamada en caso de éxito o genera una excepción y retorna *NULL* en caso de error.

Esta función no es parte de la *API limitada*.

Nuevo en la versión 3.9.

*PyObject** **PyObject_VectorcallDict** (*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwdict)

Llamada *invocable* con argumentos posicionales pasados exactamente como en el protocolo *vectorcall*, pero con argumentos de palabras clave pasados como un diccionario *kwdict*. El arreglo *args* contiene solo los argumentos posicionales.

Independientemente del protocolo que se utilice internamente, es necesario realizar una conversión de argumentos. Por lo tanto, esta función solo debe usarse si la persona que llama ya tiene un diccionario listo para usar para los argumentos de palabras clave, pero no una tupla para los argumentos posicionales.

Esta función no es parte de la *API limitada*.

Nuevo en la versión 3.9.

*PyObject** **PyObject_VectorcallMethod** (*PyObject* *name, *PyObject* *const *args, size_t nargsf, *PyObject* *kw-names)

Llama a un método usando la convención de llamada *vectorcall*. El nombre del método se proporciona como una cadena de Python *name*. El objeto cuyo método se llama es *args[0]*, y el arreglo *args* que comienza en *args*

[1] representa los argumentos de la llamada. Debe haber al menos un argumento posicional. *nargsf* es el número de argumentos posicionales que incluyen *args* [0], más `PY_VECTORCALL_ARGUMENTS_OFFSET` si el valor de *args* [0] puede cambiarse temporalmente. Los argumentos de palabras clave se pueden pasar como en `PyObject_Vectorcall()`.

Si el objeto tiene la característica `Py_TPFLAGS_METHOD_DESCRIPTOR`, esto llamará al objeto de método independiente con el vector *args* completo como argumentos.

Retorna el resultado de la llamada en caso de éxito o genera una excepción y retorna `NULL` en caso de error.

Esta función no es parte de la *API limitada*.

Nuevo en la versión 3.9.

7.2.4 API de soporte de llamadas

`int PyCallable_Check (PyObject *o)`

Determina si el objeto *o* es invocable. Retorna 1 si el objeto es invocable y 0 en caso contrario. Esta función siempre finaliza con éxito.

7.3 Protocolo de números

`int PyNumber_Check (PyObject *o)`

Retorna 1 si el objeto *o* proporciona protocolos numéricos, y falso en caso contrario. Esta función siempre finaliza con éxito.

Distinto en la versión 3.8: Retorna 1 si *o* es un índice entero.

`PyObject* PyNumber_Add (PyObject *o1, PyObject *o2)`

Return value: New reference. Retorna el resultado de agregar *o1* y *o2*, o `NULL` en caso de falla. Este es el equivalente de la expresión de Python `o1 + o2`.

`PyObject* PyNumber_Subtract (PyObject *o1, PyObject *o2)`

Return value: New reference. Retorna el resultado de restar *o2* de *o1*, o `NULL` en caso de falla. Este es el equivalente de la expresión de Python `o1 - o2`.

`PyObject* PyNumber_Multiply (PyObject *o1, PyObject *o2)`

Return value: New reference. Retorna el resultado de multiplicar *o1* y *o2*, o `NULL` en caso de error. Este es el equivalente de la expresión de Python `o1 * o2`.

`PyObject* PyNumber_MatrixMultiply (PyObject *o1, PyObject *o2)`

Return value: New reference. Retorna el resultado de la multiplicación de matrices en *o1* y *o2*, o `NULL` en caso de falla. Este es el equivalente de la expresión de Python `o1 @ o2`.

Nuevo en la versión 3.5.

`PyObject* PyNumber_FloorDivide (PyObject *o1, PyObject *o2)`

Return value: New reference. Return the floor of *o1* divided by *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 // o2`.

`PyObject* PyNumber_TrueDivide (PyObject *o1, PyObject *o2)`

Return value: New reference. Return a reasonable approximation for the mathematical value of *o1* divided by *o2*, or `NULL` on failure. The return value is «approximate» because binary floating point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers. This is the equivalent of the Python expression `o1 / o2`.

*PyObject** **PyNumber_Remainder** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el resto de dividir *o1* entre *o2* o NULL en caso de error. Este es el equivalente de la expresión de Python `o1 % o2`.

*PyObject** **PyNumber_Divmod** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Vea la función incorporada `divmod()`. Retorna NULL en caso de falla. Este es el equivalente de la expresión de Python `divmod(o1, o2)`.

*PyObject** **PyNumber_Power** (*PyObject* **o1*, *PyObject* **o2*, *PyObject* **o3*)

Return value: *New reference.* Consulte la función incorporada `pow()`. Retorna NULL en caso de falla. Este es el equivalente de la expresión de Python `pow(o1, o2, o3)`, donde *o3* es opcional. Si se ignora *o3*, pase *Py_None* en su lugar (pasar NULL por *o3* provocaría un acceso ilegal a la memoria).

*PyObject** **PyNumber_Negative** (*PyObject* **o*)

Return value: *New reference.* Retorna la negación de *o* en caso de éxito o NULL en caso de error. Este es el equivalente de la expresión de Python `-o`.

*PyObject** **PyNumber_Positive** (*PyObject* **o*)

Return value: *New reference.* Retorna *o* en caso de éxito o NULL en caso de error. Este es el equivalente de la expresión de Python `+o`.

*PyObject** **PyNumber_Absolute** (*PyObject* **o*)

Return value: *New reference.* Retorna el valor absoluto de *o* o NULL en caso de error. Este es el equivalente de la expresión de Python `abs(o)`.

*PyObject** **PyNumber_Invert** (*PyObject* **o*)

Return value: *New reference.* Retorna la negación bit a bit de *o* en caso de éxito o NULL en caso de error. Este es el equivalente de la expresión de Python `~o`.

*PyObject** **PyNumber_Lshift** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el resultado del desplazamiento a la izquierda *o1* por *o2* en caso de éxito o NULL en caso de error. Este es el equivalente de la expresión de Python `o1 << o2`.

*PyObject** **PyNumber_Rshift** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el resultado del desplazamiento a la derecha *o1* por *o2* en caso de éxito o NULL en caso de error. Este es el equivalente de la expresión de Python `o1 >> o2`.

*PyObject** **PyNumber_And** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el «bit a bit y» (*bitwise and*) de *o1* y *o2* en caso de éxito y NULL en caso de error. Este es el equivalente de la expresión de Python `o1 & o2`.

*PyObject** **PyNumber_Xor** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el «bit a bit o exclusivo» (*bitwise exclusive or*) de *o1* por *o2* en caso de éxito, o NULL en caso de error. Este es el equivalente de la expresión de Python `o1 ^ o2`.

*PyObject** **PyNumber_Or** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el «bit a bit o» (*bitwise or*) de *o1* y *o2* en caso de éxito, o NULL en caso de error. Este es el equivalente de la expresión de Python `o1 | o2`.

*PyObject** **PyNumber_InPlaceAdd** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el resultado de agregar *o1* y *o2*, o NULL en caso de falla. La operación se realiza en su lugar (*in-place*) cuando *o1* lo admite. Este es el equivalente de la declaración de Python `o1 += o2`.

*PyObject** **PyNumber_InPlaceSubtract** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el resultado de restar *o2* de *o1*, o NULL en caso de falla. La operación se realiza en su lugar (*in-place*) cuando *o1* lo admite. Este es el equivalente de la declaración de Python `o1 -= o2`.

*PyObject** **PyNumber_InPlaceMultiply** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el resultado de multiplicar *o1* y *o2*, o NULL en caso de error. La operación se realiza en su lugar (*in-place*) cuando *o1* lo admite. Este es el equivalente de la declaración de Python `o1 *= o2`.

*PyObject** **PyNumber_InPlaceMatrixMultiply** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el resultado de la multiplicación de matrices en *o1* y *o2*, o NULL en caso de falla. La operación se realiza en su lugar (*in-place*) cuando *o1* lo admite. Este es el equivalente de la declaración de Python `o1 @= o2`.

Nuevo en la versión 3.5.

*PyObject** **PyNumber_InPlaceFloorDivide** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el piso matemático de dividir *o1* por *o2*, o NULL en caso de falla. La operación se realiza en su lugar (*in-place*) cuando *o1* lo admite. Este es el equivalente de la declaración de Python `o1 //= o2`.

*PyObject** **PyNumber_InPlaceTrueDivide** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Return a reasonable approximation for the mathematical value of *o1* divided by *o2*, or NULL on failure. The return value is «approximate» because binary floating point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 /= o2`.

*PyObject** **PyNumber_InPlaceRemainder** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el resto de dividir *o1* entre *o2* o NULL en caso de error. La operación se realiza en su lugar (*in-place*) cuando *o1* lo admite. Este es el equivalente de la declaración de Python `o1 %= o2`.

*PyObject** **PyNumber_InPlacePower** (*PyObject* **o1*, *PyObject* **o2*, *PyObject* **o3*)

Return value: *New reference.* Consulte la función incorporada `pow()`. Retorna NULL en caso de falla. La operación se realiza en su lugar (*in-place*) cuando *o1* lo admite. Este es el equivalente de la declaración de Python `o1 **= o2` cuando *o3* es `Py_None`, o una variante en su lugar (*in-place*) de `pow(o1, o2, o3)` de lo contrario. Si se ignora *o3*, pase `Py_None` en su lugar (pasar NULL para *o3* provocaría un acceso ilegal a la memoria).

*PyObject** **PyNumber_InPlaceLshift** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el resultado del desplazamiento a la izquierda *o1* por *o2* en caso de éxito o NULL en caso de error. La operación se realiza en su sitio (*in-place*) cuando *o1* lo admite. Este es el equivalente de la declaración de Python `o1 <<= o2`.

*PyObject** **PyNumber_InPlaceRshift** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el resultado del desplazamiento a la derecha *o1* por *o2* en caso de éxito o NULL en caso de error. La operación se realiza en su lugar (*in-place*) cuando *o1* lo admite. Este es el equivalente de la declaración de Python `o1 >>= o2`.

*PyObject** **PyNumber_InPlaceAnd** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el «bit a bit y» (*bitwise and*) de *o1* y *o2* en caso de éxito y NULL en caso de error. La operación se realiza en su lugar (*in-place*) cuando *o1* lo admite. Este es el equivalente de la declaración de Python `o1 &= o2`.

*PyObject** **PyNumber_InPlaceXor** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el «bit a bit o exclusivo» (*bitwise exclusive or*) de *o1* por *o2* en caso de éxito, o NULL en caso de error. La operación se realiza en su lugar (*in-place*) cuando *o1* lo admite. Este es el equivalente de la declaración de Python `o1 ^= o2`.

*PyObject** **PyNumber_InPlaceOr** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Retorna el «bit a bit o» (*bitwise or*) de *o1* y *o2* en caso de éxito, o NULL en caso de error. La operación se realiza en su lugar *in-place* cuando *o1* lo admite. Este es el equivalente de la declaración de Python `o1 |= o2`.

*PyObject** **PyNumber_Long** (*PyObject* **o*)

Return value: *New reference.* Retorna el *o* convertido a un objeto entero en caso de éxito, o NULL en caso de error. Este es el equivalente de la expresión de Python `int(o)`.

*PyObject** **PyNumber_Float** (*PyObject* **o*)

Return value: *New reference.* Retorna el *o* convertido a un objeto flotante en caso de éxito o NULL en caso de error.

Este es el equivalente de la expresión de Python `float(o)`.

*PyObject** **PyNumber_Index** (*PyObject* *o)

Return value: *New reference.* Retorna el *o* convertido a un entero de Python (*int*) en caso de éxito o NULL con una excepción `TypeError` generada en caso de error.

*PyObject** **PyNumber_ToBase** (*PyObject* *n, int base)

Return value: *New reference.* Retorna el entero *n* convertido a base *base* como una cadena de caracteres. El argumento *base* debe ser uno de 2, 8, 10 o 16. Para la base 2, 8 o 16, la cadena retornada está prefijada con un marcador base de `'0b'`, `'0o'` o `'0x'`, respectivamente. Si *n* no es un entero (*int*) Python, primero se convierte con `PyNumber_Index()`.

Py_ssize_t **PyNumber_AsSsize_t** (*PyObject* *o, *PyObject* *exc)

Returns *o* converted to a *Py_ssize_t* value if *o* can be interpreted as an integer. If the call fails, an exception is raised and `-1` is returned.

If *o* can be converted to a Python `int` but the attempt to convert to a *Py_ssize_t* value would raise an `OverflowError`, then the *exc* argument is the type of exception that will be raised (usually `IndexError` or `OverflowError`). If *exc* is NULL, then the exception is cleared and the value is clipped to `PY_SSIZE_T_MIN` for a negative integer or `PY_SSIZE_T_MAX` for a positive integer.

int **PyIndex_Check** (*PyObject* *o)

Returns 1 if *o* is an index integer (has the `nb_index` slot of the `tp_as_number` structure filled in), and 0 otherwise. This function always succeeds.

7.4 Protocolo de secuencia

int **PySequence_Check** (*PyObject* *o)

Return 1 if the object provides the sequence protocol, and 0 otherwise. Note that it returns 1 for Python classes with a `__getitem__()` method, unless they are `dict` subclasses, since in general it is impossible to determine what type of keys the class supports. This function always succeeds.

Py_ssize_t **PySequence_Size** (*PyObject* *o)

Py_ssize_t **PySequence_Length** (*PyObject* *o)

Retorna el número de objetos en secuencia *o* en caso de éxito y `-1` en caso de error. Esto es equivalente a la expresión de Python `len(o)`.

*PyObject** **PySequence_Concat** (*PyObject* *o1, *PyObject* *o2)

Return value: *New reference.* Retorna la concatenación de *o1* y *o2* en caso de éxito, y NULL en caso de error. Este es el equivalente de la expresión de Python `o1+o2`.

*PyObject** **PySequence_Repeat** (*PyObject* *o, *Py_ssize_t* count)

Return value: *New reference.* Retorna el resultado de repetir el objeto de secuencia *o* *count* veces, o NULL en caso de falla. Este es el equivalente de la expresión de Python `o*count`.

*PyObject** **PySequence_InPlaceConcat** (*PyObject* *o1, *PyObject* *o2)

Return value: *New reference.* Retorna la concatenación de *o1* y *o2* en caso de éxito, y NULL en caso de error. La operación se realiza en su lugar *in-place* cuando *o1* lo admite. Este es el equivalente de la expresión de Python `o1+=o2`.

*PyObject** **PySequence_InPlaceRepeat** (*PyObject* *o, *Py_ssize_t* count)

Return value: *New reference.* Retorna el resultado de repetir el objeto de secuencia *o* *count* veces, o NULL en caso de falla. La operación se realiza en su lugar (*in-place*) cuando *o* lo admite. Este es el equivalente de la expresión de Python `o*=count`.

*PyObject** **PySequence_GetItem** (*PyObject* *o, *Py_ssize_t* i)

Return value: *New reference.* Retorna el elemento *i*-ésimo de *o* o NULL en caso de error. Este es el equivalente de la expresión de Python `o[i]`.

*PyObject** **PySequence_GetSlice** (*PyObject* **o*, *Py_ssize_t* *i1*, *Py_ssize_t* *i2*)

Return value: *New reference.* Retorna la rebanada del objeto secuencia *o* entre *i1* y *i2*, o NULL en caso de error. Este es el equivalente de la expresión de Python `o[i1:i2]`.

int **PySequence_SetItem** (*PyObject* **o*, *Py_ssize_t* *i*, *PyObject* **v*)

Asigna el objeto *v* al elemento *i*-ésimo de *o*. Lanza una excepción y retorna -1 en caso de falla; retorna 0 en caso de éxito. Este es el equivalente de la declaración de Python `o[i]=v`. Esta función *no* roba una referencia a *v*.

If *v* is NULL, the element is deleted, but this feature is deprecated in favour of using *PySequence_DelItem()*.

int **PySequence_DelItem** (*PyObject* **o*, *Py_ssize_t* *i*)

Elimina el elemento *i*-ésimo del objeto *o*. Retorna -1 en caso de falla. Este es el equivalente de la declaración de Python `del o[i]`.

int **PySequence_SetSlice** (*PyObject* **o*, *Py_ssize_t* *i1*, *Py_ssize_t* *i2*, *PyObject* **v*)

Asigna el objeto secuencia *v* al segmento en el objeto secuencia *o* de *i1* a *i2*. Este es el equivalente de la declaración de Python `o[i1:i2]=v`.

int **PySequence_DelSlice** (*PyObject* **o*, *Py_ssize_t* *i1*, *Py_ssize_t* *i2*)

Elimina el segmento en el objeto secuencia *o* de *i1* a *i2*. Retorna -1 en caso de falla. Este es el equivalente de la declaración de Python `del o[i1:i2]`.

Py_ssize_t **PySequence_Count** (*PyObject* **o*, *PyObject* **value*)

Retorna el número de apariciones de *value* en *o*, es decir, retorna el número de claves para las que `o[clave]==value`. En caso de fallo, retorna -1. Esto es equivalente a la expresión de Python `o.count(value)`.

int **PySequence_Contains** (*PyObject* **o*, *PyObject* **value*)

Determine si *o* contiene *valor*. Si un elemento en *o* es igual a *value*, retorna 1; de lo contrario, retorna 0. En caso de error, retorna -1. Esto es equivalente a la expresión de Python `value in o`.

Py_ssize_t **PySequence_Index** (*PyObject* **o*, *PyObject* **value*)

Retorna el primer índice *i* para el que `o[i]==value`. En caso de error, retorna -1. Esto es equivalente a la expresión de Python `o.index(value)`.

*PyObject** **PySequence_List** (*PyObject* **o*)

Return value: *New reference.* Retorna un objeto lista con el mismo contenido que la secuencia o iterable *o*, o NULL en caso de error. La lista retornada está garantizada como nueva. Esto es equivalente a la expresión de Python `list(o)`.

*PyObject** **PySequence_Tuple** (*PyObject* **o*)

Return value: *New reference.* Retorna un objeto tupla con el mismo contenido que la secuencia o iterable *o*, o NULL en caso de error. Si *o* es una tupla, se retornará una nueva referencia; de lo contrario, se construirá una tupla con el contenido apropiado. Esto es equivalente a la expresión de Python `tupla(o)`.

*PyObject** **PySequence_Fast** (*PyObject* **o*, const char **m*)

Return value: *New reference.* Retorna la secuencia o iterable *o* como un objeto utilizable por la otra familia de funciones *PySequence_Fast**. Si el objeto no es una secuencia o no es iterable, lanza *TypeError* con *m* como texto del mensaje. Retorna NULL en caso de falla.

Las funciones *PySequence_Fast** se denominan así porque suponen que *o* es un *PyTupleObject* o un *PyListObject* y acceden a los campos de datos de *o* directamente.

Como detalle de implementación de CPython, si *o* ya es una secuencia o lista, se retornará.

Py_ssize_t **PySequence_Fast_GET_SIZE** (*PyObject* **o*)

Returns the length of *o*, assuming that *o* was returned by *PySequence_Fast()* and that *o* is not NULL. The size can also be retrieved by calling *PySequence_Size()* on *o*, but *PySequence_Fast_GET_SIZE()* is faster because it can assume *o* is a list or tuple.

*PyObject** **PySequence_Fast_GET_ITEM** (*PyObject* **o*, *Py_ssize_t* *i*)

Return value: Borrowed reference. Retorna el elemento *i*-ésimo de *o*, suponiendo que *o* haya sido retornado por *PySequence_Fast()*, *o* no es NULL y que *i* está dentro de los límites.

*PyObject*** **PySequence_Fast_ITEMS** (*PyObject* **o*)

Retorna el arreglo subyacente de punteros *PyObject*. Asume que *o* fue retornado por *PySequence_Fast()* y *o* no es NULL.

Tenga en cuenta que si una lista cambia de tamaño, la reasignación puede reubicar el arreglo de elementos. Por lo tanto, solo use el puntero de arreglo subyacente en contextos donde la secuencia no puede cambiar.

*PyObject** **PySequence_ITEM** (*PyObject* **o*, *Py_ssize_t* *i*)

Return value: New reference. Retorna el elemento *i*-ésimo de *o* o NULL en caso de error. Es la forma más rápida de *PySequence_GetItem()* pero sin verificar que *PySequence_Check()* en *o* es verdadero y sin ajuste para índices negativos.

7.5 Protocolo de mapeo

Consulte también *PyObject_GetItem()*, *PyObject_SetItem()* y *PyObject_DelItem()*.

int **PyMapping_Check** (*PyObject* **o*)

Return 1 if the object provides the mapping protocol or supports slicing, and 0 otherwise. Note that it returns 1 for Python classes with a `__getitem__()` method, since in general it is impossible to determine what type of keys the class supports. This function always succeeds.

Py_ssize_t **PyMapping_Size** (*PyObject* **o*)

Py_ssize_t **PyMapping_Length** (*PyObject* **o*)

Retorna el número de claves en el objeto *o* en caso de éxito, y -1 en caso de error. Esto es equivalente a la expresión de Python `len(o)`.

*PyObject** **PyMapping_GetItemString** (*PyObject* **o*, const char **key*)

Return value: New reference. Retorna el elemento de *o* correspondiente a la cadena de caracteres *key* o NULL en caso de error. Este es el equivalente de la expresión de Python `o[key]`. Ver también *PyObject_GetItem()*.

int **PyMapping_SetItemString** (*PyObject* **o*, const char **key*, *PyObject* **v*)

Asigna la cadena de caracteres *key* al valor *v* en el objeto *o*. Retorna -1 en caso de falla. Este es el equivalente de la declaración de Python `o[key] = v`. Ver también *PyObject_SetItem()*. Esta función *no* roba una referencia a *v*.

int **PyMapping_DelItem** (*PyObject* **o*, *PyObject* **key*)

Elimina la asignación para el objeto *key* del objeto *o*. Retorna -1 en caso de falla. Esto es equivalente a la declaración de Python `del o[key]`. Este es un alias de *PyObject_DelItem()*.

int **PyMapping_DelItemString** (*PyObject* **o*, const char **key*)

Elimina la asignación de la cadena de caracteres *key* del objeto *o*. Retorna -1 en caso de falla. Esto es equivalente a la declaración de Python `del o[key]`.

int **PyMapping_HasKey** (*PyObject* **o*, *PyObject* **key*)

Retorna 1 si el objeto de mapeo tiene la clave *key* y 0 de lo contrario. Esto es equivalente a la expresión de Python `key in o`. Esta función siempre finaliza con éxito.

Tenga en cuenta que las excepciones que se producen al llamar al método `__getitem__()` se suprimirán. Para obtener informes de errores, utilice *PyObject_GetItem()* en su lugar.

int **PyMapping_HasKeyString** (*PyObject* **o*, const char **key*)

Retorna 1 si el objeto de mapeo tiene la clave *key* y 0 de lo contrario. Esto es equivalente a la expresión de Python `key in o`. Esta función siempre finaliza con éxito.

Tenga en cuenta que las excepciones que se producen al llamar al método `__getitem__()` y al crear un objeto de cadena de caracteres temporal se suprimirán. Para obtener informes de errores, utilice `PyMapping_GetItemString()` en su lugar.

*PyObject** **PyMapping_Keys** (*PyObject* *o)

Return value: *New reference.* En caso de éxito, retorna una lista de las claves en el objeto *o*. En caso de fallo, retorna NULL.

Distinto en la versión 3.7: Anteriormente, la función retornaba una lista o una tupla.

*PyObject** **PyMapping_Values** (*PyObject* *o)

Return value: *New reference.* En caso de éxito, retorna una lista de los valores en el objeto *o*. En caso de fallo, retorna NULL.

Distinto en la versión 3.7: Anteriormente, la función retornaba una lista o una tupla.

*PyObject** **PyMapping_Items** (*PyObject* *o)

Return value: *New reference.* En caso de éxito, retorna una lista de los elementos en el objeto *o*, donde cada elemento es una tupla que contiene un par clave-valor (*key-value*). En caso de fallo, retorna NULL.

Distinto en la versión 3.7: Anteriormente, la función retornaba una lista o una tupla.

7.6 Protocolo iterador

Hay dos funciones específicas para trabajar con iteradores.

int **PyIter_Check** (*PyObject* *o)

Retorna verdadero si el objeto *o* admite el protocolo iterador. Esta función siempre finaliza con éxito.

*PyObject** **PyIter_Next** (*PyObject* *o)

Return value: *New reference.* Retorna el siguiente valor de la iteración *o*. El objeto debe ser un iterador (depende de quién llama comprobar esto). Si no quedan valores restantes, retorna NULL sin establecer ninguna excepción. Si se produce un error al recuperar el elemento, retorna NULL y pasa la excepción.

Para escribir un bucle que itera sobre un iterador, el código en C debería verse así:

```
PyObject *iterator = PyObject_GetIter(obj);
PyObject *item;

if (iterator == NULL) {
    /* propagate error */
}

while ((item = PyIter_Next(iterator))) {
    /* do something with item */
    ...
    /* release reference when done */
    Py_DECREF(item);
}

Py_DECREF(iterator);

if (PyErr_Occurred()) {
    /* propagate error */
}
else {
    /* continue doing useful work */
}
```


7.7 Protocolo Búfer

Ciertos objetos disponibles en Python ajustan el acceso a un arreglo de memoria subyacente o *buffer*. Dichos objetos incluyen el incorporado `bytes` y `bytearray`, y algunos tipos de extensión como `array.array`. Las bibliotecas de terceros pueden definir sus propios tipos para fines especiales, como el procesamiento de imágenes o el análisis numérico.

Si bien cada uno de estos tipos tiene su propia semántica, comparten la característica común de estar respaldados por un búfer de memoria posiblemente grande. Es deseable, en algunas situaciones, acceder a ese búfer directamente y sin copia intermedia.

Python proporciona una instalación de este tipo en el nivel C en la forma de *protocolo búfer*. Este protocolo tiene dos lados:

- en el lado del productor, un tipo puede exportar una «interfaz de búfer» que permite a los objetos de ese tipo exponer información sobre su búfer subyacente. Esta interfaz se describe en la sección *Estructuras de Objetos Búfer*;
- en el lado del consumidor, hay varios medios disponibles para obtener un puntero a los datos subyacentes sin procesar de un objeto (por ejemplo, un parámetro de método).

Los objetos simples como `bytes` y `bytearray` exponen su búfer subyacente en forma orientada a bytes. Otras formas son posibles; por ejemplo, los elementos expuestos por un `array.array` pueden ser valores de varios bytes.

Un consumidor de ejemplo de la interfaz del búfer es el método `write()` de objetos de archivo: cualquier objeto que pueda exportar una serie de bytes a través de la interfaz del búfer puede escribirse en un archivo. Mientras que `write()` solo necesita acceso de solo lectura a los contenidos internos del objeto que se le pasa, otros métodos como `readinto()` necesitan acceso de escritura a los contenidos de su argumento. La interfaz del búfer permite que los objetos permitan o rechacen selectivamente la exportación de búferes de lectura-escritura y solo lectura.

Hay dos formas para que un consumidor de la interfaz del búfer adquiera un búfer sobre un objeto de destino:

- llamar `PyObject_GetBuffer()` con los parámetros correctos;
- llamar `PyArg_ParseTuple()` (o uno de sus hermanos) con uno de los `y*`, `w*` o `s*` *códigos de formato*.

En ambos casos, se debe llamar a `PyBuffer_Release()` cuando ya no se necesita el búfer. De lo contrario, podrían surgir varios problemas, como pérdidas de recursos.

7.7.1 Estructura de búfer

Las estructuras de búfer (o simplemente «búferes») son útiles como una forma de exponer los datos binarios de otro objeto al programador de Python. También se pueden usar como un mecanismo de corte de copia cero. Usando su capacidad para hacer referencia a un bloque de memoria, es posible exponer cualquier información al programador Python con bastante facilidad. La memoria podría ser una matriz grande y constante en una extensión C, podría ser un bloque de memoria sin procesar para su manipulación antes de pasar a una biblioteca del sistema operativo, o podría usarse para pasar datos estructurados en su formato nativo en memoria.

Contrariamente a la mayoría de los tipos de datos expuestos por el intérprete de Python, los búferes no son punteros `PyObject` sino estructuras C simples. Esto les permite ser creados y copiados de manera muy simple. Cuando se necesita un contenedor genérico alrededor de un búfer, un objeto *memoryview* puede ser creado.

Para obtener instrucciones breves sobre cómo escribir un objeto de exportación, consulte *Estructuras de objetos búfer*. Para obtener un búfer, consulte `PyObject_GetBuffer()`.

Py_buffer

void ***buf**

Un puntero al inicio de la estructura lógica descrita por los campos del búfer. Puede ser cualquier ubicación

dentro del bloque de memoria física subyacente del exportador. Por ejemplo, con negativo *strides* el valor puede apuntar al final del bloque de memoria.

Para arreglos *contiguous*, el valor apunta al comienzo del bloque de memoria.

void *obj

Una nueva referencia al objeto exportador. La referencia es propiedad del consumidor y automáticamente disminuye y se establece en NULL por *PyBuffer_Release()*. El campo es el equivalente del valor de retorno de cualquier función estándar de C-API.

Como un caso especial, para los búferes *temporary* que están envueltos por *PyMemoryView_FromBuffer()* o *PyBuffer_FillInfo()* este campo es NULL. En general, los objetos de exportación NO DEBEN usar este esquema.

Py_ssize_t len

`product(shape) * itemize`. Para arreglos contiguos, esta es la longitud del bloque de memoria subyacente. Para arreglos no contiguos, es la longitud que tendría la estructura lógica si se copiara en una representación contigua.

Accede a `((char *)buf)[0]` hasta `((char *)buf)[len-1]` solo es válido si el búfer se ha obtenido mediante una solicitud que garantiza la contigüidad. En la mayoría de los casos, dicha solicitud será *PyBUF_SIMPLE* o *PyBUF_WRITABLE*.

int readonly

Un indicador de si el búfer es de solo lectura. Este campo está controlado por el indicador *PyBUF_WRITABLE*.

Py_ssize_t itemsize

Tamaño del elemento en bytes de un solo elemento. Igual que el valor de `struct.calcsize()` invocado en valores no NULL *format*.

Excepción importante: si un consumidor solicita un búfer sin el indicador *PyBUF_FORMAT*, *format* se establecerá en NULL, pero *itemsize* todavía tiene el valor para el formato original.

Si *shape* está presente, la igualdad `product(shape) * itemsize == len` aún se mantiene y el consumidor puede usar *itemsize* para navegar el búfer.

Si *shape* es NULL como resultado de un *PyBUF_SIMPLE* o un *PyBUF_WRITABLE*, el consumidor debe ignorar *itemsize* y asumir `itemsize == 1`.

const char *format

Una cadena de caracteres terminada en NUL en sintaxis de estilo del modulo `struct` que describe el contenido de un solo elemento. Si esto es NULL, se supone "B" (bytes sin signo).

Este campo está controlado por el indicador *PyBUF_FORMAT*.

int ndim

El número de dimensiones que representa la memoria como un arreglo n-dimensional. Si es "0", *buf* apunta a un solo elemento que representa un escalar. En este caso, *shape*, *strides* y *suboffsets* DEBE ser NULL.

La macro `PyBUF_MAX_NDIM` limita el número máximo de dimensiones a 64. Los exportadores DEBEN respetar este límite, los consumidores de búfer multidimensionales DEBEN poder manejar hasta dimensiones `PyBUF_MAX_NDIM`.

Py_ssize_t *shape

Un arreglo de *Py_ssize_t* de longitud *ndim* que indica la forma de la memoria como un arreglo n-dimensional. Tenga en cuenta que `shape[0] * ... * shape[ndim-1] * itemsize` DEBE ser igual a *len*.

Los valores de forma están restringidos a `shape[n] >= 0`. El caso `shape[n] == 0` requiere atención especial. Vea arreglos complejos (*complex arrays*) para más información.

El arreglo de formas es de sólo lectura para el consumidor.

`Py_ssize_t *strides`

Un arreglo de `Py_ssize_t` de longitud `ndim` que proporciona el número de bytes que se omiten para llegar a un nuevo elemento en cada dimensión.

Los valores de `stride` pueden ser cualquier número entero. Para los arreglos regulares, los pasos son generalmente positivos, pero un consumidor DEBE ser capaz de manejar el caso `strides[n] <= 0`. Ver [complex arrays](#) para más información.

El arreglo `strides` es de sólo lectura para el consumidor.

`Py_ssize_t *suboffsets`

Un arreglo de `Py_ssize_t` de longitud `ndim`. Si `suboffsets[n] >= 0`, los valores almacenados a lo largo de la `n`-ésima dimensión son punteros y el valor del `suboffsets` dicta cuántos bytes agregar a cada puntero después de desreferenciarlos. Un valor de `suboffsets` negativo indica que no debe producirse una desreferenciación (*striding* en un bloque de memoria contiguo).

Si todos los `suboffsets` son negativos (es decir, no se necesita desreferenciar), entonces este campo debe ser NULL (el valor predeterminado).

Python Imaging Library (PIL) utiliza este tipo de representación de arreglos. Consulte [complex arrays](#) para obtener más información sobre cómo acceder a los elementos de dicho arreglo.

El arreglo de `suboffsets` es de sólo lectura para el consumidor.

`void *internal`

Esto es para uso interno del objeto exportador. Por ejemplo, el exportador podría volver a emitirlo como un número entero y utilizarlo para almacenar indicadores sobre si las matrices de forma, `strides` y `suboffsets` deben liberarse cuando se libera el búfer. El consumidor NO DEBE alterar este valor.

7.7.2 Tipos de solicitud búfer

Los búferes obtienen generalmente enviando una solicitud de búfer a un objeto de exportación a través de `PyObject_GetBuffer()`. Dado que la complejidad de la estructura lógica de la memoria puede variar drásticamente, el consumidor usa el argumento `flags` para especificar el tipo de búfer exacto que puede manejar.

Todos los campos `Py_buffer` están definidos inequívocamente por el tipo de solicitud.

campos independientes de solicitud

Los siguientes campos no están influenciados por `flags` y siempre deben completarse con los valores correctos: `obj`, `buf`, `len`, `itemsize`, `ndim`.

formato de sólo lectura

`PyBUF_WRITABLE`

Controla el campo `readonly`. Si se establece, el exportador DEBE proporcionar un búfer de escritura o, de lo contrario, informar de un error. De lo contrario, el exportador PUEDE proporcionar un búfer de sólo lectura o de escritura, pero la elección DEBE ser coherente para todos los consumidores.

`PyBUF_FORMAT`

Controla el campo `format`. Si se establece, este campo DEBE completarse correctamente. De lo contrario, este campo DEBE ser NULL.

`PyBUF_WRITABLE` puede ser l'd a cualquiera de las banderas en la siguiente sección. Dado que `PyBUF_SIMPLE` se define como 0, `PyBUF_WRITABLE` puede usarse como un indicador independiente para solicitar un búfer de escritura simple.

`PyBUF_FORMAT` puede ser l'd para cualquiera de las banderas excepto `PyBUF_SIMPLE`. Este último ya implica el formato B (bytes sin signo).

formas, *strides*, *suboffsets*

Las banderas que controlan la estructura lógica de la memoria se enumeran en orden decreciente de complejidad. Tenga en cuenta que cada bandera contiene todos los bits de las banderas debajo de ella.

Solicitud	forma	<i>strides</i>	<i>suboffsets</i>
PyBUF_INDIRECT	sí	sí	si es necesario
PyBUF_STRIDES	sí	sí	NULL
PyBUF_ND	sí	NULL	NULL
PyBUF_SIMPLE	NULL	NULL	NULL

solicitudes de contigüidad

La *contigüidad* C o Fortran se puede solicitar explícitamente, con y sin información de paso. Sin información de paso, el búfer debe ser C-contiguo.

Solicitud	forma	<i>strides</i>	<i>suboffsets</i>	contig
PyBUF_C_CONTIGUOUS	sí	sí	NULL	C
PyBUF_F_CONTIGUOUS	sí	sí	NULL	F
PyBUF_ANY_CONTIGUOUS	sí	sí	NULL	C o F
<code>PyBUF_ND</code>	sí	NULL	NULL	C

solicitudes compuestas

Todas las solicitudes posibles están completamente definidas por alguna combinación de las banderas en la sección anterior. Por conveniencia, el protocolo de memoria intermedia proporciona combinaciones de uso frecuente como indicadores únicos.

En la siguiente tabla *U* significa contigüidad indefinida. El consumidor tendría que llamar a `PyBuffer_IsContiguous()` para determinar la contigüidad.

Solicitud	forma	<i>strides</i>	<i>suboffsets</i>	contig	sólo lectura	formato
<code>PyBUF_FULL</code>	sí	sí	si es necesario	U	0	sí
<code>PyBUF_FULL_RO</code>	sí	sí	si es necesario	U	1 o 0	sí
<code>PyBUF_RECORDS</code>	sí	sí	NULL	U	0	sí
<code>PyBUF_RECORDS_RO</code>	sí	sí	NULL	U	1 o 0	sí
<code>PyBUF_STRIDED</code>	sí	sí	NULL	U	0	NULL
<code>PyBUF_STRIDED_RO</code>	sí	sí	NULL	U	1 o 0	NULL
<code>PyBUF_CONTIG</code>	sí	NULL	NULL	C	0	NULL
<code>PyBUF_CONTIG_RO</code>	sí	NULL	NULL	C	1 o 0	NULL

7.7.3 Arreglos complejos

Estilo NumPy: forma y *strides*

La estructura lógica de las matrices de estilo NumPy está definida por *itemsize*, *ndim*, *shape* y *strides*.

Si *ndim* == 0, la ubicación de memoria señalada por *buf* se interpreta como un escalar de tamaño *itemsize*. En ese caso, tanto *shape* como *strides* son NULL.

Si *strides* es NULL, el arreglo se interpreta como un arreglo C n-dimensional estándar. De lo contrario, el consumidor debe acceder a un arreglo n-dimensional de la siguiente manera:

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1] * strides[n-1];
item = *((typeof(item) *)ptr);
```

Como se señaló anteriormente, *buf* puede apuntar a cualquier ubicación dentro del bloque de memoria real. Un exportador puede verificar la validez de un búfer con esta función:

```
def verify_structure(memlen, itemsize, ndim, shape, strides, offset):
    """Verify that the parameters represent a valid array within
    the bounds of the allocated memory:
    char *mem: start of the physical memory block
    memlen: length of the physical memory block
    offset: (char *)buf - mem
    """
    if offset % itemsize:
        return False
    if offset < 0 or offset+itemsize > memlen:
        return False
    if any(v % itemsize for v in strides):
        return False
```

(continué en la próxima página)

(proviene de la página anterior)

```

if ndim <= 0:
    return ndim == 0 and not shape and not strides
if 0 in shape:
    return True

imin = sum(strides[j]*(shape[j]-1) for j in range(ndim)
            if strides[j] <= 0)
imax = sum(strides[j]*(shape[j]-1) for j in range(ndim)
            if strides[j] > 0)

return 0 <= offset+imin and offset+imax+itemsiz <= memlen

```

Estilo PIL: forma, *strides* y *suboffsets*

Además de los elementos normales, los arreglos de estilo PIL pueden contener punteros que deben seguirse para llegar al siguiente elemento en una dimensión. Por ejemplo, el arreglo C tridimensional regular `char v[2][2][3]` también se puede ver como un arreglo de 2 punteros a 2 arreglos bidimensionales: `char (*v[2])[2][3]`. En la representación de *suboffsets*, esos dos punteros pueden incrustarse al comienzo de *buf*, apuntando a dos matrices `char x[2][3]` que pueden ubicarse en cualquier lugar de la memoria.

Aquí hay una función que retorna un puntero al elemento en un arreglo N-D a la que apunta un índice N-dimensional cuando hay *strides* y *suboffsets* no NULL:

```

void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides,
                      Py_ssize_t *suboffsets, Py_ssize_t *indices) {
    char *pointer = (char*)buf;
    int i;
    for (i = 0; i < ndim; i++) {
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >= 0) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}

```

7.7.4 Funciones relacionadas a búfer

int **PyObject_CheckBuffer** (PyObject *obj)

Retorna 1 si *obj* admite la interfaz de búfer; de lo contrario, 0 cuando se retorna 1, no garantiza que *PyObject_GetBuffer()* tenga éxito. Esta función siempre finaliza con éxito.

int **PyObject_GetBuffer** (PyObject *exporter, Py_buffer *view, int flags)

Envía una solicitud al *exporter* para completar la *view* según lo especificado por *flags*. Si el exportador no puede proporcionar un búfer del tipo exacto, DEBE lanzar `PyExc_BufferError`, establecer *view->obj* en NULL y retornar -1.

Si tiene éxito, completa *view*, establece *view->obj* en una nueva referencia a *exporter* y retorna 0. En el caso de proveedores de búfer encadenados que redirigen las solicitudes a un solo objeto, *view->obj* PUEDE referirse a este objeto en lugar de *exporter* (Ver *Estructuras de objetos de búfer*).

Las llamadas exitosas a *PyObject_GetBuffer()* deben combinarse con las llamadas a *PyBuffer_Release()*, similar a `malloc()` y `free()`. Por lo tanto, después de que el consumidor

haya terminado con el búfer, `PyBuffer_Release()` debe llamarse exactamente una vez.

void **PyBuffer_Release** (*Py_buffer* *view)

Libera el búfer *view* y disminuye el conteo de referencias para *view->obj*. Esta función DEBE llamarse cuando el búfer ya no se utiliza, de lo contrario, pueden producirse fugas de referencia.

Es un error llamar a esta función en un búfer que no se obtuvo a través de `PyObject_GetBuffer()`.

Py_ssize_t **PyBuffer_SizeFromFormat** (const char *format)

Retorna el *itemsize* implícito de *format*. En caso de error, lanza una excepción y retorna -1.

Nuevo en la versión 3.9.

int **PyBuffer_IsContiguous** (*Py_buffer* *view, char order)

Retorna 1 si la memoria definida por *view* es de estilo C (*order* es 'C') o de estilo Fortran (*order* es 'F') *contiguous* o uno cualquiera (*order* es 'A'). Retorna 0 de lo contrario. Esta función siempre finaliza con éxito.

void* **PyBuffer_GetPointer** (*Py_buffer* *view, *Py_ssize_t* *indices)

Obtiene el área de memoria señalada por los *indices* dentro del *view* dado. *indices* deben apuntar a un arreglo de índices *view->ndim*.

int **PyBuffer_FromContiguous** (*Py_buffer* *view, void *buf, *Py_ssize_t* len, char fort)

Copia *len* bytes contiguos de *buf* a *view*. *fort* puede ser 'C' o 'F' (para pedidos al estilo C o al estilo Fortran). 0 se retorna en caso de éxito, -1 en caso de error.

int **PyBuffer_ToContiguous** (void *buf, *Py_buffer* *src, *Py_ssize_t* len, char order)

Copia *len* bytes de *src* a su representación contigua en *buf*. *order* puede ser 'C' o 'F' o 'A' (para pedidos al estilo C o al estilo Fortran o cualquiera) 0 se retorna en caso de éxito, -1 en caso de error.

Esta función falla si *len* != *src->len*.

void **PyBuffer_FillContiguousStrides** (int ndims, *Py_ssize_t* *shape, *Py_ssize_t* *strides, int itemsize, char order)

Rellena el arreglo *strides* con bytes de paso de un *contiguous* (estilo C si *order* es 'C' o estilo Fortran si *order* es 'F') arreglo de la forma dada con el número dado de bytes por elemento.

int **PyBuffer_FillInfo** (*Py_buffer* *view, *PyObject* *exporter, void *buf, *Py_ssize_t* len, int readonly, int flags)

Maneje las solicitudes de búfer para un exportador que quiera exponer *buf* de tamaño *len* con capacidad de escritura establecida de acuerdo con *readonly*. *buf* se interpreta como una secuencia de bytes sin signo.

El argumento *flags* indica el tipo de solicitud. Esta función siempre llena *view* según lo especificado por *flags*, a menos que *buf* haya sido designado como solo lectura y `PyBUF_WRITABLE` esté configurado en *flags*.

Si tiene éxito, establece *view->obj* en una nueva referencia a *exporter* y retorna 0. De lo contrario, aumenta `PyExc_BufferError`, establece *view->obj* en NULL y retorna -1;

Si esta función se usa como parte de a *getbufferproc*, *exporter* DEBE establecerse en el objeto exportador y *flags* deben pasarse sin modificaciones. De lo contrario, *exporter* DEBE ser NULL.

7.8 Protocolo de búfer antiguo

Obsoleto desde la versión 3.0.

Estas funciones formaban parte de la API del «antiguo protocolo de búfer» en Python 2. En Python 3, este protocolo ya no existe, pero las funciones aún están expuestas para facilitar la transferencia del código 2.x. Actúan como una envoltura de compatibilidad alrededor del *nuevo protocolo de búfer*, pero no le dan control sobre la vida útil de los recursos adquiridos cuando se exporta un búfer.

Por lo tanto, se recomienda que llame `PyObject_GetBuffer()` (o `y*`o`w* format codes` con la familia de funciones `PyArg_ParseTuple()`) para obtener una vista de búfer sobre un objeto, y `PyBuffer_Release()` cuando se puede liberar la vista de búfer.

int PyObject_AsCharBuffer (*PyObject* *obj, const char **buffer, *Py_ssize_t* *buffer_len)

Retorna un puntero a una ubicación de memoria de solo lectura que se puede usar como entrada basada en caracteres. El argumento *obj* debe admitir la interfaz de búfer de caracteres de segmento único. En caso de éxito, retorna 0, establece *buffer* en la ubicación de memoria y *buffer_len* en la longitud del búfer. Retorna -1 y lanza `TypeError` en caso de error.

int PyObject_AsReadBuffer (*PyObject* *obj, const void **buffer, *Py_ssize_t* *buffer_len)

Retorna un puntero a una ubicación de memoria de solo lectura que contiene datos arbitrarios. El argumento *obj* debe admitir la interfaz de búfer legible de segmento único. En caso de éxito, retorna 0, establece *buffer* en la ubicación de memoria y *buffer_len* en la longitud del búfer. Retorna -1 y lanza un `TypeError` en caso de error.

int PyObject_CheckReadBuffer (*PyObject* *o)

Retorna 1 si *o* admite la interfaz de búfer legible de segmento único. De lo contrario, retorna 0. Esta función siempre finaliza con éxito.

Tenga en cuenta que esta función intenta obtener y liberar un búfer, y las excepciones que se producen al llamar a las funciones correspondientes se suprimirán. Para obtener informes de errores, utilice `PyObject_GetBuffer()` en su lugar.

int PyObject_AsWriteBuffer (*PyObject* *obj, void **buffer, *Py_ssize_t* *buffer_len)

Retorna un puntero a una ubicación de memoria de escritura. El argumento *obj* debe admitir la interfaz de búfer de caracteres de segmento único. En caso de éxito, retorna 0, establece *buffer* en la ubicación de memoria y *buffer_len* en la longitud del búfer. Retorna -1 y lanza un `TypeError` en caso de error.

Capa de objetos concretos

Las funciones de este capítulo son específicas de ciertos tipos de objetos de Python. Pasarles un objeto del tipo incorrecto no es una buena idea; si recibe un objeto de un programa Python y no está seguro de que tenga el tipo correcto, primero debe realizar una verificación de tipo; por ejemplo, para verificar que un objeto es un diccionario, utilice `PyDict_Check()`. El capítulo está estructurado como el «árbol genealógico» de los tipos de objetos Python.

Advertencia: Si bien las funciones descritas en este capítulo verifican cuidadosamente el tipo de objetos que se pasan, muchos de ellos no verifican si se pasa `NULL` en lugar de un objeto válido. Permitir que se pase `NULL` puede causar violaciones de acceso a la memoria y la terminación inmediata del intérprete.

8.1 Objetos fundamentales

Esta sección describe los objetos de tipo Python y el objeto singleton `None`.

8.1.1 Objetos Tipos

PyTypeObject

La estructura C de los objetos utilizados para describir los tipos incorporados.

PyTypeObject **PyType_Type**

Este es el objeto tipo para objetos tipo; es el mismo objeto que `type` en la capa Python.

int PyType_Check (*PyObject* **o*)

Retorna un valor distinto de cero si el objeto *o* es un objeto tipo, incluidas las instancias de tipos derivados del objeto de tipo estándar. Retorna 0 en todos los demás casos. Esta función siempre finaliza con éxito.

int PyType_CheckExact (*PyObject* **o*)

Retorna un valor distinto de cero si el objeto *o* es un objeto tipo, pero no un subtipo del objeto tipo estándar. Retorna 0 en todos los demás casos. Esta función siempre finaliza con éxito.

unsigned int **PyType_ClearCache** ()

Borra la caché de búsqueda interna. Retorna la etiqueta (*tag*) de la versión actual.

unsigned long **PyType_GetFlags** (*PyTypeObject** *type*)

Retorna el miembro *tp_flags* de *type*. Esta función está destinada principalmente para su uso con *Py_LIMITED_API*; se garantiza que los bits de bandera (*flag*) individuales serán estables en las versiones de Python, pero el acceso a *tp_flags* en sí mismo no forma parte de la API limitada.

Nuevo en la versión 3.2.

Distinto en la versión 3.4: El tipo de retorno es ahora `unsigned long` en vez de `long`.

void **PyType_Modified** (*PyTypeObject** *type*)

Invalida la memoria caché de búsqueda interna para el tipo y todos sus subtipos. Esta función debe llamarse después de cualquier modificación manual de los atributos o clases base del tipo.

int **PyType_HasFeature** (*PyTypeObject** *o*, int *feature*)

Retorna un valor distinto de cero si el tipo objeto *o* establece la característica *feature*. Las características de tipo se indican mediante flags de un solo bit.

int **PyType_IS_GC** (*PyTypeObject** *o*)

Retorna verdadero si el objeto tipo incluye soporte para el detector de ciclo; Esto prueba el indicador de tipo *Py_TPFLAGS_HAVE_GC*.

int **PyType_IsSubtype** (*PyTypeObject** *a*, *PyTypeObject** *b*)

Retorna verdadero si *a* es un subtipo de *b*.

Esta función solo busca subtipos reales, lo que significa que `__subclasscheck__()` no se llama en *b*. Llama *PyObject_IsSubclass()* para hacer el mismo chequeo que `issubclass()` haría.

*PyObject** **PyType_GenericAlloc** (*PyTypeObject** *type*, *Py_ssize_t* *nitems*)

Return value: *New reference*. Controlador genérico para la ranura *tp_alloc* de un objeto tipo. Usa el mecanismo de asignación de memoria predeterminado de Python para asignar una nueva instancia e inicializar todo su contenido a NULL.

*PyObject** **PyType_GenericNew** (*PyTypeObject** *type*, *PyObject** *args*, *PyObject** *kwds*)

Return value: *New reference*. Controlador genérico para la ranura *tp_new* de un objeto tipo. Crea una nueva instancia utilizando la ranura del tipo *tp_alloc*.

int **PyType_Ready** (*PyTypeObject** *type*)

Finalizar un objeto tipo. Se debe llamar a todos los objetos tipo para finalizar su inicialización. Esta función es responsable de agregar ranuras heredadas de la clase base de un tipo. Retorna 0 en caso de éxito o retorna -1 y establece una excepción en caso de error.

Nota: If some of the base classes implements the GC protocol and the provided type does not include the *Py_TPFLAGS_HAVE_GC* in its flags, then the GC protocol will be automatically implemented from its parents. On the contrary, if the type being created does include *Py_TPFLAGS_HAVE_GC* in its flags then it **must** implement the GC protocol itself by at least implementing the *tp_traverse* handle.

void* **PyType_GetSlot** (*PyTypeObject** *type*, int *slot*)

Retorna el puntero de función almacenado en la ranura dada. Si el resultado es NULL, esto indica que la ranura es NULL o que la función se llamó con parámetros no válidos. Las personas que llaman suelen convertir el puntero de resultado en el tipo de función apropiado.

Consulte `PyType_Slot.slot` para conocer los posibles valores del argumento *slot*.

Se lanza una excepción si *type* no es un tipo montículo (*heap*).

Nuevo en la versión 3.4.

*PyObject** **PyType_GetModule** (*PyTypeObject* *type)

Retorna el objeto módulo asociado con el tipo dado cuando se creó el tipo usando *PyType_FromModuleAndSpec()*.

Si no hay ningún módulo asociado con el tipo dado, establece *TypeError* y retorna *NULL*.

Esta función se suele utilizar para obtener el módulo en el que se define un método. Tenga en cuenta que en un método de este tipo, es posible que *PyType_GetModule(Py_TYPE(self))* no retorne el resultado deseado. *Py_TYPE(self)* puede ser una *subclass* de la clase deseada, y las subclasses no están necesariamente definidas en el mismo módulo que su superclase. Consulte *PyCMethod* para obtener la clase que define el método.

Nuevo en la versión 3.9.

*void** **PyType_GetModuleState** (*PyTypeObject* *type)

Retorna el estado del objeto de módulo asociado con el tipo dado. Este es un atajo para llamar *PyModule_GetState()* en el resultado de *PyType_GetModule()*.

Si no hay ningún módulo asociado con el tipo dado, establece *TypeError* y retorna *NULL*.

Si el tipo *type* tiene un módulo asociado pero su estado es *NULL*, retorna *NULL* sin establecer una excepción.

Nuevo en la versión 3.9.

Crear tipos asignados en montículo (*heap*)

Las siguientes funciones y estructuras se utilizan para crear *heap types*.

*PyObject** **PyType_FromModuleAndSpec** (*PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

Return value: *New reference.* Crea y retorna un objeto montículo (*heap*) a partir de *spec* (*Py_TPFLAGS_HEAPTYPE*).

Si *bases* es una tupla, el tipo montículo (*heap*) creado contiene todos los tipos contenidos en él como tipos básicos.

Si *bases* es *NULL*, en su lugar se usa la ranura *Py_tp_base*. Si eso también es *NULL*, en su lugar, se utiliza la ranura *Py_tp_base*. Si eso también es *NULL*, el nuevo tipo se deriva de *object*.

The *module* argument can be used to record the module in which the new class is defined. It must be a module object or *NULL*. If not *NULL*, the module is associated with the new type and can later be retrieved with *PyType_GetModule()*. The associated module is not inherited by subclasses; it must be specified for each class individually.

Esta función llama *PyType_Ready()* en el tipo nuevo.

Nuevo en la versión 3.9.

*PyObject** **PyType_FromSpecWithBases** (*PyType_Spec* *spec, *PyObject* *bases)

Return value: *New reference.* Equivalente a *PyType_FromModuleAndSpec(NULL, spec, bases)*.

Nuevo en la versión 3.3.

*PyObject** **PyType_FromSpec** (*PyType_Spec* *spec)

Return value: *New reference.* Equivalente a *PyType_FromSpecWithBases(spec, NULL)*.

PyType_Spec

Estructura que define el comportamiento de un tipo.

*const char** **PyType_Spec.name**

Nombre del tipo, utilizado para establecer *PyTypeObject.tp_name*.

int **PyType_Spec.basicsize**

int `PyType_Spec.itemsize`

Tamaño de la instancia en bytes, utilizado para establecer `PyTypeObject.tp_basicsize` y `PyTypeObject.tp_itemsize`.

int `PyType_Spec.flags`

Banderas (*flags*) del tipo, que se usan para establecer `PyTypeObject.tp_flags`.

Si el indicador `Py_TPFLAGS_HEAPTYPE` no está establecido, `PyType_FromSpecWithBases()` lo establece automáticamente.

`PyType_Slot *PyType_Spec.slots`

Arreglo de estructuras `PyType_Slot`. Terminado por el valor de ranura especial `{0, NULL}`.

`PyType_Slot`

Estructura que define la funcionalidad opcional de un tipo, que contiene una ranura ID y un puntero de valor.

int `PyType_Slot.slot`

Una ranura ID.

Las ranuras IDs se nombran como los nombres de campo de las estructuras `PyTypeObject`, `PyNumberMethods`, `PySequenceMethods`, `PyMappingMethods` y `PyAsyncMethods` con un prefijo `Py_` agregado. Por ejemplo, use:

- `Py_tp_dealloc` para establecer `PyTypeObject.tp_dealloc`
- `Py_nb_add` para establecer `PyNumberMethods.nb_add`
- `Py_sq_length` para establecer `PySequenceMethods.sq_length`

Los siguientes campos no se pueden configurar en absoluto usando `PyType_Spec` y `PyType_Slot`:

- `tp_dict`
- `tp_mro`
- `tp_cache`
- `tp_subclasses`
- `tp_weaklist`
- `tp_vectorcall`
- `tp_weaklistoffset` (vea `PyMemberDef`)
- `tp_dictoffset` (vea `PyMemberDef`)
- `tp_vectorcall_offset` (vea `PyMemberDef`)

Los siguientes campos no se pueden establecer usando `PyType_Spec` y `PyType_Slot` cuando se utiliza la API limitada:

- `bf_getbuffer`
- `bf_releasebuffer`

Estableciendo `Py_tp_bases` o `Py_tp_base` puede ser problemático en algunas plataformas. Para evitar problemas, use el argumento `bases` de `PyType_FromSpecWithBases()` en su lugar.

Distinto en la versión 3.9: Slots in `PyBufferProcs` may be set in the unlimited API.

void `*PyType_Slot.pfunc`

El valor deseado de la ranura. En la mayoría de los casos, este es un puntero a una función.

Puede no ser NULL.

8.1.2 El objeto None

Tenga en cuenta que *PyObject* para None no está expuesto directamente en la API de Python/C. Como None es un singleton, es suficiente probar la identidad del objeto (usando `==` en C). No existe la función `PyNone_Check()` por la misma razón.

*PyObject** **Py_None**

El objeto None de Python, denota falta de valor. Este objeto no tiene métodos. Debe tratarse como cualquier otro objeto con respecto a los recuentos de referencia.

Py_RETURN_NONE

Maneje adecuadamente el retorno *Py_None* desde una función en C (es decir, incremente el recuento de referencia de None y devuélvalo).

8.2 Objetos numéricos

8.2.1 Objetos Enteros

Todos los enteros se implementan como objetos enteros «largos» (*long*) de tamaño arbitrario.

En caso de error, la mayoría de las API *PyLong_As** retornan (tipo de retorno) `-1` que no se puede distinguir de un número. Use *PyErr_Occurred()* para desambiguar.

PyLongObject

Este subtipo de *PyObject* representa un objeto entero de Python.

PyObject **PyLong_Type**

Esta instancia de *PyObject* representa el tipo entero de Python. Este es el mismo objeto que `int` en la capa de Python.

`int` **PyLong_Check** (*PyObject* *p)

Retorna verdadero si su argumento es un *PyLongObject* o un subtipo de *PyLongObject*. Esta función siempre finaliza con éxito.

`int` **PyLong_CheckExact** (*PyObject* *p)

Retorna verdadero si su argumento es un *PyLongObject*, pero no un subtipo de *PyLongObject*. Esta función siempre finaliza con éxito.

*PyObject** **PyLong_FromLong** (long v)

Return value: New reference. Retorna un objeto *PyLongObject* nuevo desde v, o NULL en caso de error.

The current implementation keeps an array of integer objects for all integers between `-5` and `256`. When you create an `int` in that range you actually just get back a reference to the existing object.

*PyObject** **PyLong_FromUnsignedLong** (unsigned long v)

Return value: New reference. Retorna un objeto *PyLongObject* nuevo desde un C `unsigned long`, o NULL en caso de error.

*PyObject** **PyLong_FromSsize_t** (Py_ssize_t v)

Return value: New reference. Retorna un objeto *PyLongObject* nuevo desde un C `Py_ssize_t`, o NULL en caso de error.

*PyObject** **PyLong_FromSize_t** (size_t v)

Return value: New reference. Retorna un objeto *PyLongObject* nuevo desde un C `size_t`, o NULL en caso de error.

*PyObject** **PyLong_FromLongLong** (long long *v*)

Return value: *New reference.* Retorna un objeto *PyLongObject* nuevo desde un C long long, o NULL en caso de error.

*PyObject** **PyLong_FromUnsignedLongLong** (unsigned long long *v*)

Return value: *New reference.* Retorna un objeto *PyLongObject* nuevo desde un C unsigned long long, o NULL en caso de error.

*PyObject** **PyLong_FromDouble** (double *v*)

Return value: *New reference.* Retorna un nuevo objeto *PyLongObject* de la parte entera de *v*, o NULL en caso de error.

*PyObject** **PyLong_FromString** (const char **str*, char ***pend*, int *base*)

Return value: *New reference.* Retorna un nuevo *PyLongObject* basado en el valor de cadena de caracteres en *str*, que se interpreta de acuerdo con la raíz en *base*. Si *pend* no es NULL, **pend* apuntará al primer carácter en *str* que sigue a la representación del número. Si *base* es 0, *str* se interpreta utilizando la definición integers; en este caso, los ceros a la izquierda en un número decimal distinto de cero lanzan un *ValueError*. Si *base* no es 0, debe estar entre 2 y 36, inclusive. Se ignoran los espacios iniciales y los guiones bajos individuales después de un especificador base y entre dígitos. Si no hay dígitos, se lanzará *ValueError*.

*PyObject** **PyLong_FromUnicode** (*Py_UNICODE* **u*, *Py_ssize_t* *length*, int *base*)

Return value: *New reference.* Convierte una secuencia de dígitos Unicode en un valor entero de Python.

Deprecated since version 3.3, will be removed in version 3.10: Parte de la API de viejo estilo *Py_UNICODE*; por favor migrar para usar *PyLong_FromUnicodeObject* ().

*PyObject** **PyLong_FromUnicodeObject** (*PyObject* **u*, int *base*)

Return value: *New reference.* Convierte una secuencia de dígitos Unicode en la cadena de caracteres *u* en un valor entero de Python.

Nuevo en la versión 3.3.

*PyObject** **PyLong_FromVoidPtr** (void **p*)

Return value: *New reference.* Crea un entero de Python desde el puntero *p*. El valor del puntero se puede recuperar del valor resultante usando *PyLong_AsVoidPtr* ().

long **PyLong_AsLong** (*PyObject* **obj*)

Retorna una representación de C long de *obj*. Si *obj* no es una instancia de *PyLongObject*, primero llama a su método *__index__* () o *__int__* () (si está presente) para convertirlo en un *PyLongObject*.

Lanza *OverflowError* si el valor de *obj* está fuera de rango para un long.

Retorna -1 en caso de error. Use *PyErr_Occurred* () para desambiguar.

Distinto en la versión 3.8: Use *__index__* () si está disponible.

Obsoleto desde la versión 3.8: Usar *__int__* () está deprecado.

long **PyLong_AsLongAndOverflow** (*PyObject* **obj*, int **overflow*)

Retorna una representación de C long de *obj*. Si *obj* no es una instancia de *PyLongObject*, primero llama a su método *__index__* () o *__int__* () (si está presente) para convertirlo en un *PyLongObject*.

Si el valor de *obj* es mayor que *LONG_MAX* o menor que *LONG_MIN*, establece **overflow* en "1" o "-1", respectivamente, y retorna "-1"; de lo contrario, establece ***overflow* en 0. Si se produce alguna otra excepción, configura **overflow* en 0 y retorna -1 como de costumbre.

Retorna -1 en caso de error. Use *PyErr_Occurred* () para desambiguar.

Distinto en la versión 3.8: Use *__index__* () si está disponible.

Obsoleto desde la versión 3.8: Usar *__int__* () está deprecado.

`long long PyLong_AsLongLong (PyObject *obj)`

Retorna una representación de C `long long` de *obj*. Si *obj* no es una instancia de *PyLongObject*, primero llama a su método `__index__()` o `__int__()` (si está presente) para convertirlo en un *PyLongObject*.

Lanza `OverflowError` si el valor de *obj* está fuera de rango para un `long long`.

Retorna `-1` en caso de error. Use `PyErr_Occurred()` para desambiguar.

Distinto en la versión 3.8: Use `__index__()` si está disponible.

Obsoleto desde la versión 3.8: Usar `__int__()` está deprecado.

`long long PyLong_AsLongLongAndOverflow (PyObject *obj, int *overflow)`

Retorna una representación de C `long long` de *obj*. Si *obj* no es una instancia de *PyLongObject*, primero llama a su método `__index__()` o `__int__()` (si está presente) para convertirlo en un *PyLongObject*.

Si el valor de *obj* es mayor que `LLONG_MAX` o menor que `LLONG_MIN`, establece *overflow* en 1 o `-1`, respectivamente, y retorna `-1`; de lo contrario, establece *overflow* en 0. Si se produce alguna otra excepción, configura *overflow* en 0 y retorna `-1` como de costumbre.

Retorna `-1` en caso de error. Use `PyErr_Occurred()` para desambiguar.

Nuevo en la versión 3.2.

Distinto en la versión 3.8: Use `__index__()` si está disponible.

Obsoleto desde la versión 3.8: Usar `__int__()` está deprecado.

`Py_ssize_t PyLong_AsSsize_t (PyObject *pylong)`

Retorna una representación de C `Py_ssize_t` de *pylong*. *pylong* debe ser una instancia de *PyLongObject*.

Lanza `OverflowError` si el valor de *pylong* está fuera de rango para un `Py_ssize_t`.

Retorna `-1` en caso de error. Use `PyErr_Occurred()` para desambiguar.

`unsigned long PyLong_AsUnsignedLong (PyObject *pylong)`

Retorna una representación de C `unsigned long` de *pylong*. *pylong* debe ser una instancia de *PyLongObject*.

Lanza `OverflowError` si el valor de *pylong* está fuera de rango para un `unsigned long`.

Retorna `(unsigned long) -1` en caso de error. Use `PyErr_Occurred()` para desambiguar.

`size_t PyLong_AsSize_t (PyObject *pylong)`

Retorna una representación de C `size_t` de *pylong*. *pylong* debe ser una instancia de *PyLongObject*.

Lanza `OverflowError` si el valor de *pylong* está fuera de rango para un `size_t`.

Retorna `(size_t) -1` en caso de error. Use `PyErr_Occurred()` para desambiguar.

`unsigned long long PyLong_AsUnsignedLongLong (PyObject *pylong)`

Retorna una representación de C `unsigned long long` de *pylong*. *pylong* debe ser una instancia de *PyLongObject*.

Lanza `OverflowError` si el valor de *pylong* está fuera de rango para un `unsigned long long`.

Retorna `(unsigned long long) -1` en caso de error. Use `PyErr_Occurred()` para desambiguar.

Distinto en la versión 3.1: Ahora un *pylong* negativo lanza un `OverflowError`, no `TypeError`.

`unsigned long PyLong_AsUnsignedLongMask (PyObject *obj)`

Retorna una representación de C `unsigned long` de *obj*. Si *obj* no es una instancia de *PyLongObject*, primero llama a su método `__index__()` o `__int__()` (si está presente) para convertirlo en un *PyLongObject*.

Si el valor de *obj* está fuera del rango para `unsigned long`, retorna la reducción de ese valor módulo `ULONG_MAX + 1`.

Retorna (unsigned long) -1 en caso de error. Use `PyErr_Occurred()` para desambiguar.

Distinto en la versión 3.8: Use `__index__()` si está disponible.

Obsoleto desde la versión 3.8: Usar `__int__()` está deprecado.

unsigned long long **PyLong_AsUnsignedLongLongMask** (*PyObject* *obj)

Retorna una representación de C unsigned long long de *obj*. Si *obj* no es una instancia de `PyLongObject`, primero llama a su método `__index__()` o `__int__()` (si está presente) para convertirlo en un `PyLongObject`.

Si el valor de *obj* está fuera del rango para unsigned long long, retorna la reducción de ese valor módulo `ULLONG_MAX + 1`.

Retorna (unsigned long long) -1 por error. Use `PyErr_Occurred()` para desambiguar.

Distinto en la versión 3.8: Use `__index__()` si está disponible.

Obsoleto desde la versión 3.8: Usar `__int__()` está deprecado.

double **PyLong_AsDouble** (*PyObject* *pylong)

Retorna una representación de C double de *pylong*. *pylong* debe ser una instancia de `PyLongObject`.

Lanza `OverflowError` si el valor de *pylong* está fuera de rango para un double.

Retorna -1.0 en caso de error. Use `PyErr_Occurred()` para desambiguar.

void* **PyLong_AsVoidPtr** (*PyObject* *pylong)

Convierte un entero Python *pylong* en un puntero C void. Si *pylong* no se puede convertir, se generará un `OverflowError`. Esto solo se garantiza para producir un puntero utilizable void para valores creados con `PyLong_FromVoidPtr()`.

Retorna NULL en caso de error. Use `PyErr_Occurred()` para desambiguar.

8.2.2 Objetos Booleanos

Los booleanos en Python se implementan como una subclase de enteros. Solo hay dos booleanos `Py_False` y `Py_True`. Como tal, las funciones normales de creación y eliminación no se aplican a los booleanos. Sin embargo, los siguientes macros están disponibles.

int **PyBool_Check** (*PyObject* *o)

Retorna verdadero si *o* es de tipo `PyBool_Type`. Esta función siempre finaliza con éxito.

*PyObject** **Py_False**

El objeto `False` de Python. Este objeto no tiene métodos. Debe tratarse como cualquier otro objeto con respecto a los recuentos de referencia.

*PyObject** **Py_True**

El objeto `True` de Python. Este objeto no tiene métodos. Debe tratarse como cualquier otro objeto con respecto a los recuentos de referencia.

Py_RETURN_FALSE

Retorna `Py_False` de una función, incrementando adecuadamente su recuento de referencia.

Py_RETURN_TRUE

Retorna `Py_True` desde una función, incrementando adecuadamente su recuento de referencia.

*PyObject** **PyBool_FromLong** (long v)

Return value: New reference. Retorna una nueva referencia a `Py_True` o `Py_False` dependiendo del valor de verdad de *v*.

8.2.3 Objetos de punto flotante

PyFloatObject

Este subtipo de *PyObject* representa un objeto de punto flotante de Python.

PyObject **PyFloat_Type**

Esta instancia de *PyTypeObject* representa el tipo de punto flotante de Python. Este es el mismo objeto que `float` en la capa de Python.

int **PyFloat_Check** (*PyObject* *p)

Retorna verdadero si su argumento es un *PyFloatObject* o un subtipo de *PyFloatObject*. Esta función siempre finaliza con éxito.

int **PyFloat_CheckExact** (*PyObject* *p)

Retorna verdadero si su argumento es un *PyFloatObject*, pero no un subtipo de *PyFloatObject*. Esta función siempre finaliza con éxito.

*PyObject** **PyFloat_FromString** (*PyObject* *str)

Return value: New reference. Crea un objeto *PyFloatObject* en función del valor de cadena de caracteres en *str* o NULL en caso de error.

*PyObject** **PyFloat_FromDouble** (double v)

Return value: New reference. Crea un objeto *PyFloatObject* a partir de *v*, o NULL en caso de error.

double **PyFloat_AsDouble** (*PyObject* *pyfloat)

Retorna una representación C `double` de los contenidos de *pyfloat*. Si *pyfloat* no es un objeto de punto flotante de Python pero tiene un método `__float__()`, primero se llamará a este método para convertir *pyfloat* en un flotante. Si `__float__()` no está definido, entonces recurre a `__index__()`. Este método retorna `-1.0` en caso de falla, por lo que se debe llamar a *PyErr_Occurred()* para verificar si hay errores.

Distinto en la versión 3.8: Utilice `__index__()` si está disponible.

double **PyFloat_AS_DOUBLE** (*PyObject* *pyfloat)

Retorna una representación C `double` de los contenidos de *pyfloat*, pero sin verificación de errores.

*PyObject** **PyFloat_GetInfo** (void)

Return value: New reference. Retorna una instancia de *structseq* que contiene información sobre la precisión, los valores mínimos y máximos de un flotante. Es una envoltura delgada alrededor del archivo de encabezado `float.h`.

double **PyFloat_GetMax** ()

Retorna el máximo flotante finito representable `DBL_MAX` como C `double`.

double **PyFloat_GetMin** ()

Retorna el flotante positivo normalizado mínimo `DBL_MIN` como C `double`.

8.2.4 Objetos de números complejos

Los objetos de números complejos de Python se implementan como dos tipos distintos cuando se ven desde la API de C: uno es el objeto de Python expuesto a los programas de Python, y el otro es una estructura en C que representa el valor de número complejo real. La API proporciona funciones para trabajar con ambos.

Números complejos como estructuras C

Tenga en cuenta que las funciones que aceptan estas estructuras como parámetros y las retornan como resultados lo hacen *por valor* en lugar de desreferenciarlas a través de punteros. Esto es consistente en toda la API.

Py_complex

La estructura C que corresponde a la porción de valor de un objeto de número complejo de Python. La mayoría de las funciones para tratar con objetos de números complejos utilizan estructuras de este tipo como valores de entrada o salida, según corresponda. Se define como:

```
typedef struct {  
    double real;  
    double imag;  
} Py_complex;
```

Py_complex **_Py_c_sum** (*Py_complex left*, *Py_complex right*)

Retorna la suma de dos números complejos, utilizando la representación C *Py_complex*.

Py_complex **_Py_c_diff** (*Py_complex left*, *Py_complex right*)

Retorna la diferencia entre dos números complejos, usando la representación C *Py_complex*.

Py_complex **_Py_c_neg** (*Py_complex num*)

Return the negation of the complex number *num*, using the C *Py_complex* representation.

Py_complex **_Py_c_prod** (*Py_complex left*, *Py_complex right*)

Retorna el producto de dos números complejos, usando la representación C *Py_complex*.

Py_complex **_Py_c_quot** (*Py_complex dividend*, *Py_complex divisor*)

Retorna el cociente de dos números complejos, utilizando la representación C *Py_complex*.

Si *divisor* es nulo, este método retorna cero y establece `errno` en EDOM.

Py_complex **_Py_c_pow** (*Py_complex num*, *Py_complex exp*)

Retorna la exponenciación de *num* por *exp*, utilizando la representación C *Py_complex*.

Si *num* es nulo y *exp* no es un número real positivo, este método retorna cero y establece `errno` a EDOM.

Números complejos como objetos de Python

PyComplexObject

Este subtipo de *PyObject* representa un objeto de número complejo de Python.

PyTypeObject **PyComplex_Type**

Esta instancia de *PyTypeObject* representa el tipo de número complejo de Python. Es el mismo objeto que `complex` en la capa de Python.

int **PyComplex_Check** (*PyObject *p*)

Retorna verdadero si su argumento es un *PyComplexObject* o un subtipo de *PyComplexObject*. Esta función siempre finaliza con éxito.

int **PyComplex_CheckExact** (*PyObject *p*)

Retorna verdadero si su argumento es un *PyComplexObject*, pero no un subtipo de *PyComplexObject*. Esta función siempre finaliza con éxito.

*PyObject** **PyComplex_FromCComplex** (*Py_complex v*)

Return value: New reference. Crea un nuevo objeto de número complejo de Python a partir de un valor C *Py_complex*.

*PyObject** **PyComplex_FromDoubles** (double *real*, double *imag*)

Return value: New reference. Retorna un nuevo objeto *PyComplexObject* de *real* e *imag*.

double **PyComplex_RealAsDouble** (*PyObject* **op*)

Retorna la parte real de *op* como double en C.

double **PyComplex_ImagAsDouble** (*PyObject* **op*)

Retorna la parte imaginaria de *op* como un double de C.

Py_complex **PyComplex_AsCComplex** (*PyObject* **op*)

Retorna el valor *Py_complex* del número complejo *op*.

Si *op* no es un objeto de número complejo de Python pero tiene un método `__complex__()`, primero se llamará a este método para convertir *op* en un objeto de número complejo de Python. Si `__complex__()` no está definido, vuelve a `__float__()`. Si `__float__()` no está definido, entonces recurre a `__index__()`. En caso de falla, este método retorna `-1.0` como un valor real.

Distinto en la versión 3.8: Use `__index__()` si está disponible.

8.3 Objetos de secuencia

Las operaciones genéricas en los objetos de secuencia se discutieron en el capítulo anterior; Esta sección trata sobre los tipos específicos de objetos de secuencia que son intrínsecos al lenguaje Python.

8.3.1 Objetos Bytes

These functions raise `TypeError` when expecting a bytes parameter and called with a non-bytes parameter.

PyBytesObject

Este subtipo de *PyObject* representa un objeto bytes de Python.

PyTypeObject **PyBytes_Type**

Esta instancia de *PyTypeObject* representa el tipo bytes de Python; es el mismo objeto que `bytes` en la capa de Python.

int **PyBytes_Check** (*PyObject* **o*)

Retorna verdadero si el objeto *o* es un objeto bytes o una instancia de un subtipo del tipo bytes. Esta función siempre finaliza con éxito.

int **PyBytes_CheckExact** (*PyObject* **o*)

Retorna verdadero si el objeto *o* es un objeto bytes, pero no una instancia de un subtipo del tipo bytes. Esta función siempre finaliza con éxito.

*PyObject** **PyBytes_FromString** (const char **v*)

Return value: New reference. Retorna un nuevo objeto bytes con una copia de la cadena de caracteres *v* como valor en caso de éxito y NULL en caso de error. El parámetro *v* no debe ser NULL; no se comprobará.

*PyObject** **PyBytes_FromStringAndSize** (const char **v*, *Py_ssize_t* *len*)

Return value: New reference. Retorna un nuevo objeto bytes con una copia de la cadena de caracteres *v* como valor y longitud *len* en caso de éxito y NULL en caso de error. Si *v* es NULL, el contenido del objeto bytes no se inicializa.

*PyObject** **PyBytes_FromFormat** (const char **format*, ...)

Return value: New reference. Toma una cadena de caracteres *format* del estilo `C printf()` y un número variable de argumentos, calcula el tamaño del objeto bytes Python resultante y retorna un objeto bytes con los valores formateados. Los argumentos variables deben ser tipos C y deben corresponder exactamente a los caracteres de formato en la cadena de caracteres *format*. Se permiten los siguientes caracteres de formato:

Caracteres de formato	Tipo	Comentario
<code>%%</code>	<i>n/a</i>	El carácter literal <code>%</code> .
<code>%c</code>	<code>int</code>	Un solo byte, representado como un C int.
<code>%d</code>	<code>int</code>	Equivalente a <code>printf("%d").</code> ¹
<code>%u</code>	<code>unsigned int</code>	Equivalente a <code>printf("%u").</code> ¹
<code>%ld</code>	<code>long</code>	Equivalente a <code>printf("%ld").</code> ¹
<code>%lu</code>	<code>unsigned long</code>	Equivalente a <code>printf("%lu").</code> ¹
<code>%zd</code>	<code>Py_ssize_t</code>	Equivalente a <code>printf("%zd").</code> ¹
<code>%zu</code>	<code>size_t</code>	Equivalente a <code>printf("%zu").</code> ¹
<code>%i</code>	<code>int</code>	Equivalente a <code>printf("%i").</code> ¹
<code>%x</code>	<code>int</code>	Equivalente a <code>printf("%x").</code> ¹
<code>%s</code>	<code>const char*</code>	Un arreglo de caracteres C terminados en nulo.
<code>%p</code>	<code>const void*</code>	La representación hexadecimal de un puntero en C. Principalmente equivalente a <code>printf("%p")</code> excepto que se garantiza que comience con el literal <code>0x</code> , independientemente de lo que produzca el <code>printf</code> de la plataforma.

Un carácter de formato no reconocido hace que todo el resto de la cadena de caracteres de formato se copie como está en el objeto de resultado y se descartan los argumentos adicionales.

*PyObject** **PyBytes_FromFormatV** (`const char *format`, *va_list* *vargs*)

Return value: *New reference.* Idéntica a `PyBytes_FromFormat()` excepto que toma exactamente dos argumentos.

*PyObject** **PyBytes_FromObject** (*PyObject* **o*)

Return value: *New reference.* Retorna la representación en bytes del objeto *o* que implementa el protocolo de búfer.

Py_ssize_t **PyBytes_Size** (*PyObject* **o*)

Retorna la longitud de los bytes en el objeto bytes *o*.

Py_ssize_t **PyBytes_GET_SIZE** (*PyObject* **o*)

Forma macro de `PyBytes_Size()` pero sin verificación de errores.

`char*` **PyBytes_AsString** (*PyObject* **o*)

Retorna un puntero al contenido de *o*. El puntero se refiere al búfer interno de *o*, que consiste en bytes `len(o) + 1`. El último byte en el búfer siempre es nulo, independientemente de si hay otros bytes nulos. Los datos no deben modificarse de ninguna manera, a menos que el objeto se haya creado usando `PyBytes_FromStringAndSize(NULL, size)`. No debe ser desasignado. Si *o* no es un objeto de bytes en absoluto, `PyBytes_AsString()` retorna `NULL` y lanza un `TypeError`.

`char*` **PyBytes_AS_STRING** (*PyObject* **string*)

Forma macro de `PyBytes_AsString()` pero sin verificación de errores.

`int` **PyBytes_AsStringAndSize** (*PyObject* **obj*, `char **buffer`, *Py_ssize_t* **length*)

Retorna los contenidos terminados en nulo del objeto *obj* a través de las variables de salida *buffer* y *length*.

Si *length* es `NULL`, el objeto bytes no puede contener bytes nulos incrustados; si lo hace, la función retorna `-1` y se genera un `ValueError`.

El búfer se refiere a un búfer interno de *obj*, que incluye un byte nulo adicional al final (sin contar en *length*). Los datos no deben modificarse de ninguna manera, a menos que el objeto se haya creado usando `PyBytes_FromStringAndSize(NULL, size)`. No debe ser desasignado. Si *obj* no es un objeto bytes en absoluto, `PyBytes_AsStringAndSize()` retorna `-1` y lanza `TypeError`.

Distinto en la versión 3.5: Anteriormente, `TypeError` se lanzaba cuando se encontraban bytes nulos incrustados en el objeto bytes.

¹ Para especificadores de enteros (*d*, *u*, *ld*, *lu*, *zd*, *zu*, *i*, *x*): el indicador de conversión *0* tiene efecto incluso cuando se proporciona una precisión.

void **PyBytes_Concat** (*PyObject* **bytes, *PyObject* *newpart)

Crea un nuevo objeto de bytes en *bytes que contiene el contenido de newpart agregado a bytes; la persona que llama poseerá la nueva referencia. La referencia al valor anterior de bytes será robada. Si no se puede crear el nuevo objeto, la referencia anterior a bytes se seguirá descartando y el valor de *bytes se establecerá en NULL; Se establecerá la excepción apropiada.

void **PyBytes_ConcatAndDel** (*PyObject* **bytes, *PyObject* *newpart)

Crea un nuevo objeto de bytes en *bytes que contenga el contenido de newpart agregado a bytes. Esta versión disminuye el recuento de referencias de newpart.

int **_PyBytes_Resize** (*PyObject* **bytes, *Py_ssize_t* newsize)

Una forma de cambiar el tamaño de un objeto bytes aunque sea «inmutable». Solo use esto para construir un nuevo objeto bytes; no use esto si los bytes ya pueden ser conocidos en otras partes del código. Es un error llamar a esta función si el recuento en el objeto bytes de entrada no es uno. Pasa la dirección de un objeto de bytes existente como un lvalue (puede escribirse en él) y el nuevo tamaño deseado. En caso de éxito, *bytes retiene el objeto de bytes redimensionados y se retorna 0; la dirección en *bytes puede diferir de su valor de entrada. Si la reasignación falla, el objeto de bytes original en *bytes se desasigna, *bytes se establece en NULL, MemoryError se establece y se retorna -1.

8.3.2 Objetos de arreglos de bytes (bytearrays)

PyByteArrayObject

Este subtipo de *PyObject* representa un objeto arreglo de bytes de Python.

PyTypeObject **PyByteArray_Type**

Esta instancia de *PyTypeObject* representa el tipo arreglo de bytes de Python; es el mismo objeto que bytearray en la capa de Python.

Macros de verificación de tipos

int **PyByteArray_Check** (*PyObject* *o)

Retorna verdadero si el objeto o es un objeto de arreglo de bytes o una instancia de un subtipo del tipo arreglo de bytes. Esta función siempre finaliza con éxito.

int **PyByteArray_CheckExact** (*PyObject* *o)

Retorna verdadero si el objeto o es un objeto de arreglo de bytes, pero no una instancia de un subtipo del tipo arreglo de bytes. Esta función siempre finaliza con éxito.

Funciones API directas

*PyObject** **PyByteArray_FromObject** (*PyObject* *o)

Return value: New reference. Retorna un nuevo objeto de arreglo de bytes de cualquier objeto, o, que implementa el buffer protocol.

*PyObject** **PyByteArray_FromStringAndSize** (const char *string, *Py_ssize_t* len)

Return value: New reference. Crea un nuevo objeto de arreglo de bytes a partir de string y su longitud, len. En caso de fallo, se retorna NULL.

*PyObject** **PyByteArray_Concat** (*PyObject* *a, *PyObject* *b)

Return value: New reference. Une los arreglos de bytes (bytearrays) a y b y retorna un nuevo arreglo de bytes (bytearray) con el resultado.

Py_ssize_t **PyByteArray_Size** (*PyObject* *bytearray)

Retorna el tamaño de bytearray después de buscar un puntero NULL.

`char* PyByteArray_AsString (PyObject *bytearray)`

Retorna el contenido de *bytearray* como un arreglo de caracteres después de verificar un puntero NULL. La arreglo retornado siempre tiene un byte nulo adicional agregado.

`int PyByteArray_Resize (PyObject *bytearray, Py_ssize_t len)`

Cambia el tamaño del búfer interno de *bytearray* a *len*.

Macros

Estos macros intercambian seguridad por velocidad y no comprueban punteros.

`char* PyByteArray_AS_STRING (PyObject *bytearray)`

Versión macro de *PyByteArray_AsString()*.

`Py_ssize_t PyByteArray_GET_SIZE (PyObject *bytearray)`

Versión macro de *PyByteArray_Size()*.

8.3.3 Objetos y códecs Unicode

Objetos Unicode

Desde la implementación del **PEP 393** en Python 3.3, los objetos Unicode utilizan internamente una variedad de representaciones, para permitir el manejo del rango completo de caracteres Unicode mientras se mantiene la eficiencia de memoria. Hay casos especiales para cadenas de caracteres donde todos los puntos de código están por debajo de 128, 256 o 65536; de lo contrario, los puntos de código deben estar por debajo de 1114112 (que es el rango completo de Unicode).

*Py_UNICODE ** y las representaciones UTF-8 se crean a pedido y se almacenan en caché en el objeto Unicode. La representación *Py_UNICODE ** está en desuso y es ineficiente; debe evitarse en situaciones sensibles al rendimiento o la memoria.

Debido a la transición entre las API antiguas y las API nuevas, los objetos Unicode pueden estar internamente en dos estados dependiendo de cómo se crearon:

- Los objetos Unicode «canónicos» son todos los objetos creados por una API Unicode no obsoleta. Utilizan la representación más eficiente permitida por la implementación.
- Los objetos Unicode «heredados» se han creado a través de una de las API obsoletas (normalmente *PyUnicode_FromUnicode()*) y solo tienen la representación *Py_UNICODE**; Será necesario llamar a *PyUnicode_READY()* en ellos antes de llamar a cualquier otra API.

Nota: El objeto Unicode «heredado» se eliminará en Python 3.12 con APIs obsoletas. Todos los objetos Unicode serán «canónicos» desde entonces. Consulte **PEP 623** para obtener más información.

Tipo Unicode

Estos son los tipos básicos de objetos Unicode utilizados para la implementación de Unicode en Python:

Py_UCS4

Py_UCS2

Py_UCS1

Estos tipos son definiciones de tipo (*typedefs*) para los tipos “enteros sin signo” (*unsigned int*) lo suficientemente anchos como para contener caracteres de 32 bits, 16 bits y 8 bits, respectivamente. Cuando se trate con caracteres Unicode individuales, use *Py_UCS4*.

Nuevo en la versión 3.3.

Py_UNICODE

Este es una definición de tipo (*typedef*) de *wchar_t*, que es un tipo de 16 bits o de 32 bits dependiendo de la plataforma.

Distinto en la versión 3.3: En versiones anteriores, este era un tipo de 16 bits o de 32 bits, dependiendo de si seleccionó una versión Unicode «estrecha» o «amplia» de Python en el momento de la compilación.

PyASCIIObject

PyCompactUnicodeObject

PyUnicodeObject

Estos subtipos de *PyObject* representan un objeto Python Unicode. En casi todos los casos, no deben usarse directamente, ya que todas las funciones API que se ocupan de objetos Unicode toman y retornan punteros *PyObject*.

Nuevo en la versión 3.3.

PyTypeObject **PyUnicode_Type**

Esta instancia de *PyTypeObject* representa el tipo Python Unicode. Está expuesto al código de Python como *str*.

Las siguientes API son realmente macros de C y se pueden utilizar para realizar comprobaciones rápidas y acceder a datos internos de solo lectura de objetos Unicode:

int **PyUnicode_Check** (*PyObject* **o*)

Retorna verdadero si el objeto *o* es un objeto Unicode o una instancia de un subtipo Unicode.

int **PyUnicode_CheckExact** (*PyObject* **o*)

Retorna verdadero (*True*) si el objeto *o* es un objeto Unicode, pero no una instancia de un subtipo.

int **PyUnicode_READY** (*PyObject* **o*)

Asegura que el objeto de cadena de caracteres *o* esté en la representación «canónica». Esto es necesario antes de usar cualquiera de las macros de acceso que se describen a continuación.

Retorna 0 en caso de éxito y -1 con una excepción establecida en caso de error, que ocurre en particular si falla la asignación de memoria.

Nuevo en la versión 3.3.

Deprecated since version 3.10, will be removed in version 3.12: Esta API será removida con *PyUnicode_FromUnicode()*.

Py_ssize_t **PyUnicode_GET_LENGTH** (*PyObject* **o*)

Retorna la longitud de la cadena de caracteres Unicode, en puntos de código. *o* tiene que ser un objeto Unicode en la representación «canónica» (no marcada).

Nuevo en la versión 3.3.

*Py_UCS1** **PyUnicode_1BYTE_DATA** (*PyObject* **o*)

*Py_UCS2** **PyUnicode_2BYTE_DATA** (*PyObject* **o*)

***Py_UCS4** PyUnicode_4BYTE_DATA (*PyObject* **o*)**

Retorna un puntero a la representación canónica emitida a los tipos enteros UCS1, UCS2 o UCS4 para el acceso directo a los caracteres. No se realizan verificaciones si la representación canónica tiene el tamaño de carácter correcto; use *PyUnicode_KIND()* para seleccionar el macro correcto. Asegúrese de que se haya llamado a *PyUnicode_READY()* antes de acceder a esto.

Nuevo en la versión 3.3.

PyUnicode_WCHAR_KIND**PyUnicode_1BYTE_KIND****PyUnicode_2BYTE_KIND****PyUnicode_4BYTE_KIND**

Retorna los valores de la macro *PyUnicode_KIND()*.

Nuevo en la versión 3.3.

Deprecated since version 3.10, will be removed in version 3.12: *PyUnicode_WCHAR_KIND* está deprecada.

unsigned int PyUnicode_KIND (*PyObject* **o*)

Retorna una de las constantes de tipo *PyUnicode* (ver arriba) que indican cuántos bytes por carácter utiliza este objeto Unicode para almacenar sus datos. *o* tiene que ser un objeto Unicode en la representación «canónica» (no marcada).

Nuevo en la versión 3.3.

void* PyUnicode_DATA (*PyObject* **o*)

Retorna un puntero vacío al búfer Unicode sin formato. *o* tiene que ser un objeto Unicode en la representación «canónica» (no marcada).

Nuevo en la versión 3.3.

void PyUnicode_WRITE (int *kind*, void **data*, *Py_ssize_t* *index*, *Py_UCS4* *value*)

Escribe en una representación canónica *data* (como se obtiene con *PyUnicode_DATA()*). Esta macro no realiza ninguna comprobación de cordura y está diseñado para su uso en bucles. La persona que llama debe almacenar en caché el valor *kind* y el puntero *data* como se obtiene de otras llamadas de la macro. *index* es el índice en la cadena de caracteres (comienza en 0) y *value* es el nuevo valor del punto de código que debe escribirse en esa ubicación.

Nuevo en la versión 3.3.

***Py_UCS4* PyUnicode_READ (int *kind*, void **data*, *Py_ssize_t* *index*)**

Lee un punto de código de una representación canónica *data* (obtenido con *PyUnicode_DATA()*). No se realizan verificaciones ni llamadas preparadas.

Nuevo en la versión 3.3.

***Py_UCS4* PyUnicode_READ_CHAR (*PyObject* **o*, *Py_ssize_t* *index*)**

Lee un carácter de un objeto Unicode *o*, que debe estar en la representación «canónica». Esto es menos eficiente que *PyUnicode_READ()* si realiza varias lecturas consecutivas.

Nuevo en la versión 3.3.

PyUnicode_MAX_CHAR_VALUE (*o*)

Retorna el punto de código máximo adecuado para crear otra cadena de caracteres basada en *o*, que debe estar en la representación «canónica». Esto siempre es una aproximación pero más eficiente que iterar sobre la cadena.

Nuevo en la versión 3.3.

***Py_ssize_t* PyUnicode_GET_SIZE (*PyObject* **o*)**

Retorna el tamaño de la representación en desuso *Py_UNICODE*, en unidades de código (esto incluye pares sustitutos como 2 unidades). *o* tiene que ser un objeto Unicode (no marcado).

Deprecated since version 3.3, will be removed in version 3.12: Parte de la API Unicode de estilo antiguo, por favor migrar para usar *PyUnicode_GET_LENGTH()*.

Py_ssize_t **PyUnicode_GET_DATA_SIZE** (*PyObject* **o*)

Retorna el tamaño de la representación en desuso *Py_UNICODE* en bytes. *o* tiene que ser un objeto Unicode (no marcado).

Deprecated since version 3.3, will be removed in version 3.12: Parte de la API Unicode de estilo antiguo, por favor migrar para usar *PyUnicode_GET_LENGTH()*.

*Py_UNICODE** **PyUnicode_AS_UNICODE** (*PyObject* **o*)

const char* **PyUnicode_AS_DATA** (*PyObject* **o*)

Retorna un puntero a una representación *Py_UNICODE* del objeto. El búfer retornado siempre termina con un punto de código nulo adicional. También puede contener puntos de código nulo incrustados, lo que provocaría que la cadena de caracteres se truncara cuando se usara en la mayoría de las funciones de C. La forma *AS_DATA* arroja el puntero a const char *. El argumento *o* tiene que ser un objeto Unicode (no marcado).

Distinto en la versión 3.3: Esta macro ahora es ineficiente, porque en muchos casos la representación *Py_UNICODE* no existe y necesita ser creada, y puede fallar (retornar NULL con un conjunto de excepciones). Intente portar el código para usar las nuevas macros *PyUnicode_nBYTE_DATA()* o use *PyUnicode_WRITE()* o *PyUnicode_READ()*.

Deprecated since version 3.3, will be removed in version 3.12: Parte de la antigua API Unicode, por favor migre para usar la familia de macros *PyUnicode_nBYTE_DATA()*.

int **PyUnicode_IsIdentifier** (*PyObject* **o*)

Retorna 1 si la cadena de caracteres es un identificador válido de acuerdo con la definición del lenguaje, sección identifiers. Retorna 0 de lo contrario.

Distinto en la versión 3.9: La función ya no llama a *Py_FatalError()* si la cadena de caracteres no está lista.

Propiedades de caracteres Unicode

Unicode proporciona muchas propiedades de caracteres diferentes. Los que se necesitan con mayor frecuencia están disponibles a través de estas macros que se asignan a las funciones de C según la configuración de Python.

int **Py_UNICODE_ISSPACE** (*Py_UCS4* *ch*)

Retorna 1 o 0 dependiendo de si *ch* es un carácter de espacio en blanco.

int **Py_UNICODE_ISLOWER** (*Py_UCS4* *ch*)

Retorna 1 o 0 dependiendo de si *ch* es un carácter en minúscula.

int **Py_UNICODE_ISUPPER** (*Py_UCS4* *ch*)

Retorna 1 o 0 dependiendo de si *ch* es un carácter en mayúscula.

int **Py_UNICODE_ISTITLE** (*Py_UCS4* *ch*)

Retorna 1 o 0 dependiendo de si *ch* es un carácter en caso de título (*titlecase*).

int **Py_UNICODE_ISLINEBREAK** (*Py_UCS4* *ch*)

Retorna 1 o 0 dependiendo de si *ch* es un carácter de salto de línea.

int **Py_UNICODE_ISDECIMAL** (*Py_UCS4* *ch*)

Retorna 1 o 0 dependiendo de si *ch* es un carácter decimal o no.

int **Py_UNICODE_ISDIGIT** (*Py_UCS4* *ch*)

Retorna 1 o 0 dependiendo de si *ch* es un carácter de dígitos.

int **Py_UNICODE_ISNUMERIC** (*Py_UCS4* *ch*)

Retorna 1 o 0 dependiendo de si *ch* es un carácter numérico.

int **Py_UNICODE_ISALPHA** (*Py_UCS4* *ch*)

Retorna 1 o 0 dependiendo de si *ch* es un carácter alfabético.

int **Py_UNICODE_ISALNUM** (*Py_UCS4 ch*)

Retorna 1 o 0 dependiendo de si *ch* es un carácter alfanumérico.

int **Py_UNICODE_ISPRINTABLE** (*Py_UCS4 ch*)

Retorna 1 o 0 dependiendo de si *ch* es un carácter imprimible. Los caracteres no imprimibles son aquellos definidos en la base de datos de caracteres Unicode como «Otro» o «Separador», excepto el espacio ASCII (0x20) que se considera imprimible. (Tenga en cuenta que los caracteres imprimibles en este contexto son aquellos a los que no se debe escapar cuando `repr()` se invoca en una cadena de caracteres. No tiene relación con el manejo de cadenas de caracteres escritas en `sys.stdout` o `sys.stderr`.)

Estas API se pueden usar para conversiones caracteres rápidas y directos:

Py_UCS4 **Py_UNICODE_TOLOWER** (*Py_UCS4 ch*)

Retorna el carácter *ch* convertido a minúsculas.

Obsoleto desde la versión 3.3: Esta función utiliza conversiones simples.

Py_UCS4 **Py_UNICODE_TOUPPER** (*Py_UCS4 ch*)

Retorna el carácter *ch* convertido a mayúsculas.

Obsoleto desde la versión 3.3: Esta función utiliza conversiones simples.

Py_UCS4 **Py_UNICODE_TOTITLE** (*Py_UCS4 ch*)

Retorna el carácter *ch* convertido a formato de título (*titlecase*).

Obsoleto desde la versión 3.3: Esta función utiliza conversiones simples.

int **Py_UNICODE_TODECIMAL** (*Py_UCS4 ch*)

Retorna el carácter *ch* convertido a un entero positivo decimal. Retorna -1 si esto no es posible. Esta macro no genera excepciones.

int **Py_UNICODE_TODIGIT** (*Py_UCS4 ch*)

Retorna el carácter *ch* convertido a un entero de un solo dígito. Retorna -1 si esto no es posible. Esta macro no genera excepciones.

double **Py_UNICODE_TONUMERIC** (*Py_UCS4 ch*)

Retorna el carácter *ch* convertido a doble. retorne -1.0 si esto no es posible. Esta macro no genera excepciones.

Estas API se pueden usar para trabajar con sustitutos:

Py_UNICODE_IS_SURROGATE (*ch*)

Comprueba si *ch* es un sustituto (0xD800 <= *ch* <= 0xDFFF).

Py_UNICODE_IS_HIGH_SURROGATE (*ch*)

Comprueba si *ch* es un sustituto alto (0xD800 <= *ch* <= 0xDFFF).

Py_UNICODE_IS_LOW_SURROGATE (*ch*)

Comprueba si *ch* es un sustituto bajo (0xD800 <= *ch* <= 0xDFFF).

Py_UNICODE_JOIN_SURROGATES (*high*, *low*)

Une dos caracteres sustitutos y retorna un solo valor *Py_UCS4*. *high* y *low* son respectivamente los sustitutos iniciales y finales en un par sustituto.

Creando y accediendo a cadenas de caracteres Unicode

Para crear objetos Unicode y acceder a sus propiedades de secuencia básicas, use estas API:

*PyObject** **PyUnicode_New** (*Py_ssize_t* size, *Py_UCS4* maxchar)

Return value: *New reference.* Crea un nuevo objeto Unicode. *maxchar* debe ser el punto de código máximo que se colocará en la cadena de caracteres. Como una aproximación, se puede redondear al valor más cercano en la secuencia 127, 255, 65535, 1114111.

Esta es la forma recomendada de asignar un nuevo objeto Unicode. Los objetos creados con esta función no se pueden redimensionar.

Nuevo en la versión 3.3.

*PyObject** **PyUnicode_FromKindAndData** (int *kind*, const void **buffer*, *Py_ssize_t* size)

Return value: *New reference.* Crea un nuevo objeto Unicode con el tipo *kind* dado (los valores posibles son *PyUnicode_1BYTE_KIND* etc., según lo retornado por *PyUnicode_KIND()*). El búfer debe apuntar a un vector (*array*) de tamaño unidades de 1, 2 o 4 bytes por carácter, según el tipo.

Nuevo en la versión 3.3.

*PyObject** **PyUnicode_FromStringAndSize** (const char **u*, *Py_ssize_t* size)

Return value: *New reference.* Crea un objeto Unicode desde el búfer de caracteres *u*. Los bytes se interpretarán como codificados en UTF-8. El búfer se copia en el nuevo objeto. Si el búfer no es NULL, el valor de retorno podría ser un objeto compartido, es decir, no se permite la modificación de los datos.

Si *u* es NULL, esta función se comporta como *PyUnicode_FromUnicode()* con el búfer establecido en NULL. Este uso se considera obsoleto (*deprecated*) en favor de *PyUnicode_New()*.

*PyObject** **PyUnicode_FromString** (const char **u*)

Return value: *New reference.* Crea un objeto Unicode a partir de un búfer *u* de caracteres terminado en nulo y codificado en UTF-8.

*PyObject** **PyUnicode_FromFormat** (const char **format*, ...)

Return value: *New reference.* Toma una cadena de caracteres *format* con el estilo de *printf()* en C y un número variable de argumentos, calcula el tamaño de la cadena Python Unicode resultante y retorna una cadena de caracteres con los valores formateados. Los argumentos variables deben ser tipos de C y deben corresponder exactamente a los caracteres de formato en la cadena de caracteres *format* codificada en ASCII. Se permiten los siguientes caracteres de formato:

Formatear caracteres	Tipo	Comentario
%%	<i>n/a</i>	El carácter literal %.
%c	int	Un solo carácter, representado como un entero (<i>int</i>) de C.
%d	int	Equivalente a <code>printf("%d").</code> ¹
%u	unsigned int	Equivalente a <code>printf("%u").</code> ¹
%ld	long	Equivalente a <code>printf("%ld").</code> ¹
%li	long	Equivalente a <code>printf("%li").</code> ¹
%lu	unsigned long	Equivalente a <code>printf("%lu").</code> ¹
%lld	long long	Equivalente a <code>printf("%lld").</code> ¹
%lli	long long	Equivalente a <code>printf("%lli").</code> ¹
%llu	unsigned long long	Equivalente a <code>printf("%llu").</code> ¹
%zd	<code>Py_ssize_t</code>	Equivalente a <code>printf("%zd").</code> ¹
%zi	<code>Py_ssize_t</code>	Equivalente a <code>printf("%zi").</code> ¹
%zu	<code>size_t</code>	Equivalente a <code>printf("%zu").</code> ¹
%i	int	Equivalente a <code>printf("%i").</code> ¹
%x	int	Equivalente a <code>printf("%x").</code> ¹
%s	const char*	Un arreglo de caracteres de C terminada en nulo.
%p	const void*	La representación hexadecimal de un puntero en C. Principalmente equivalente a <code>printf("%p")</code> excepto que se garantiza que comience con el literal 0x, independiente de lo que produzca el <code>printf</code> de la plataforma.
%A	PyObject*	El resultado de llamar <code>ascii()</code> .
%U	PyObject*	Un objeto Unicode.
%V	PyObject*, const char*	Un objeto Unicode (que puede ser NULL) y un arreglo de caracteres de C terminada en nulo como segundo parámetro (que se utilizará, si el primer parámetro es NULL).
%S	PyObject*	El resultado de llamar <code>PyObject_Str()</code> .
%R	PyObject*	El resultado de llamar <code>PyObject_Repr()</code> .

Un carácter de formato no reconocido hace que todo el resto de la cadena de formato se copie tal cual a la cadena de resultado y se descartan los argumentos adicionales.

Nota: La unidad del formateador de ancho es el número de caracteres en lugar de bytes. La unidad del formateador de precisión es la cantidad de bytes para "%s" y "%V" `` (si el argumento ``PyObject* es NULL), y una cantidad de caracteres para "%A", "%U", "%S", "%R" y "%V" (si el argumento PyObject* no es NULL).

Distinto en la versión 3.2: Soporte agregado para "%lld" y "%llu".

Distinto en la versión 3.3: Soporte agregado para "%li", "%lli" y "%zi".

Distinto en la versión 3.4: Soporte agregado para formateadores de anchura y precisión para "%s", "%A", "%U", "%V", "%S", "%R".

*PyObject** **PyUnicode_FromFormatV** (const char *format, va_list args)

Return value: New reference. Idéntico a `PyUnicode_FromFormat()` excepto que toma exactamente dos argumentos.

¹ Para especificadores de enteros (*d, u, ld, li, lu, lld, lli, llu, zd, zi, zu, i, x*): el indicador de conversión 0 tiene efecto incluso cuando se proporciona una precisión.

*PyObject** **PyUnicode_FromEncodedObject** (*PyObject* *obj, const char *encoding, const char *errors)

Return value: New reference. Decodifica un objeto codificado *obj* en un objeto Unicode.

bytes, bytearray y otros *los objetos similares a bytes* se decodifican de acuerdo con el *encoding* dado y utilizan el manejo de errores definido por *errors*. Ambos pueden ser NULL para que la interfaz use los valores predeterminados (ver *Códecs incorporados* para más detalles).

Todos los demás objetos, incluidos los objetos Unicode, hacen que se establezca un `TypeError`.

La API retorna NULL si hubo un error. La entidad que hace la llamadas es la responsable de desreferenciar los objetos retornados.

Py_ssize_t **PyUnicode_GetLength** (*PyObject* *unicode)

Retorna la longitud del objeto Unicode, en puntos de código.

Nuevo en la versión 3.3.

Py_ssize_t **PyUnicode_CopyCharacters** (*PyObject* *to, *Py_ssize_t* to_start, *PyObject* *from, *Py_ssize_t* from_start, *Py_ssize_t* how_many)

Copia caracteres de un objeto Unicode en otro. Esta función realiza la conversión de caracteres cuando es necesario y recurre a `memcpy()` si es posible. Retorna -1 y establece una excepción en caso de error; de lo contrario, retorna el número de caracteres copiados.

Nuevo en la versión 3.3.

Py_ssize_t **PyUnicode_Fill** (*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* length, *Py_UCS4* fill_char)

Rellena una cadena con un carácter: escriba *fill_char* en `unicode[inicio:inicio+longitud]`.

Falla si *fill_char* es más grande que el carácter máximo de la cadena, o si la cadena tiene más de 1 referencia.

Retorna el número de caracteres escritos o retorna -1 y genera una excepción en caso de error.

Nuevo en la versión 3.3.

int **PyUnicode_WriteChar** (*PyObject* *unicode, *Py_ssize_t* index, *Py_UCS4* character)

Escribe un carácter en una cadena de caracteres. La cadena debe haberse creado a través de `PyUnicode_New()`. Dado que se supone que las cadenas de caracteres Unicode son inmutables, la cadena no debe compartirse o no se ha cifrado todavía.

Esta función comprueba que *unicode* es un objeto Unicode, que el índice no está fuera de los límites y que el objeto se puede modificar de forma segura (es decir, si su número de referencia es uno).

Nuevo en la versión 3.3.

Py_UCS4 **PyUnicode_ReadChar** (*PyObject* *unicode, *Py_ssize_t* index)

Lee un carácter de una cadena de caracteres. Esta función verifica que *unicode* es un objeto Unicode y que el índice no está fuera de límites, en contraste con la versión de macro `PyUnicode_READ_CHAR()`.

Nuevo en la versión 3.3.

*PyObject** **PyUnicode_Substring** (*PyObject* *str, *Py_ssize_t* start, *Py_ssize_t* end)

Return value: New reference. Retorna una subcadena de caracteres de *str*, desde el índice de caracteres *start* (incluido) al índice de caracteres *end* (excluido). Los índices negativos no son compatibles.

Nuevo en la versión 3.3.

*Py_UCS4** **PyUnicode_AsUCS4** (*PyObject* *u, *Py_UCS4* *buffer, *Py_ssize_t* buflen, int copy_null)

Copia la cadena de caracteres *u* en un búfer UCS4, incluido un carácter nulo, si *copy_null* está configurado. Retorna NULL y establece una excepción en caso de error (en particular, a `SystemError` si *buflen* es menor que la longitud de *u*). *buffer* se retorna en caso de éxito.

Nuevo en la versión 3.3.

***Py_UCS4** PyUnicode_AsUCS4Copy (*PyObject* **u*)**

Copia la cadena de caracteres *u* en un nuevo búfer UCS4 que se asigna usando *PyMem_Malloc()*. Si esto falla, se retorna NULL con un *MemoryError* establecido. El búfer retornado siempre tiene un punto de código nulo adicional agregado.

Nuevo en la versión 3.3.

APIs de Py_UNICODE deprecadas

Deprecated since version 3.3, will be removed in version 3.12.

Estas funciones API están en desuso con la implementación de **PEP 393**. Los módulos de extensión pueden continuar usándolos, ya que no se eliminarán en Python 3.x, pero deben ser conscientes de que su uso ahora puede causar problemas de rendimiento y memoria.

***PyObject** PyUnicode_FromUnicode (const *Py_UNICODE* **u*, *Py_ssize_t* *size*)**

Return value: *New reference.* Crea un objeto Unicode desde el búfer *Py_UNICODE* *u* del tamaño dado. *u* puede ser NULL, lo que hace que el contenido no esté definido. Es responsabilidad del usuario completar los datos necesarios. El búfer se copia en el nuevo objeto.

Si el búfer no es NULL, el valor de retorno podría ser un objeto compartido. Por lo tanto, la modificación del objeto Unicode resultante solo se permite cuando *u* es NULL.

Si el búfer es NULL, se debe llamar a *PyUnicode_READY()* una vez que se haya llenado el contenido de la cadena de caracteres antes de usar cualquiera de las macros de acceso, como *PyUnicode_KIND()*.

Deprecated since version 3.3, will be removed in version 3.12: Por favor migrar para usar *PyUnicode_FromKindAndData()*, *PyUnicode_FromWideChar()* o *PyUnicode_New()*.

***Py_UNICODE** PyUnicode_AsUnicode (*PyObject* **unicode*)**

Retorna un puntero de solo lectura al búfer *Py_UNICODE* interno del objeto Unicode, o NULL en caso de error. Esto creará la representación *Py_UNICODE** del objeto si aún no está disponible. El búfer siempre termina con un punto de código nulo adicional. Tenga en cuenta que la cadena de caracteres resultante *Py_UNICODE* también puede contener puntos de código nulo incrustados, lo que provocaría que la cadena se truncara cuando se usara en la mayoría de las funciones de C.

Deprecated since version 3.3, will be removed in version 3.12: Parte del estilo antiguo de la API Unicode, por favor migrar para usar *PyUnicode_AsUCS4()*, *PyUnicode_AsWideChar()*, *PyUnicode_ReadChar()* o APIs nuevas similares.

Deprecated since version 3.3, will be removed in version 3.10.

***PyObject** PyUnicode_TransformDecimalToASCII (*Py_UNICODE* **s*, *Py_ssize_t* *size*)**

Return value: *New reference.* Crea un objeto Unicode reemplazando todos los dígitos decimales en el búfer *Py_UNICODE* del *size* dado por dígitos ASCII 0–9 de acuerdo con su valor decimal. Retorna NULL si ocurre una excepción.

Deprecated since version 3.3, will be removed in version 3.11: Parte del estilo antiguo de la API *Py_UNICODE*; por favor migrar para usar *Py_UNICODE_TODECIMAL()*.

***Py_UNICODE** PyUnicode_AsUnicodeAndSize (*PyObject* **unicode*, *Py_ssize_t* **size*)**

Como *PyUnicode_AsUnicode()*, pero también guarda la longitud del arreglo *Py_UNICODE()* (excluyendo el terminador nulo adicional) en *size*. Tenga en cuenta que la cadena de caracteres resultante *Py_UNICODE** puede contener puntos de código nulo incrustados, lo que provocaría que la cadena se truncara cuando se usara en la mayoría de las funciones de C.

Nuevo en la versión 3.3.

Deprecated since version 3.3, will be removed in version 3.12: Parte del estilo antiguo de la API Unicode, por favor migrar para usar `PyUnicode_AsUCS4()`, `PyUnicode_AsWideChar()`, `PyUnicode_ReadChar()` o APIs nuevas similares.

`Py_UNICODE*` `PyUnicode_AsUnicodeCopy` (`PyObject *`*unicode*)

Crea una copia de una cadena de caracteres Unicode que termina con un punto de código nulo. Retorna NULL y genera una excepción `MemoryError` en caso de fallo de asignación de memoria; de lo contrario, retorna un nuevo búfer asignado (use `PyMem_Free()` para liberar el búfer). Tenga en cuenta que la cadena de caracteres resultante `Py_UNICODE*` puede contener puntos de código nulo incrustados, lo que provocaría que la cadena se truncara cuando se usara en la mayoría de las funciones de C.

Nuevo en la versión 3.2.

Por favor migrar para usar `PyUnicode_AsUCS4Copy()` o API nuevas similares.

`Py_ssize_t` `PyUnicode_GetSize` (`PyObject *`*unicode*)

Retorna el tamaño de la representación en desuso `Py_UNICODE`, en unidades de código (esto incluye pares sustitutos como 2 unidades).

Deprecated since version 3.3, will be removed in version 3.12: Parte de la API Unicode de estilo antiguo, por favor migrar para usar `PyUnicode_GET_LENGTH()`.

`PyObject*` `PyUnicode_FromObject` (`PyObject *`*obj*)

Return value: *New reference.* Copia una instancia de un subtipo Unicode a un nuevo objeto Unicode verdadero si es necesario. Si *obj* ya es un verdadero objeto Unicode (no un subtipo), retorna la referencia con un recuento incrementado.

Los objetos que no sean Unicode o sus subtipos causarán un `TypeError`.

Codificación regional

La codificación local actual se puede utilizar para decodificar texto del sistema operativo.

`PyObject*` `PyUnicode_DecodeLocaleAndSize` (`const char *`*str*, `Py_ssize_t` *len*, `const char *`*errors*)

Return value: *New reference.* Decodifica una cadena de caracteres UTF-8 en Android y VxWorks, o de la codificación de configuración regional actual en otras plataformas. Los manejadores de errores admitidos son "estricto" y "subrogateescape" (**PEP 383**). El decodificador usa el controlador de errores "estricto" si *errors* es "NULL". *str* debe terminar con un carácter nulo pero no puede contener caracteres nulos incrustados.

Utilice `PyUnicode_DecodeFSDefaultAndSize()` para decodificar una cadena de `Py_FileSystemDefaultEncoding` (la codificación de la configuración regional leída al iniciar Python).

Esta función ignora el modo Python UTF-8.

Ver también:

La función `Py_DecodeLocale()`.

Nuevo en la versión 3.3.

Distinto en la versión 3.7: La función ahora también usa la codificación de configuración regional actual para el controlador de errores `subrogateescape`, excepto en Android. Anteriormente, `Py_DecodeLocale()` se usaba para el `subrogateescape`, y la codificación local actual se usaba para `estricto`.

`PyObject*` `PyUnicode_DecodeLocale` (`const char *`*str*, `const char *`*errors*)

Return value: *New reference.* Similar a `PyUnicode_DecodeLocaleAndSize()`, pero calcula la longitud de la cadena de caracteres usando `strlen()`.

Nuevo en la versión 3.3.

*PyObject** **PyUnicode_EncodeLocale** (*PyObject* *unicode, const char *errors)

Return value: New reference. Codifica un objeto Unicode UTF-8 en Android y VxWorks, o en la codificación local actual en otras plataformas. Los manejadores de errores admitidos son "estricto" y "subrogatescape" (**PEP 383**). El codificador utiliza el controlador de errores "estricto" si *errors* es NULL. Retorna un objeto bytes. *unicode* no puede contener caracteres nulos incrustados.

Utilice `PyUnicode_EncodeFSDefault()` para codificar una cadena de caracteres en `Py_FileSystemDefaultEncoding` (la codificación de la configuración regional leída al iniciar Python).

Esta función ignora el modo Python UTF-8.

Ver también:

La función `Py_EncodeLocale()`.

Nuevo en la versión 3.3.

Distinto en la versión 3.7: La función ahora también usa la codificación de configuración regional actual para el controlador de errores `subrogatescape`, excepto en Android. Anteriormente, `Py_EncodeLocale()` se usaba para el `subrogatescape`, y la codificación local actual se usaba para `estricto`.

Codificación del sistema de archivos

Para codificar y decodificar nombres de archivo y otras cadenas de caracteres de entorno, `Py_FileSystemDefaultEncoding` debe usarse como codificación, y `Py_FileSystemDefaultEncodeErrors` debe usarse como controlador de errores (**PEP 383** y **PEP 529**). Para codificar nombres de archivo a bytes durante el análisis de argumentos, se debe usar el convertidor "O&", pasando `PyUnicode_FSConverter()` como la función de conversión:

int **PyUnicode_FSConverter** (*PyObject** obj, void* result)

Convertidor `ParseTuple`: codificar objetos `str` – obtenidos directamente o mediante la interfaz `os.PathLike` – a bytes usando `PyUnicode_EncodeFSDefault()`; los objetos bytes se emiten tal cual. *result* debe ser un `PyBytesObject*` que debe liberarse cuando ya no se use.

Nuevo en la versión 3.1.

Distinto en la versión 3.6: Acepta un objeto similar a una ruta (*path-like object*).

Para decodificar nombres de archivo a `str` durante el análisis de argumentos, se debe usar el convertidor "O&", pasando `PyUnicode_FSDecoder()` como la función de conversión:

int **PyUnicode_FSDecoder** (*PyObject** obj, void* result)

Convertor `ParseTuple`: decodifica objetos bytes – obtenidos directa o indirectamente a través de la interfaz `os.PathLike` – a `str` usando `PyUnicode_DecodeFSDefaultAndSize()`; los objetos `str` se generan tal cual. *result* debe ser `PyUnicodeObject*` que debe liberarse cuando ya no se use.

Nuevo en la versión 3.2.

Distinto en la versión 3.6: Acepta un objeto similar a una ruta (*path-like object*).

*PyObject** **PyUnicode_DecodeFSDefaultAndSize** (const char *s, *Py_ssize_t* size)

Return value: New reference. Decodifica una cadena usando `Py_FileSystemDefaultEncoding` y el controlador de errores `Py_FileSystemDefaultEncodeErrors`.

Si `Py_FileSystemDefaultEncoding` no está configurado, recurre a la codificación de configuración regional.

`Py_FileSystemDefaultEncoding` se inicializa al inicio desde la codificación local y no se puede modificar más tarde. Si se necesita decodificar una cadena de caracteres de la codificación local actual, utilice `PyUnicode_DecodeLocaleAndSize()`.

Ver también:

La función `Py_DecodeLocale()`.

Distinto en la versión 3.6: Utilice el controlador de errores `Py_FileSystemDefaultEncodeErrors`.

*PyObject** **PyUnicode_DecodeFSDefault** (const char *s)

Return value: *New reference.* Decodifique una cadena terminada en nulo usando `Py_FileSystemDefaultEncoding` y el `Py_FileSystemDefaultEncodeErrors` controlador de errores.

Si `Py_FileSystemDefaultEncoding` no está configurado, recurre a la codificación de configuración regional.

Utilice `PyUnicode_DecodeFSDefaultAndSize()` si conoce la longitud de la cadena.

Distinto en la versión 3.6: Utilice el controlador de errores `Py_FileSystemDefaultEncodeErrors`.

*PyObject** **PyUnicode_EncodeFSDefault** (*PyObject* *unicode)

Return value: *New reference.* Codifica un objeto Unicode para `Py_FileSystemDefaultEncoding` con el manejador de errores `Py_FileSystemDefaultEncodeErrors`, y retorna bytes. Tenga en cuenta que el objeto resultante bytes puede contener bytes nulos.

Si `Py_FileSystemDefaultEncoding` no está configurado, recurre a la codificación de configuración regional.

`Py_FileSystemDefaultEncoding` se inicializa al inicio desde la codificación local y no se puede modificar más tarde. Si necesita codificar una cadena a la codificación local actual, utilice `PyUnicode_EncodeLocale()`.

Ver también:

La función `Py_EncodeLocale()`.

Nuevo en la versión 3.2.

Distinto en la versión 3.6: Utilice el controlador de errores `Py_FileSystemDefaultEncodeErrors`.

soporte wchar_t

soporte `wchar_t` para plataformas que lo soportan:

*PyObject** **PyUnicode_FromWideChar** (const wchar_t *w, *Py_ssize_t* size)

Return value: *New reference.* Crea un objeto Unicode a partir del búfer `wchar_t` `w` del tamaño `size` dado. Pasar -1 como `size` indica que la función debe calcular la longitud, usando `wcslen`. Retorna NULL en caso de falla.

Py_ssize_t **PyUnicode_AsWideChar** (*PyObject* *unicode, wchar_t *w, *Py_ssize_t* size)

Copia el contenido del objeto Unicode en el búfer `wchar_t` `w`. A lo sumo `size` se copian los caracteres `wchar_t` (excluyendo un posible carácter de terminación nulo final). Retorna el número de caracteres `wchar_t` copiados o -1 en caso de error. Tenga en cuenta que la cadena resultante `wchar_t*` puede o no tener terminación nula. Es responsabilidad de la persona que llama asegurarse de que la cadena `wchar_t*` tenga una terminación nula en caso de que la aplicación lo requiera. Además, tenga en cuenta que la cadena `wchar_t*` podría contener caracteres nulos, lo que provocaría que la cadena se truncara cuando se usara con la mayoría de las funciones de C.

`wchar_t*` **PyUnicode_AsWideCharString** (*PyObject* *unicode, *Py_ssize_t* *size)

Convierte el objeto Unicode en una cadena de caracteres ancha. La cadena de salida siempre termina con un carácter nulo. Si `size` no es NULL, escribe el número de caracteres anchos (excluyendo el carácter de terminación nulo final) en `*size`. Tenga en cuenta que la cadena resultante `wchar_t` podría contener caracteres nulos, lo que provocaría que la cadena se truncara cuando se usara con la mayoría de las funciones de C. Si `size` es NULL y la cadena `wchar_t*` contiene caracteres nulos un `ValueError` aparece.

Retorna un búfer asignado por `PyMem_Alloc()` (utilice `PyMem_Free()` para liberarlo) en caso de éxito. En caso de error, retorna `NULL` y `*size` no está definido. Provoca un `MemoryError` si falla la asignación de memoria.

Nuevo en la versión 3.2.

Distinto en la versión 3.7: Provoca un `ValueError` si `size` es `NULL` y la cadena `wchar_t*` contiene caracteres nulos.

Códecs incorporados

Python proporciona un conjunto de códecs integrados que están escritos en C para mayor velocidad. Todos estos códecs se pueden usar directamente a través de las siguientes funciones.

Muchas de las siguientes API toman dos argumentos de *encoding* y *errors*, y tienen la misma semántica que las del constructor de objetos de cadena incorporado `str()`.

Establecer la codificación en `NULL` hace que se use la codificación predeterminada, que es ASCII. Las llamadas al sistema de archivos deben usar `PyUnicode_FSConverter()` para codificar nombres de archivos. Esto utiliza la variable `Py_FileSystemDefaultEncoding` internamente. Esta variable debe tratarse como de solo lectura: en algunos sistemas, será un puntero a una cadena de caracteres estática, en otros, cambiará en tiempo de ejecución (como cuando la aplicación invoca `setlocale`).

El manejo de errores se establece mediante *errors* que también pueden establecerse en `NULL`, lo que significa usar el manejo predeterminado definido para el códec. El manejo de errores predeterminado para todos los códecs integrados es «estricto» (se lanza `ValueError`).

The codecs all use a similar interface. Only deviations from the following generic ones are documented for simplicity.

Códecs genéricos

Estas son las APIs de códecs genéricos:

*PyObject** **PyUnicode_Decode** (const char *s, *Py_ssize_t* size, const char *encoding, const char *errors)

Return value: New reference. Crea un objeto Unicode decodificando *size* bytes de la cadena codificada *s*. *encoding* y *errors* tienen el mismo significado que los parámetros del mismo nombre en la función incorporada `str()`. El códec que se utilizará se busca utilizando el registro de códec Python. Retorna `NULL` si el códec provocó una excepción.

*PyObject** **PyUnicode_AsEncodedString** (*PyObject* *unicode, const char *encoding, const char *errors)

Return value: New reference. Codifica un objeto Unicode y retorna el resultado como un objeto de bytes de Python. *encoding* y *errors* tienen el mismo significado que los parámetros del mismo nombre en el método Unicode `encode()`. El códec que se utilizará se busca utilizando el registro de códec Python. Retorna `NULL` si el códec provocó una excepción.

*PyObject** **PyUnicode_Encode** (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *encoding, const char *errors)

Return value: New reference. Codifica el búfer *Py_UNICODE* *s* del tamaño *size* dado y retorna un objeto de bytes de Python. *encoding* y *errors* tienen el mismo significado que los parámetros del mismo nombre en el método Unicode `encode()`. El códec que se utilizará se busca utilizando el registro de códec Python. Retorna `NULL` si el códec provocó una excepción.

Deprecated since version 3.3, will be removed in version 3.11: Parte del viejo estilo de la API *Py_UNICODE*; por favor migrar para usar `PyUnicode_AsEncodedString()`.

Códecs UTF-8

Estas son las APIs del códec UTF-8:

*PyObject** **PyUnicode_DecodeUTF8** (const char *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Crea un objeto Unicode decodificando *size* bytes de la cadena codificada UTF-8 *s*. Retorna NULL si el códec provocó una excepción.

*PyObject** **PyUnicode_DecodeUTF8Stateful** (const char *s, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Return value: New reference. Si *consumed* es NULL, se comporta como *PyUnicode_DecodeUTF8()*. Si *consumed* no es NULL, las secuencias de bytes UTF-8 incompletas no se tratarán como un error. Esos bytes no serán decodificados y la cantidad de bytes que han sido decodificados se almacenará en *consumed*.

*PyObject** **PyUnicode_AsUTF8String** (*PyObject* *unicode)

Return value: New reference. Codifica un objeto Unicode usando UTF-8 y retorna el resultado como un objeto de bytes de Python. El manejo de errores es «estricto». Retorna NULL si el códec provocó una excepción.

const char* **PyUnicode_AsUTF8AndSize** (*PyObject* *unicode, *Py_ssize_t* *size)

Retorna un puntero a la codificación UTF-8 del objeto Unicode y almacena el tamaño de la representación codificada (en bytes) en *size*. El argumento *size* puede ser NULL; en este caso no se almacenará el tamaño. El búfer retornado siempre tiene un byte nulo adicional agregado (no incluido en *size*), independientemente de si hay otros puntos de código nulo.

En caso de error, se retorna NULL con un conjunto de excepciones y no se almacena *size*.

This caches the UTF-8 representation of the string in the Unicode object, and subsequent calls will return a pointer to the same buffer. The caller is not responsible for deallocating the buffer. The buffer is deallocated and pointers to it become invalid when the Unicode object is garbage collected.

Nuevo en la versión 3.3.

Distinto en la versión 3.7: El tipo de retorno ahora es `const char *` en lugar de `char *`.

const char* **PyUnicode_AsUTF8** (*PyObject* *unicode)

Como *PyUnicode_AsUTF8AndSize()*, pero no almacena el tamaño.

Nuevo en la versión 3.3.

Distinto en la versión 3.7: El tipo de retorno ahora es `const char *` en lugar de `char *`.

*PyObject** **PyUnicode_EncodeUTF8** (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Codifica el búfer *Py_UNICODE s* del tamaño *size* dado usando UTF-8 y retorna un objeto de bytes de Python. Retorna NULL si el códec provocó una excepción.

Deprecated since version 3.3, will be removed in version 3.11: Parte del viejo estilo de la API *Py_UNICODE*; por favor migrar para usar *PyUnicode_AsUTF8String()*, *PyUnicode_AsUTF8AndSize()* o *PyUnicode_AsEncodedString()*.

Códecs UTF-32

Estas son las APIs de códecs para UTF-32:

*PyObject** **PyUnicode_DecodeUTF32** (const char *s, *Py_ssize_t* size, const char *errors, int *byteorder)

Return value: New reference. Decodifica *size* bytes de una cadena de búfer codificada UTF-32 y retorna el objeto Unicode correspondiente. *errors* (si no es NULL) define el manejo de errores. Su valor predeterminado es «estricto».

Si *byteorder* no es NULL, el decodificador comienza a decodificar utilizando el orden de bytes dado:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

Si `*byteorder` es cero, y los primeros cuatro bytes de los datos de entrada son una marca de orden de bytes (BOM), el decodificador cambia a este orden de bytes y la BOM no se copia en la cadena de caracteres Unicode resultante. Si `*byteorder` es `-1` o `1`, cualquier marca de orden de bytes se copia en la salida.

Una vez completado, `*byteorder` se establece en el orden de bytes actual al final de los datos de entrada.

Si `byteorder` es `NULL`, el códec se inicia en modo de orden nativo.

Retorna `NULL` si el códec provocó una excepción.

PyObject* PyUnicode_DecodeUTF32Stateful (const char *s, Py_ssize_t size, const char *errors, int *byteorder, Py_ssize_t *consumed)

Return value: New reference. Si `consumed` es `NULL`, se comporta como `PyUnicode_DecodeUTF32()`. Si `consumed` no es `NULL`, `PyUnicode_DecodeUTF32Stateful()` no tratará las secuencias de bytes UTF-32 incompletas finales (como un número de bytes no divisible por cuatro) como un error. Esos bytes no serán decodificados y la cantidad de bytes que han sido decodificados se almacenará en `consumed`.

PyObject* PyUnicode_AsUTF32String (PyObject *unicode)

Return value: New reference. Retorna una cadena de bytes de Python usando la codificación UTF-32 en orden de bytes nativo. La cadena siempre comienza con una marca BOM. El manejo de errores es «estricto». Retorna `NULL` si el códec provocó una excepción.

PyObject* PyUnicode_EncodeUTF32 (const Py_UNICODE *s, Py_ssize_t size, const char *errors, int byteorder)

Return value: New reference. Retorna un objeto de bytes de Python que contiene el valor codificado UTF-32 de los datos Unicode en `s`. La salida se escribe de acuerdo con el siguiente orden de bytes:

```
byteorder == -1: little endian
byteorder == 0:  native byte order (writes a BOM mark)
byteorder == 1:  big endian
```

Si `byteorder` es `0`, la cadena de caracteres de salida siempre comenzará con la marca Unicode BOM (U+FEFF). En los otros dos modos, no se antepone ninguna marca BOM.

Si `Py_UNICODE_WIDE` no está definido, los pares sustitutos se mostrarán como un único punto de código.

Retorna `NULL` si el códec provocó una excepción.

Deprecated since version 3.3, will be removed in version 3.11: Parte del viejo estilo de la API `Py_UNICODE`; por favor migrar para usar `PyUnicode_AsUTF32String()` o `PyUnicode_AsEncodedString()`.

Códecs UTF-16

Estas son las APIs de códecs para UTF-16:

PyObject* PyUnicode_DecodeUTF16 (const char *s, Py_ssize_t size, const char *errors, int *byteorder)

Return value: New reference. Decodifica `size` bytes de una cadena de caracteres de búfer codificada UTF-16 y retorna el objeto Unicode correspondiente. `errors` (si no es `NULL`) define el manejo de errores. Su valor predeterminado es «estricto».

Si `byteorder` no es `NULL`, el decodificador comienza a decodificar utilizando el orden de bytes dado:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

Si `*byteorder` es cero, y los primeros dos bytes de los datos de entrada son una marca de orden de bytes (BOM), el decodificador cambia a este orden de bytes y la BOM no se copia en la cadena de caracteres Unicode resultante. Si `*byteorder` es `-1` o `1`, cualquier marca de orden de bytes se copia en la salida (donde dará como resultado un `\uffeff` o un carácter `\ufffe`).

After completion, `*byteorder` is set to the current byte order at the end of input data.

Si `byteorder` es `NULL`, el códec se inicia en modo de orden nativo.

Retorna `NULL` si el códec provocó una excepción.

PyObject* PyUnicode_DecodeUTF16Stateful (const char *s, Py_ssize_t size, const char *errors, int *byteorder, Py_ssize_t *consumed)

Return value: New reference. Si `consumed` es `NULL`, se comporta como `PyUnicode_DecodeUTF16()`. Si `consumed` no es `NULL`, `PyUnicode_DecodeUTF16Stateful()` no tratará las secuencias de bytes UTF-16 incompletas finales (como un número impar de bytes o un par sustituto dividido) como un error. Esos bytes no serán decodificados y la cantidad de bytes que han sido decodificados se almacenará en `consumed`.

PyObject* PyUnicode_AsUTF16String (PyObject *unicode)

Return value: New reference. Retorna una cadena de bytes de Python usando la codificación UTF-16 en orden de bytes nativo. La cadena siempre comienza con una marca BOM. El manejo de errores es «estricto». Retorna `NULL` si el códec provocó una excepción.

PyObject* PyUnicode_EncodeUTF16 (const Py_UNICODE *s, Py_ssize_t size, const char *errors, int byteorder)

Return value: New reference. Retorna un objeto de bytes de Python que contiene el valor codificado UTF-16 de los datos Unicode en `s`. La salida se escribe de acuerdo con el siguiente orden de bytes:

```
byteorder == -1: little endian
byteorder == 0: native byte order (writes a BOM mark)
byteorder == 1: big endian
```

Si `byteorder` es `0`, la cadena de caracteres de salida siempre comenzará con la marca Unicode BOM (U+FEFF). En los otros dos modos, no se antepone ninguna marca BOM.

Si se define `Py_UNICODE_WIDE`, un solo valor de `Py_UNICODE` puede representarse como un par sustituto. Si no está definido, cada uno de los valores `Py_UNICODE` se interpreta como un carácter UCS-2.

Retorna `NULL` si el códec provocó una excepción.

Deprecated since version 3.3, will be removed in version 3.11: Parte del viejo estilo de la API `Py_UNICODE`; por favor migrar para usar `PyUnicode_AsUTF16String()` o `PyUnicode_AsEncodedString()`.

Códecs UTF-7

Estas son las APIs del códec UTF-7:

PyObject* PyUnicode_DecodeUTF7 (const char *s, Py_ssize_t size, const char *errors)

Return value: New reference. Crea un objeto Unicode decodificando `size` bytes de la cadena de caracteres codificada UTF-7 `s`. Retorna `NULL` si el códec provocó una excepción.

PyObject* PyUnicode_DecodeUTF7Stateful (const char *s, Py_ssize_t size, const char *errors, Py_ssize_t *consumed)

Return value: New reference. Si `consumed` es `NULL`, se comporta como `PyUnicode_DecodeUTF7()`. Si `consumed` no es `NULL`, las secciones UTF-7 base-64 incompletas no se tratarán como un error. Esos bytes no serán decodificados y la cantidad de bytes que han sido decodificados se almacenará en `consumed`.

PyObject* PyUnicode_EncodeUTF7 (const Py_UNICODE *s, Py_ssize_t size, int base64SetO, int base64WhiteSpace, const char *errors)

Return value: New reference. Codifica el búfer `Py_UNICODE` del tamaño dado usando UTF-7 y retorna un objeto

de bytes de Python. Retorna NULL si el códec provocó una excepción.

Si *base64SetO* no es cero, «Set O» (puntuación que no tiene un significado especial) se codificará en base-64. Si *base64WhiteSpace* no es cero, el espacio en blanco se codificará en base-64. Ambos se establecen en cero para el códec Python «utf-7».

Deprecated since version 3.3, will be removed in version 3.11: Parte del viejo estilo de la API *Py_UNICODE*; por favor migrar para usar *PyUnicode_AsEncodedString()*.

Códecs Unicode escapado

Estas son las APIs de códecs para Unicode escapado:

*PyObject** **PyUnicode_DecodeUnicodeEscape** (const char *s, *Py_ssize_t* size, const char *errors)
Return value: New reference. Crea un objeto Unicode decodificando *size* bytes de la cadena codificada Unicode escapada (*Unicode-Escape*) s. Retorna NULL si el códec provocó una excepción.

*PyObject** **PyUnicode_AsUnicodeEscapeString** (*PyObject* *unicode)
Return value: New reference. Codifica un objeto Unicode usando Unicode escapado (*Unicode-Escape*) y retorna el resultado como un objeto de bytes. El manejo de errores es «estricto». Retorna NULL si el códec provocó una excepción.

*PyObject** **PyUnicode_EncodeUnicodeEscape** (const *Py_UNICODE* *s, *Py_ssize_t* size)
Return value: New reference. Codifica el búfer *Py_UNICODE* del tamaño *size* dado utilizando Unicode escapado y retorna un objeto de bytes. Retorna NULL si el códec provocó una excepción.

Deprecated since version 3.3, will be removed in version 3.11: Parte del viejo estilo de la API *Py_UNICODE*; por favor migrar para usar *PyUnicode_AsUnicodeEscapeString()*.

Códecs para Unicode escapado en bruto

Estas son las API del códec Unicode escapado en bruto (*Raw Unicode Escape*):

*PyObject** **PyUnicode_DecodeRawUnicodeEscape** (const char *s, *Py_ssize_t* size, const char *errors)
Return value: New reference. Crea un objeto Unicode decodificando *size* bytes de la cadena de caracteres codificada Unicode escapada en bruto (*Raw-Unicode-Escape*) s. Retorna NULL si el códec provocó una excepción.

*PyObject** **PyUnicode_AsRawUnicodeEscapeString** (*PyObject* *unicode)
Return value: New reference. Codifica un objeto Unicode usando Unicode escapado en bruto (*Raw-Unicode-Escape*) y retorna el resultado como un objeto de bytes. El manejo de errores es «estricto». Retorna NULL si el códec provocó una excepción.

*PyObject** **PyUnicode_EncodeRawUnicodeEscape** (const *Py_UNICODE* *s, *Py_ssize_t* size)
Return value: New reference. Codifica el búfer *Py_UNICODE* del tamaño *size* dado usando Unicode escapado en bruto (*Raw-Unicode-Escape*) y retorna un objeto de bytes. Retorna NULL si el códec provocó una excepción.

Deprecated since version 3.3, will be removed in version 3.11: Parte del viejo estilo de la API *Py_UNICODE*; por favor migrar para usar *PyUnicode_AsRawUnicodeEscapeString()* o *PyUnicode_AsEncodedString()*.

Códecs Latin-1

Estas son las API del códec Latin-1: Latin-1 corresponde a los primeros 256 ordinales Unicode y solo estos son aceptados por los códecs durante la codificación.

*PyObject** **PyUnicode_DecodeLatin1** (const char *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Crea un objeto Unicode decodificando *size* bytes de la cadena de caracteres codificada en latin-1 s. Retorna NULL si el códec provocó una excepción.

*PyObject** **PyUnicode_AsLatin1String** (*PyObject* *unicode)

Return value: New reference. Codifica un objeto Unicode usando Latin-1 y retorna el resultado como un objeto de bytes Python. El manejo de errores es «estricto». Retorna NULL si el códec provocó una excepción.

*PyObject** **PyUnicode_EncodeLatin1** (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Codifica el búfer *Py_UNICODE* del tamaño *size* dado usando Latin-1 y retorna un objeto de bytes de Python. Retorna NULL si el códec provocó una excepción.

Deprecated since version 3.3, will be removed in version 3.11: Parte del viejo estilo de la API *Py_UNICODE*; por favor migrar para usar *PyUnicode_AsLatin1String()* o *PyUnicode_AsEncodedString()*.

Códecs ASCII

Estas son las API del códec ASCII. Solo se aceptan datos ASCII de 7 bits. Todos los demás códigos generan errores.

*PyObject** **PyUnicode_DecodeASCII** (const char *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Crea un objeto Unicode decodificando *size* bytes de la cadena de caracteres codificada ASCII s. Retorna NULL si el códec provocó una excepción.

*PyObject** **PyUnicode_AsASCIIString** (*PyObject* *unicode)

Return value: New reference. Codifica un objeto Unicode usando ASCII y retorna el resultado como un objeto de bytes de Python. El manejo de errores es «estricto». Retorna NULL si el códec provocó una excepción.

*PyObject** **PyUnicode_EncodeASCII** (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Codifica el búfer *Py_UNICODE* del tamaño *size* dado utilizando ASCII y retorna un objeto de bytes de Python. Retorna NULL si el códec provocó una excepción.

Deprecated since version 3.3, will be removed in version 3.11: Parte del viejo estilo de la API *Py_UNICODE*; por favor migrar para usar *PyUnicode_AsASCIIString()* o *PyUnicode_AsEncodedString()*.

Códecs de mapa de caracteres

This codec is special in that it can be used to implement many different codecs (and this is in fact what was done to obtain most of the standard codecs included in the `encodings` package). The codec uses mappings to encode and decode characters. The mapping objects provided must support the `__getitem__()` mapping interface; dictionaries and sequences work well.

Estos son las API de códec de mapeo:

*PyObject** **PyUnicode_DecodeCharmap** (const char *data, *Py_ssize_t* size, *PyObject* *mapping, const char *errors)

Return value: New reference. Crea un objeto Unicode decodificando *size* bytes de la cadena de caracteres codificada s usando el objeto *mapping* dado. Retorna NULL si el códec provocó una excepción.

Si *mapping* es NULL, se aplicará la decodificación Latin-1. De lo contrario, *mapping* debe asignar bytes ordinales (enteros en el rango de 0 a 255) a cadenas de caracteres Unicode, enteros (que luego se interpretan como ordinales Unicode) o None. Los bytes de datos sin asignar - los que causan un `LookupError`, así como los que se asignan a None, `0xFFFFE` o `'\ uffffe'`, se tratan como asignaciones indefinidas y causan un error.

*PyObject** **PyUnicode_AsCharmapString** (*PyObject* *unicode, *PyObject* *mapping)

Return value: *New reference.* Codifica un objeto Unicode usando el objeto *mapping* dado y retorna el resultado como un objeto de bytes. El manejo de errores es «estricto». Retorna NULL si el códec provocó una excepción.

El objeto *mapping* debe asignar enteros ordinales Unicode a objetos de bytes, enteros en el rango de 0 a 255 o None. Los ordinales de caracteres no asignados (los que causan un `LookupError`), así como los asignados a Ninguno, se tratan como «mapeo indefinido» y causan un error.

*PyObject** **PyUnicode_EncodeCharmap** (const *Py_UNICODE* *s, *Py_ssize_t* size, *PyObject* *mapping, const char *errors)

Return value: *New reference.* Codifica el búfer *Py_UNICODE* del tamaño *size* dado utilizando el objeto *mapping* dado y retorna el resultado como un objeto de bytes. Retorna NULL si el códec provocó una excepción.

Deprecated since version 3.3, will be removed in version 3.11: Parte del viejo estilo de la API *Py_UNICODE*; por favor migrar para usar *PyUnicode_AsCharmapString()* o *PyUnicode_AsEncodedString()*.

La siguiente API de códec es especial en que asigna Unicode a Unicode.

*PyObject** **PyUnicode_Translate** (*PyObject* *str, *PyObject* *table, const char *errors)

Return value: *New reference.* Traduce una cadena de caracteres aplicando una tabla de mapeo y retornando el objeto Unicode resultante. Retorna NULL cuando el códec provocó una excepción.

La tabla de mapeo debe mapear enteros ordinales Unicode a enteros ordinales Unicode o None (causando la eliminación del carácter).

Las tablas de mapeo solo necesitan proporcionar la interfaz `__getitem__()`; Los diccionarios y las secuencias funcionan bien. Los ordinales de caracteres no asignados (los que causan un `LookupError`) se dejan intactos y se copian tal cual.

errors tiene el significado habitual para los códecs. Puede ser NULL, lo que indica que debe usar el manejo de errores predeterminado.

*PyObject** **PyUnicode_TranslateCharmap** (const *Py_UNICODE* *s, *Py_ssize_t* size, *PyObject* *mapping, const char *errors)

Return value: *New reference.* Traduce un búfer *Py_UNICODE* del tamaño *size* dado al aplicarle una tabla de *mapping* de caracteres y retornar el objeto Unicode resultante. Retorna NULL cuando el códec provocó una excepción.

Deprecated since version 3.3, will be removed in version 3.11: Parte del viejo estilo de la API *Py_UNICODE*; por favor migrar para usar *PyUnicode_Translate()* o *generic codec based API*

Códecs MBCS para Windows

Estas son las API de códec MBCS. Actualmente solo están disponibles en Windows y utilizan los convertidores Win32 MBCS para implementar las conversiones. Tenga en cuenta que MBCS (o DBCS) es una clase de codificaciones, no solo una. La codificación de destino está definida por la configuración del usuario en la máquina que ejecuta el códec.

*PyObject** **PyUnicode_DecodeMBCS** (const char *s, *Py_ssize_t* size, const char *errors)

Return value: *New reference.* Crea un objeto Unicode decodificando *size* bytes de la cadena de caracteres codificada con MBCS *s*. Retorna NULL si el códec provocó una excepción.

*PyObject** **PyUnicode_DecodeMBCSStateful** (const char *s, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Return value: *New reference.* Si *consumed* es NULL, se comporta como *PyUnicode_DecodeMBCS()*. Si *consumed* no es NULL, *PyUnicode_DecodeMBCSStateful()* no decodificará el byte inicial y el número de bytes que se han decodificado se almacenará en *consumed*.

*PyObject** **PyUnicode_AsMBCSString** (*PyObject* *unicode)

Return value: *New reference.* Codifica un objeto Unicode usando MBCS y retorna el resultado como un objeto de bytes de Python. El manejo de errores es «estricto». Retorna NULL si el códec provocó una excepción.

*PyObject** **PyUnicode_EncodeCodePage** (int *code_page*, *PyObject* **unicode*, const char **errors*)

Return value: New reference. Codifica el objeto Unicode utilizando la página de códigos especificada y retorna un objeto de bytes de Python. Retorna NULL si el códec provocó una excepción. Use la página de códigos CP_ACP para obtener el codificador MBCS.

Nuevo en la versión 3.3.

*PyObject** **PyUnicode_EncodeMBCS** (const *Py_UNICODE* **s*, *Py_ssize_t* *size*, const char **errors*)

Return value: New reference. Codifica el búfer *Py_UNICODE* del tamaño *size* dado usando MBCS y retorna un objeto de bytes de Python. Retorna NULL si el códec provocó una excepción.

Deprecated since version 3.3, will be removed in version 4.0: Parte del viejo estilo *Py_UNICODE* de la API; por favor migrar a *PyUnicode_AsMBCSString()*, *PyUnicode_EncodeCodePage()* o *PyUnicode_AsEncodedString()*.

Métodos & Ranuras (Slots)

Métodos y funciones de ranura (Slot)

Las siguientes API son capaces de manejar objetos Unicode y cadenas de caracteres en la entrada (nos referimos a ellos como cadenas de caracteres en las descripciones) y retorna objetos Unicode o enteros según corresponda.

Todos retornan NULL o -1 si ocurre una excepción.

*PyObject** **PyUnicode_Concat** (*PyObject* **left*, *PyObject* **right*)

Return value: New reference. Une dos cadenas de caracteres que dan una nueva cadena de caracteres Unicode.

*PyObject** **PyUnicode_Split** (*PyObject* **s*, *PyObject* **sep*, *Py_ssize_t* *maxsplit*)

Return value: New reference. Divide una cadena de caracteres dando una lista de cadenas de caracteres Unicode. Si *sep* es NULL, la división se realizará en todas las subcadenas de espacios en blanco. De lo contrario, las divisiones ocurren en el separador dado. A lo sumo se realizarán *maxsplit* divisiones. Si es negativo, no se establece ningún límite. Los separadores no están incluidos en la lista resultante.

*PyObject** **PyUnicode_Splitlines** (*PyObject* **s*, int *keepend*)

Return value: New reference. Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If *keepend* is 0, the line break characters are not included in the resulting strings.

*PyObject** **PyUnicode_Join** (*PyObject* **separator*, *PyObject* **seq*)

Return value: New reference. Une una secuencia de cadenas de caracteres usando el *separator* dado y retorna la cadena de caracteres Unicode resultante.

Py_ssize_t **PyUnicode_Tailmatch** (*PyObject* **str*, *PyObject* **substr*, *Py_ssize_t* *start*, *Py_ssize_t* *end*, int *direction*)

Retorna 1 si *substr* coincide con *str*[*start*:*end*] en el final de cola dado (*direction* == -1 significa hacer una coincidencia de prefijo, *direction* == 1 una coincidencia de sufijo), 0 de lo contrario. retorne -1 si ocurrió un error.

Py_ssize_t **PyUnicode_Find** (*PyObject* **str*, *PyObject* **substr*, *Py_ssize_t* *start*, *Py_ssize_t* *end*, int *direction*)

Retorna la primera posición de *substr* en *str*[*start*:*end*] usando la *direction* dada (*direction* == 1 significa hacer una búsqueda hacia adelante, *direction* == -1 una búsqueda hacia atrás). El valor de retorno es el índice de la primera coincidencia; un valor de -1 indica que no se encontró ninguna coincidencia, y -2 indica que se produjo un error y se ha establecido una excepción.

Py_ssize_t **PyUnicode_FindChar** (*PyObject* **str*, *Py_UCS4* *ch*, *Py_ssize_t* *start*, *Py_ssize_t* *end*, int *direction*)

Retorna la primera posición del carácter *ch* en *str*[*inicio*:*fin*] usando la *direction* dada (*direction* == 1 significa hacer una búsqueda hacia adelante, *direction* == -1 una búsqueda hacia atrás). El valor de retorno es el índice de la primera coincidencia; un valor de -1 indica que no se encontró ninguna coincidencia, y -2 indica que se produjo un error y se ha establecido una excepción.

Nuevo en la versión 3.3.

Distinto en la versión 3.7: *start* y *end* ahora están ajustados para comportarse como `str[start:end]`.

Py_ssize_t **PyUnicode_Count** (*PyObject* **str*, *PyObject* **substr*, *Py_ssize_t* *start*, *Py_ssize_t* *end*)

Retorna el número de ocurrencias no superpuestas de *substr* en `str[start:end]`. Retorna `-1` si ocurrió un error.

*PyObject** **PyUnicode_Replace** (*PyObject* **str*, *PyObject* **substr*, *PyObject* **replstr*, *Py_ssize_t* *maxcount*)

Return value: New reference. Reemplaza como máximo *maxcount* ocurrencias de *substr* en *str* con *replstr* y retorna el objeto Unicode resultante. *maxcount* == `-1` significa reemplazar todas las ocurrencias.

int **PyUnicode_Compare** (*PyObject* **left*, *PyObject* **right*)

Compara dos cadenas de caracteres y retorna `-1`, `0`, `1` para menor que, igual y mayor que, respectivamente.

Esta función retorna `-1` en caso de falla, por lo que se debe llamar a `PyErr_Occurred()` para verificar si hay errores.

int **PyUnicode_CompareWithASCIIString** (*PyObject* **uni*, const char **string*)

Compare un objeto Unicode, *uni*, con *string* y retorna `-1`, `0`, `1` para menor que, igual y mayor que, respectivamente. Es mejor pasar solo cadenas de caracteres codificadas en ASCII, pero la función interpreta la cadena de entrada como ISO-8859-1 si contiene caracteres no ASCII.

Esta función no genera excepciones.

*PyObject** **PyUnicode_RichCompare** (*PyObject* **left*, *PyObject* **right*, int *op*)

Return value: New reference. Comparación enriquecida de dos cadenas de caracteres Unicode y retorna uno de los siguientes:

- `NULL` en caso de que se produzca una excepción
- `Py_True` o `Py_False` para comparaciones exitosas
- `Py_NotImplemented` en caso que se desconozca la combinación de tipos

Los posibles valores para *op* son `Py_GT`, `Py_GE`, `Py_EQ`, `Py_NE`, `Py_LT`, y `Py_LE`.

*PyObject** **PyUnicode_Format** (*PyObject* **format*, *PyObject* **args*)

Return value: New reference. Retorna un nuevo objeto de cadena de caracteres desde *format* y *args*; esto es análogo al `format % args`.

int **PyUnicode_Contains** (*PyObject* **container*, *PyObject* **element*)

Comprueba si *element* está contenido en *container* y retorna verdadero o falso en consecuencia.

element tiene que convertir a una cadena de caracteres Unicode. Se retorna `-1` si hubo un error.

void **PyUnicode_InternInPlace** (*PyObject* ***string*)

Interna el argumento *string* en su lugar. El argumento debe ser la dirección de una variable de puntero que apunta a un objeto Unicode de cadena de caracteres Python. Si hay una cadena de caracteres interna existente que es igual a *string*, establece *string* (disminuyendo el recuento de referencias del objeto de cadena de caracteres anterior e incrementando el recuento de referencias del objeto de cadena de caracteres interna), de lo contrario deja solo *string* y lo interna (incrementando su recuento de referencias). (Aclaración: a pesar de que se habla mucho sobre el recuento de referencias, piense en esta función como neutral de recuento de referencia; usted es el propietario del objeto después de la llamada si y solo si lo tenía antes de la llamada).

*PyObject** **PyUnicode_InternFromString** (const char **v*)

Return value: New reference. Una combinación de `PyUnicode_FromString()` y `PyUnicode_InternInPlace()`, que retorna un nuevo objeto de cadena de caracteres Unicode que ha sido creado internamente o una nueva referencia («propia») a un objeto de cadena de caracteres interno anterior con el mismo valor.

8.3.4 Objetos Tuplas

PyTupleObject

Este subtipo de *PyObject* representa un objeto tupla de Python.

PyTypeObject PyTuple_Type

Esta instancia de *PyTypeObject* representa el tipo tupla de Python; es el mismo objeto que `tuple` en la capa de Python.

int PyTuple_Check (PyObject *p)

Retorna verdadero si *p* es un objeto tupla o una instancia de un subtipo del tipo tupla. Esta función siempre finaliza con éxito.

int PyTuple_CheckExact (PyObject *p)

Retorna verdadero si *p* es un objeto tupla pero no una instancia de un subtipo del tipo tupla. Esta función siempre finaliza con éxito.

PyObject* PyTuple_New (Py_ssize_t len)

Return value: New reference. Retorna un nuevo objeto tupla de tamaño *len* o NULL en caso de falla.

PyObject* PyTuple_Pack (Py_ssize_t n, ...)

Return value: New reference. Retorna un nuevo objeto tupla de tamaño *n*, o NULL en caso de falla. Los valores de tupla se inicializan en los argumentos C posteriores *n* que apuntan a objetos de Python. `PyTuple_Pack (2, a, b)` es equivalente a `Py_BuildValue ("OO", a, b)`.

Py_ssize_t PyTuple_Size (PyObject *p)

Toma un puntero a un objeto de tupla y retorna el tamaño de esa tupla.

Py_ssize_t PyTuple_GET_SIZE (PyObject *p)

Retorna el tamaño de la tupla *p*, que no debe ser NULL y apunta a una tupla; No se realiza ninguna comprobación de errores.

PyObject* PyTuple_GetItem (PyObject *p, Py_ssize_t pos)

Return value: Borrowed reference. Retorna el objeto en la posición *pos* en la tupla señalada por *p*. Si *pos* está fuera de los límites, retorna NULL y establece una excepción `IndexError`.

PyObject* PyTuple_GET_ITEM (PyObject *p, Py_ssize_t pos)

Return value: Borrowed reference. Como `PyTuple_GetItem()`, pero no verifica sus argumentos.

PyObject* PyTuple_GetSlice (PyObject *p, Py_ssize_t low, Py_ssize_t high)

Return value: New reference. Retorna la porción de la tupla señalada por *p* entre *low* y *high*, o NULL en caso de falla. Este es el equivalente de la expresión de Python `p[bajo:alto]`. La indexación desde el final de la lista no es compatible.

int PyTuple_SetItem (PyObject *p, Py_ssize_t pos, PyObject *o)

Inserta una referencia al objeto *o* en la posición *pos* de la tupla señalada por *p*. Retorna 0 en caso de éxito. Si *pos* está fuera de límites, retorna -1 y establece una excepción `IndexError`.

Nota: Esta función «roba» una referencia a *o* y descarta una referencia a un elemento que ya está en la tupla en la posición afectada.

void PyTuple_SET_ITEM (PyObject *p, Py_ssize_t pos, PyObject *o)

Al igual que `PyTuple_SetItem()`, pero no realiza ninguna comprobación de errores, y debe *solo* usarse para completar tuplas nuevas.

Nota: Este macro «roba» una referencia a *o* y, a diferencia de `PyTuple_SetItem()`, no descarta una referencia a ningún elemento que se está reemplazando; cualquier referencia en la tupla en la posición *pos* se filtrará.

`int _PyTuple_Resize (PyObject **p, Py_ssize_t newsize)`

Se puede usar para cambiar el tamaño de una tupla. *newsize* será el nuevo tamaño de la tupla. Debido a que se supone que las tuplas son inmutables, esto solo debe usarse si solo hay una referencia al objeto. No use esto si la tupla ya puede ser conocida por alguna otra parte del código. La tupla siempre crecerá o disminuirá al final. Piense en esto como destruir la antigua tupla y crear una nueva, solo que de manera más eficiente. Retorna 0 en caso de éxito. El código del cliente nunca debe suponer que el valor resultante de **p* será el mismo que antes de llamar a esta función. Si se reemplaza el objeto referenciado por **p*, se destruye el original **p*. En caso de fallo, retorna -1 y establece **p* en NULL, y lanza `MemoryError` o `SystemError`.

8.3.5 Objetos de secuencia de estructura

Los objetos de secuencia de estructura son el equivalente en C de los objetos `namedtuple()`, es decir, una secuencia a cuyos elementos también se puede acceder a través de atributos. Para crear una secuencia de estructura, primero debe crear un tipo de secuencia de estructura específico.

*PyObject** **PyStructSequence_NewType** (*PyStructSequence_Desc* *desc)

Return value: New reference. Crea un nuevo tipo de secuencia de estructura a partir de los datos en *desc*, que se describen a continuación. Las instancias del tipo resultante se pueden crear con `PyStructSequence_New()`.

void **PyStructSequence_InitType** (*PyObject* *type, *PyStructSequence_Desc* *desc)

Inicializa una secuencia de estructura tipo *type* desde *desc* en su lugar.

int **PyStructSequence_InitType2** (*PyObject* *type, *PyStructSequence_Desc* *desc)

Lo mismo que `PyStructSequence_InitType`, pero retorna 0 en caso de éxito y -1 en caso de error.

Nuevo en la versión 3.4.

PyStructSequence_Desc

Contiene la meta información de un tipo de secuencia de estructura para crear.

Campo	Tipo C	Significado
name	const char *	nombre del tipo de secuencia de estructura
doc	const char *	puntero al <i>docstring</i> para el tipo o NULL para omitir
fields	<code>PyStructSequence_Field</code> *	puntero al arreglo terminado en NULL con nombres de campo del nuevo tipo
n_in_sequence	int	cantidad de campos visibles para el lado de Python (si se usa como tupla)

PyStructSequence_Field

Describe un campo de una secuencia de estructura. Como una secuencia de estructura se modela como una tupla, todos los campos se escriben como `PyObject*`. El índice en el arreglo *fields* de `PyStructSequence_Desc` determina qué campo de la secuencia de estructura se describe.

Cam-po	Tipo C	Significado
name	const char *	nombre para el campo o NULL para finalizar la lista de campos con nombre, establece en <code>PyStructSequence_UnnamedField</code> para dejar sin nombre
doc	const char *	campo <i>docstring</i> o NULL para omitir

const char * const **PyStructSequence_UnnamedField**

Valor especial para un nombre de campo para dejarlo sin nombre.

Distinto en la versión 3.9: El tipo se cambió de `char *`.

*PyObject** **PyStructSequence_New** (*PyObject* *type)

Return value: *New reference.* Crea una instancia de *type*, que debe haberse creado con *PyStructSequence_NewType* ().

*PyObject** **PyStructSequence_GetItem** (*PyObject* *p, *Py_ssize_t* pos)

Return value: *Borrowed reference.* Retorna el objeto en la posición *pos* en la secuencia de estructura apuntada por *p*. No se realiza la comprobación de límites.

*PyObject** **PyStructSequence_GET_ITEM** (*PyObject* *p, *Py_ssize_t* pos)

Return value: *Borrowed reference.* Macro equivalente de *PyStructSequence_GetItem* ().

void **PyStructSequence_SetItem** (*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Establece el campo en el índice *pos* de la secuencia de estructura *p* en el valor *o*. Como *PyTuple_SET_ITEM* (), esto solo debe usarse para completar instancias nuevas.

Nota: Esta función «roba» una referencia a *o*.

void **PyStructSequence_SET_ITEM** (*PyObject* *p, *Py_ssize_t* *pos, *PyObject* *o)

Macro equivalente de *PyStructSequence_SetItem* ().

Nota: Esta función «roba» una referencia a *o*.

8.3.6 Objetos lista

PyListObject

Este subtipo de *PyObject* representa un objeto lista de Python.

PyObject **PyList_Type**

Esta instancia de *PyObject* representa el tipo de lista de Python. Este es el mismo objeto que *list* en la capa de Python.

int **PyList_Check** (*PyObject* *p)

Retorna verdadero si *p* es un objeto de lista o una instancia de un subtipo del tipo lista. Esta función siempre finaliza con éxito.

int **PyList_CheckExact** (*PyObject* *p)

Retorna verdadero si *p* es un objeto lista, pero no una instancia de un subtipo del tipo lista. Esta función siempre finaliza con éxito.

*PyObject** **PyList_New** (*Py_ssize_t* len)

Return value: *New reference.* Retorna una nueva lista de longitud *len* en caso de éxito o NULL en caso de error.

Nota: Si *len* es mayor que cero, los elementos del objeto de la lista retornada se establecen en NULL. Por lo tanto, no puede utilizar funciones API abstractas como *PySequence_SetItem* () o exponer el objeto al código Python antes de configurar todos los elementos en un objeto real con *PyList_SetItem* ().

Py_ssize_t **PyList_Size** (*PyObject* *list)

Retorna la longitud del objeto lista en *list*; esto es equivalente a *len(list)* en un objeto lista.

Py_ssize_t **PyList_GET_SIZE** (*PyObject* *list)

Forma macro de *PyList_Size* () sin comprobación de errores.

*PyObject** **PyList_GetItem** (*PyObject* *list, *Py_ssize_t* index)

Return value: *Borrowed reference.* Retorna el objeto en la posición *index* en la lista a la que apunta *list*. La posición

no debe ser negativa; La indexación desde el final de la lista no es compatible. Si *index* está fuera de los límites (< 0 o $\geq \text{len}(\text{list})$), retorna `NULL` y establece una excepción `IndexError`.

*PyObject** **PyList_GET_ITEM** (*PyObject* *list, *Py_ssize_t* i)

Return value: Borrowed reference. Forma macro de `PyList_GetItem()` sin comprobación de errores.

int **PyList_SetItem** (*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

Establece el elemento en el índice *index* en la lista a *item*. Retorna 0 en caso de éxito. Si *index* está fuera de límites, retorna -1 y establece una excepción `IndexError`.

Nota: Esta función «roba» una referencia a *item* y descarta una referencia a un elemento que ya está en la lista en la posición afectada.

void **PyList_SET_ITEM** (*PyObject* *list, *Py_ssize_t* i, *PyObject* *o)

Forma macro de `PyList_SetItem()` sin comprobación de errores. Esto normalmente solo se usa para completar nuevas listas donde no hay contenido anterior.

Nota: Este macro «roba» una referencia a *item* y, a diferencia de `PyList_SetItem()`, no descarta una referencia a ningún elemento que se está reemplazando; cualquier referencia en *list* en la posición *i* se filtrará.

int **PyList_Insert** (*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

Inserta el elemento *item* en la lista *list* delante del índice *index*. Retorna 0 si tiene éxito; retorna -1 y establece una excepción si no tiene éxito. Análogo a `list.insert(index, item)`.

int **PyList_Append** (*PyObject* *list, *PyObject* *item)

Agrega el objeto *item* al final de la lista *list*. Retorna 0 si tiene éxito; retorna -1 y establece una excepción si no tiene éxito. Análogo a `list.append(item)`.

*PyObject** **PyList_GetSlice** (*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high)

Return value: New reference. Retorna una lista de los objetos en *list* que contiene los objetos *between*, *low* y *high*. Retorna `NULL` y establece una excepción si no tiene éxito. Análogo a `list[low:high]`. La indexación desde el final de la lista no es compatible.

int **PyList_SetSlice** (*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high, *PyObject* *itemlist)

Establece el segmento de *list* entre *low* y *high* para el contenido de *itemlist*. Análogo a `list[low:high] = itemlist`. La lista *itemlist* puede ser `NULL`, lo que indica la asignación de una lista vacía (eliminación de segmentos). Retorna 0 en caso de éxito, -1 en caso de error. La indexación desde el final de la lista no es compatible.

int **PyList_Sort** (*PyObject* *list)

Ordena los elementos de *list* en su lugar. Retorna 0 en caso de éxito, -1 en caso de error. Esto es equivalente a `list.sort()`.

int **PyList_Reverse** (*PyObject* *list)

Invierte los elementos de la lista *list* en su lugar. Retorna 0 en caso de éxito, -1 en caso de error. Este es el equivalente de `list.reverse()`.

*PyObject** **PyList_AsTuple** (*PyObject* *list)

Return value: New reference. Retorna un nuevo objeto tupla que contiene el contenido de *list*; equivalente a `tuple(list)`.

8.4 Objetos contenedor

8.4.1 Objetos Diccionarios

PyDictObject

Este subtipo de *PyObject* representa un objeto diccionario de Python.

PyObject **PyDict_Type**

Esta instancia de *PyTypeObject* representa el tipo diccionario de Python. Este es el mismo objeto que `dict` en la capa de Python.

int PyDict_Check (*PyObject* *p)

Retorna verdadero si *p* es un objeto `dict` o una instancia de un subtipo del tipo `dict`. Esta función siempre finaliza con éxito.

int PyDict_CheckExact (*PyObject* *p)

Retorna verdadero si *p* es un objeto `dict`, pero no una instancia de un subtipo del tipo `dict`. Esta función siempre finaliza con éxito.

*PyObject** **PyDict_New** ()

Return value: New reference. Retorna un nuevo diccionario vacío, o NULL en caso de falla.

*PyObject** **PyDictProxy_New** (*PyObject* *mapping)

Return value: New reference. Retorna un objeto a `types.MappingProxyType` para una asignación que imponga un comportamiento de solo lectura. Esto normalmente se usa para crear una vista para evitar la modificación del diccionario para los tipos de clase no dinámicos.

void PyDict_Clear (*PyObject* *p)

Vacía un diccionario existente de todos los pares clave-valor (*key-value*).

int PyDict_Contains (*PyObject* *p, *PyObject* *key)

Determine si el diccionario *p* contiene *key*. Si un elemento en *p* coincide con la clave *key*, retorna 1; de lo contrario, retorna 0. En caso de error, retorna -1. Esto es equivalente a la expresión de Python `key in p`.

*PyObject** **PyDict_Copy** (*PyObject* *p)

Return value: New reference. Retorna un nuevo diccionario que contiene los mismos pares clave-valor (*key-value*) que *p*.

int PyDict_SetItem (*PyObject* *p, *PyObject* *key, *PyObject* *val)

Inserta *val* en el diccionario *p* con una clave *key*. *key* debe ser *hashable*; si no lo es, se lanzará `TypeError`. Retorna 0 en caso de éxito o -1 en caso de error. Esta función *no* roba una referencia a *val*.

int PyDict_SetItemString (*PyObject* *p, const char *key, *PyObject* *val)

Inserta *val* en el diccionario *p* usando *key* como clave. *key* debe ser un `const char*`. El objeto clave se crea usando `PyUnicode_FromString(key)`. Retorna 0 en caso de éxito o -1 en caso de error. Esta función *no* roba una referencia a *val*.

int PyDict_DelItem (*PyObject* *p, *PyObject* *key)

Elimina la entrada en el diccionario *p* con la clave *key*. *key* debe ser *hashable*; si no lo es, se lanza `TypeError`. Si *key* no está en el diccionario, se lanza `KeyError`. Retorna 0 en caso de éxito o -1 en caso de error.

int PyDict_DelItemString (*PyObject* *p, const char *key)

Elimina la entrada en el diccionario *p* que tiene una clave especificada por la cadena de caracteres *key*. Si *key* no está en el diccionario, se lanza `KeyError`. Retorna 0 en caso de éxito o -1 en caso de error.

*PyObject** **PyDict_GetItem** (*PyObject* *p, *PyObject* *key)

Return value: Borrowed reference. Retorna el objeto del diccionario *p* que tiene una clave *key*. Retorna NULL si la clave *key* no está presente, pero *sin* lanzar una excepción.

Tenga en cuenta que las excepciones que se producen al llamar `__hash__()` y `__eq__()` se suprimirán los métodos. Para obtener informes de errores, utilice `PyDict_GetItemWithError()` en su lugar.

*PyObject** **PyDict_GetItemWithError** (*PyObject* *p, *PyObject* *key)

Return value: Borrowed reference. Variante de `PyDict_GetItem()` que no suprime las excepciones. Retorna NULL **con** una excepción establecida si se produjo una excepción. Retorna NULL **sin** una excepción establecida si la clave no estaba presente.

*PyObject** **PyDict_GetItemString** (*PyObject* *p, const char *key)

Return value: Borrowed reference. Esto es lo mismo que `PyDict_GetItem()`, pero `key` se especifica como un `const char*`, en lugar de un `PyObject*`.

Tenga en cuenta que las excepciones que se producen al llamar a `__hash__()` y `__eq__()` y al crear un objeto de cadena de caracteres temporal se suprimirán. Para obtener informes de errores, utilice `PyDict_GetItemWithError()` en su lugar.

*PyObject** **PyDict_SetDefault** (*PyObject* *p, *PyObject* *key, *PyObject* *defaultobj)

Return value: Borrowed reference. Esto es lo mismo al nivel de `Python dict.setdefault()`. Si está presente, retorna el valor correspondiente a `key` del diccionario `p`. Si la clave no está en el `dict`, se inserta con el valor `defaultobj` y se retorna `defaultobj`. Esta función evalúa la función `hash` de `key` solo una vez, en lugar de evaluarla independientemente para la búsqueda y la inserción.

Nuevo en la versión 3.4.

*PyObject** **PyDict_Items** (*PyObject* *p)

Return value: New reference. Retorna un `PyListObject` que contiene todos los elementos del diccionario.

*PyObject** **PyDict_Keys** (*PyObject* *p)

Return value: New reference. Retorna un `PyListObject` que contiene todas las claves del diccionario.

*PyObject** **PyDict_Values** (*PyObject* *p)

Return value: New reference. Retorna un `PyListObject` que contiene todos los valores del diccionario `p`.

Py_ssize_t **PyDict_Size** (*PyObject* *p)

Retorna el número de elementos en el diccionario. Esto es equivalente a `len(p)` en un diccionario.

int **PyDict_Next** (*PyObject* *p, *Py_ssize_t* *ppos, *PyObject* **pkey, *PyObject* **pvalue)

Itera sobre todos los pares clave-valor en el diccionario `p`. El `Py_ssize_t` al que se refiere `ppos` debe inicializarse a 0 antes de la primera llamada a esta función para iniciar la iteración; la función retorna verdadero para cada par en el diccionario y falso una vez que todos los pares han sido reportados. Los parámetros `pkey` y `pvalue` deben apuntar a variables `PyObject*` que se completarán con cada clave y valor, respectivamente, o pueden ser NULL. Todas las referencias retornadas a través de ellos se toman prestadas. `ppos` no debe modificarse durante la iteración. Su valor representa compensaciones dentro de la estructura del diccionario interno y, dado que la estructura es escasa, las compensaciones no son consecutivas.

Por ejemplo:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* do something interesting with the values... */
    ...
}
```

El diccionario `p` no debe mutarse durante la iteración. Es seguro modificar los valores de las claves a medida que recorre el diccionario, pero solo mientras el conjunto de claves no cambie. Por ejemplo:

```
PyObject *key, *value;
Py_ssize_t pos = 0;
```

(continué en la próxima página)

(proviene de la página anterior)

```

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    long i = PyLong_AsLong(value);
    if (i == -1 && PyErr_Occurred()) {
        return -1;
    }
    PyObject *o = PyLong_FromLong(i + 1);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
        return -1;
    }
    Py_DECREF(o);
}
}

```

int **PyDict_Merge** (*PyObject* *a, *PyObject* *b, int override)

Itera sobre el objeto de mapeo *b* agregando pares clave-valor al diccionario *a*. *b* puede ser un diccionario o cualquier objeto que soporte *PyMapping_Keys()* y *PyObject_GetItem()*. Si *override* es verdadero, los pares existentes en *a* se reemplazarán si se encuentra una clave coincidente en *b*, de lo contrario, los pares solo se agregarán si no hay una clave coincidente en *a*. Retorna 0 en caso de éxito o -1 si se lanza una excepción.

int **PyDict_Update** (*PyObject* *a, *PyObject* *b)

Esto es lo mismo que *PyDict_Merge(a, b, 1)* en C, y es similar a *a.update(b)* en Python excepto que *PyDict_Update()* no vuelve a la iteración sobre una secuencia de pares de valores clave si el segundo argumento no tiene el atributo «claves». Retorna 0 en caso de éxito o -1 si se produjo una excepción.

int **PyDict_MergeFromSeq2** (*PyObject* *a, *PyObject* *seq2, int override)

Actualiza o combina en el diccionario *a*, desde los pares clave-valor en *seq2*. *seq2* debe ser un objeto iterable que produzca objetos iterables de longitud 2, vistos como pares clave-valor. En el caso de claves duplicadas, el último gana si *override* es verdadero, de lo contrario, el primero gana. Retorna 0 en caso de éxito o -1 si se produjo una excepción. El equivalente en Python (excepto el valor de retorno)

```

def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value

```

8.4.2 Objetos Conjunto

This section details the public API for *set* and *frozenset* objects. Any functionality not listed below is best accessed using either the abstract object protocol (including *PyObject_CallMethod()*, *PyObject_RichCompareBool()*, *PyObject_Hash()*, *PyObject_Repr()*, *PyObject_IsTrue()*, *PyObject_Print()*, and *PyObject_GetIter()*) or the abstract number protocol (including *PyNumber_And()*, *PyNumber_Subtract()*, *PyNumber_Or()*, *PyNumber_Xor()*, *PyNumber_InPlaceAnd()*, *PyNumber_InPlaceSubtract()*, *PyNumber_InPlaceOr()*, and *PyNumber_InPlaceXor()*).

PySetObject

This subtype of *PyObject* is used to hold the internal data for both *set* and *frozenset* objects. It is like a *PyDictObject* in that it is a fixed size for small sets (much like tuple storage) and will point to a separate, variable sized block of memory for medium and large sized sets (much like list storage). None of the fields of this structure should be considered public and all are subject to change. All access should be done through the documented API rather than by manipulating the values in the structure.

***PyObject* PySet_Type**

Esta es una instancia de *PyObject* que representa el tipo Python `set`.

***PyObject* PyFrozenSet_Type**

Esta es una instancia de *PyObject* que representa el tipo Python `frozenset`.

Los siguientes macros de comprobación de tipos funcionan en punteros a cualquier objeto de Python. Del mismo modo, las funciones del constructor funcionan con cualquier objeto Python iterable.

int PySet_Check (*PyObject* *p)

Retorna verdadero si *p* es un objeto `set` o una instancia de un subtipo. Esta función siempre finaliza con éxito.

int PyFrozenSet_Check (*PyObject* *p)

Retorna verdadero si *p* es un objeto `frozenset` o una instancia de un subtipo. Esta función siempre finaliza con éxito.

int PyAnySet_Check (*PyObject* *p)

Retorna verdadero si *p* es un objeto `set`, un objeto `frozenset`, o una instancia de un subtipo. Esta función siempre finaliza con éxito.

int PyAnySet_CheckExact (*PyObject* *p)

Retorna verdadero si *p* es un objeto `set` o un objeto `frozenset` pero no una instancia de un subtipo. Esta función siempre finaliza con éxito.

int PyFrozenSet_CheckExact (*PyObject* *p)

Retorna verdadero si *p* es un objeto `frozenset` pero no una instancia de un subtipo. Esta función siempre finaliza con éxito.

***PyObject** PySet_New (*PyObject* *iterable)**

Return value: New reference. Retorna un nuevo `set` que contiene objetos retornados por *iterable*. El *iterable* puede ser `NULL` para crear un nuevo conjunto vacío. Retorna el nuevo conjunto en caso de éxito o `NULL` en caso de error. Lanza `TypeError` si *iterable* no es realmente iterable. El constructor también es útil para copiar un conjunto (`c=set(s)`).

***PyObject** PyFrozenSet_New (*PyObject* *iterable)**

Return value: New reference. Retorna un nuevo `frozenset` que contiene objetos retornados por *iterable*. El *iterable* puede ser `NULL` para crear un nuevo conjunto congelado vacío. Retorna el nuevo conjunto en caso de éxito o `NULL` en caso de error. Lanza `TypeError` si *iterable* no es realmente iterable.

Las siguientes funciones y macros están disponibles para instancias de `set` o `frozenset` o instancias de sus subtipos.

***Py_ssize_t* PySet_Size (*PyObject* *anyset)**

Retorna la longitud de un objeto `set` o `frozenset`. Equivalente a `len(anyset)`. Lanza un `PyExc_SystemError` si *anyset* no es `set`, `frozenset`, o una instancia de un subtipo.

***Py_ssize_t* PySet_GET_SIZE (*PyObject* *anyset)**

Forma macro de `PySet_Size()` sin comprobación de errores.

int PySet_Contains (*PyObject* *anyset, *PyObject* *key)

Retorna 1 si se encuentra, 0 si no se encuentra y -1 si se encuentra un error. A diferencia del método Python `__contains__()`, esta función no convierte automáticamente conjuntos no compartibles en congelados temporales. Lanza un `TypeError` si la *key* no se puede compartir. Lanza `PyExc_SystemError` si *anyset* no es un `set`, `frozenset`, o una instancia de un subtipo.

int PySet_Add (*PyObject* *set, *PyObject* *key)

Add *key* to a `set` instance. Also works with `frozenset` instances (like `PyTuple_SetItem()` it can be used to fill in the values of brand new `frozensets` before they are exposed to other code). Return 0 on success or -1 on failure. Raise a `TypeError` if the *key* is unhashable. Raise a `MemoryError` if there is no room to grow. Raise a `SystemError` if *set* is not an instance of `set` or its subtype.

Las siguientes funciones están disponibles para instancias de `set` o sus subtipos, pero no para instancias de `frozenset` o sus subtipos.

int PySet_Discard (*PyObject* *set, *PyObject* *key)

Retorna 1 si se encuentra y se elimina, 0 si no se encuentra (no se realiza ninguna acción) y -1 si se encuentra un error. No lanza `KeyError` por faltar claves. Lanza un `TypeError` si la *key* no se puede compartir. A diferencia del método Python `discard()`, esta función no convierte automáticamente conjuntos no compartibles en congelados temporales. Lanza `PyExc_SystemError` si *set* no es una instancia de `set` o su subtipo.

*PyObject** **PySet_Pop** (*PyObject* *set)

Return value: New reference. Retorna una nueva referencia a un objeto arbitrario en el *set* y elimina el objeto del *set*. Retorna `NULL` en caso de falla. Lanza `KeyError` si el conjunto está vacío. Lanza a `SystemError` si *set* no es una instancia de `set` o su subtipo.

int PySet_Clear (*PyObject* *set)

Vacía un conjunto existente de todos los elementos.

8.5 Objetos de función

8.5.1 Objetos función

Hay algunas funciones específicas para las funciones de Python.

PyFunctionObject

La estructura C utilizada para las funciones.

PyObject **PyFunction_Type**

Esta es una instancia de *PyObject* y representa el tipo función de Python. Está expuesto a los programadores de Python como `types.FunctionType`.

int PyFunction_Check (*PyObject* *o)

Retorna verdadero si *o* es un objeto función (tiene tipo *PyFunction_Type*). El parámetro no debe ser `NULL`. Esta función siempre finaliza con éxito.

*PyObject** **PyFunction_New** (*PyObject* *code, *PyObject* *globals)

Return value: New reference. Retorna un nuevo objeto función asociado con el objeto código *code*. *globals* debe ser un diccionario con las variables globales accesibles para la función.

El docstring y el nombre de la función se obtiene del objeto código. `__module__` se obtiene de *globals* *. El argumento **defaults*, *annotations* y *closure* se establecen en `NULL`. `__qualname__` se establece en el mismo valor que el nombre de la función.

*PyObject** **PyFunction_NewWithQualName** (*PyObject* *code, *PyObject* *globals, *PyObject* *qualname)

Return value: New reference. Como *PyFunction_New()*, pero también permite configurar el atributo `__qualname__` del objeto función. *qualname* debe ser un objeto unicode o `NULL`; si es `NULL`, el atributo `__qualname__` se establece en el mismo valor que su atributo `__name__`.

Nuevo en la versión 3.3.

*PyObject** **PyFunction_GetCode** (*PyObject* *op)

Return value: Borrowed reference. Retorna el objeto código asociado con el objeto función *op*.

*PyObject** **PyFunction_GetGlobals** (*PyObject* *op)

Return value: Borrowed reference. Retorna el diccionario global asociado con el objeto función *op*.

*PyObject** **PyFunction_GetModule** (*PyObject* *op)

Return value: Borrowed reference. Retorna el atributo `__module__` del objeto función *op*. Normalmente es una

cadena de caracteres que contiene el nombre del módulo, pero se puede establecer en cualquier otro objeto mediante el código Python.

*PyObject** **PyFunction_GetDefaults** (*PyObject* **op*)

Return value: Borrowed reference. Retorna los valores predeterminados del argumento del objeto función *op*. Esto puede ser una tupla de argumentos o NULL.

int **PyFunction_SetDefaults** (*PyObject* **op*, *PyObject* **defaults*)

Establece los valores predeterminados del argumento para el objeto función *op*. *defaults* deben ser `Py_None` o una tupla.

Lanza `SystemError` y retorna `-1` en caso de error.

*PyObject** **PyFunction_GetClosure** (*PyObject* **op*)

Return value: Borrowed reference. Retorna el cierre asociado con el objeto función *op*. Esto puede ser NULL o una tupla de objetos celda.

int **PyFunction_SetClosure** (*PyObject* **op*, *PyObject* **closure*)

Establece el cierre asociado con el objeto función *op*. *cierre* debe ser `Py_None` o una tupla de objetos celda.

Lanza `SystemError` y retorna `-1` en caso de error.

*PyObject** **PyFunction_GetAnnotations** (*PyObject* **op*)

Return value: Borrowed reference. Retorna las anotaciones del objeto función *op*. Este puede ser un diccionario mutable o NULL.

int **PyFunction_SetAnnotations** (*PyObject* **op*, *PyObject* **annotations*)

Establece las anotaciones para el objeto función *op*. *annotations* debe ser un diccionario o `Py_None`.

Lanza `SystemError` y retorna `-1` en caso de error.

8.5.2 Objetos de método de instancia

Un método de instancia es un contenedor para una `PyCFunction` y la nueva forma de vincular una `PyCFunction` a un objeto de clase. Reemplaza la llamada anterior `PyMethod_New (func, NULL, class)`.

PyTypeObject **PyInstanceMethod_Type**

Esta instancia de `PyTypeObject` representa el tipo de método de instancia de Python. No está expuesto a los programas de Python.

int **PyInstanceMethod_Check** (*PyObject* **o*)

Retorna verdadero si *o* es un objeto de método de instancia (tiene tipo `PyInstanceMethod_Type`). El parámetro no debe ser NULL. Esta función siempre finaliza con éxito.

*PyObject** **PyInstanceMethod_New** (*PyObject* **func*)

Return value: New reference. Return a new instance method object, with *func* being any callable object. *func* is the function that will be called when the instance method is called.

*PyObject** **PyInstanceMethod_Function** (*PyObject* **im*)

Return value: Borrowed reference. Retorna el objeto de función asociado con el método de instancia *im*.

*PyObject** **PyInstanceMethod_GET_FUNCTION** (*PyObject* **im*)

Return value: Borrowed reference. Versión macro de `PyInstanceMethod_Function()` que evita la comprobación de errores.

8.5.3 Objetos método

Los métodos son objetos de función enlazados. Los métodos siempre están vinculados a una instancia de una clase definida por el usuario. Los métodos no vinculados (métodos vinculados a un objeto de clase) ya no están disponibles.

PyObject **PyMethod_Type**

Esta instancia de *PyObject* representa el tipo de método Python. Esto está expuesto a los programas de Python como `types.MethodType`.

int **PyMethod_Check** (*PyObject* *o)

Retorna verdadero si *o* es un objeto de método (tiene tipo *PyMethod_Type*). El parámetro no debe ser NULL. Esta función siempre finaliza con éxito.

*PyObject** **PyMethod_New** (*PyObject* *func, *PyObject* *self)

Return value: New reference. Retorna un nuevo objeto de método, con *func* como cualquier objeto invocable y *self* la instancia en la que se debe vincular el método. *func* es la función que se llamará cuando se llame al método. *self* no debe ser NULL.

*PyObject** **PyMethod_Function** (*PyObject* *meth)

Return value: Borrowed reference. Retorna el objeto de función asociado con el método *meth*.

*PyObject** **PyMethod_GET_FUNCTION** (*PyObject* *meth)

Return value: Borrowed reference. Versión macro de *PyMethod_Function()* que evita la comprobación de errores.

*PyObject** **PyMethod_Self** (*PyObject* *meth)

Return value: Borrowed reference. Retorna la instancia asociada con el método *meth*.

*PyObject** **PyMethod_GET_SELF** (*PyObject* *meth)

Return value: Borrowed reference. Versión macro de *PyMethod_Self()* que evita la comprobación de errores.

8.5.4 Objetos Celda

Los objetos celda (*cell*) se utilizan para implementar variables a las que hacen referencia varios ámbitos. Para cada variable, se crea un objeto de celda para almacenar el valor; Las variables locales de cada marco de pila que hace referencia al valor contienen una referencia a las celdas de ámbitos externos que también usan esa variable. Cuando se accede al valor, se utiliza el valor contenido en la celda en lugar del objeto de la celda en sí. Esta desreferenciación del objeto de celda requiere soporte del código de bytes generado; estos no se eliminan automáticamente cuando se accede a ellos. No es probable que los objetos celda sean útiles en otros lugares.

PyCellObject

La estructura C utilizada para objetos celda.

PyObject **PyCell_Type**

El objeto tipo correspondiente a los objetos celda.

int **PyCell_Check** (ob)

Retorna verdadero si *ob* es un objeto de celda; *ob* no debe ser NULL. Esta función siempre finaliza con éxito.

*PyObject** **PyCell_New** (*PyObject* *ob)

Return value: New reference. Crea y retorna un nuevo objeto de celda que contiene el valor *ob*. El parámetro puede ser NULL.

*PyObject** **PyCell_Get** (*PyObject* *cell)

Return value: New reference. Retorna el contenido de la celda *cell*.

*PyObject** **PyCell_GET** (*PyObject* *cell)

Return value: Borrowed reference. Retorna el contenido de la celda *cell*, pero sin verificar que *cell* no sea NULL y que sea un objeto de celda.

int **PyCell_Set** (*PyObject* *cell, *PyObject* *value)

Establece el contenido del objeto de celda *cell* con el valor *value*. Esto libera la referencia a cualquier contenido actual de la celda. *value* puede ser NULL. *cell* no debe ser NULL; Si no es un objeto de celda, se retornará -1. En caso de éxito, se retornará 0.

void **PyCell_SET** (*PyObject* *cell, *PyObject* *value)

Establece el valor del objeto de celda *cell* en el valor *value*. No se ajustan los recuentos de referencia y no se realizan verificaciones de seguridad; *cell* no debe ser NULL y debe ser un objeto de celda.

8.5.5 Objetos Código

Los objetos código son un detalle de bajo nivel de la implementación de CPython. Cada uno representa un fragmento de código ejecutable que aún no se ha vinculado a una función.

PyCodeObject

La estructura en C de los objetos utilizados para describir objetos código. Los campos de este tipo están sujetos a cambios en cualquier momento.

PyTypeObject **PyCode_Type**

Esta es una instancia de *PyTypeObject* que representa el tipo Python code.

int **PyCode_Check** (*PyObject* *co)

Retorna verdadero si *co* es un objeto code. Esta función siempre finaliza con éxito.

int **PyCode_GetNumFree** (*PyCodeObject* *co)

Retorna el número de variables libres en *co*.

*PyCodeObject** **PyCode_New** (int *argcount*, int *kwnonlyargcount*, int *nlocals*, int *stacksize*, int *flags*, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, int *firstlineno*, *PyObject* *notab)

Return value: New reference. Retorna un nuevo objeto de código. Si necesita un objeto de código ficticio para crear un marco (*frame*), use *PyCode_NewEmpty*() en su lugar. Llamando *PyCode_New*() directamente puede enlazarlo a una versión precisa de Python ya que la definición del código de bytes cambia a menudo.

*PyCodeObject** **PyCode_NewWithPosOnlyArgs** (int *argcount*, int *posonlyargcount*, int *kwnonlyargcount*, int *nlocals*, int *stacksize*, int *flags*, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, int *firstlineno*, *PyObject* *notab)

Return value: New reference. Similar a *PyCode_New*(), pero con un «*posonlyargcount*» adicional para argumentos solo posicionales.

Nuevo en la versión 3.8.

*PyCodeObject** **PyCode_NewEmpty** (const char *filename, const char *funcname, int *firstlineno*)

Return value: New reference. Retorna un nuevo objeto de código vacío con el nombre de archivo especificado, el nombre de la función y el número de la primera línea. Es ilegal utilizar *exec*() o *eval*() en el objeto de código resultante.

8.6 Otros objetos

8.6.1 Objetos archivo

Estas API son una emulación mínima de la API Python 2 en C para objetos de archivo incorporados, que solía depender del soporte de E/S (I/O) almacenadas en la memoria intermedia (FILE*) de la biblioteca estándar C. En Python 3, los archivos y las secuencias utilizan el nuevo módulo `io`, que define varias capas sobre las E/S sin búfer de bajo nivel del sistema operativo. Las funciones que se describen a continuación son envoltorios en C de conveniencia sobre estas nuevas API y están destinadas principalmente a la notificación de errores internos en el intérprete; se recomienda que el código de terceros acceda a las API `io`.

*PyObject** **PyFile_FromFd** (int *fd*, const char **name*, const char **mode*, int *buffering*, const char **encoding*, const char **errors*, const char **newline*, int *closefd*)

Return value: New reference. Crea un objeto archivo Python a partir del descriptor de archivo de un archivo ya abierto *fd*. Los argumentos *name*, *encoding*, *errors* y *newline* pueden ser NULL para usar los valores predeterminados; *buffering* puede ser -1 para usar el valor predeterminado. *name* se ignora y se mantiene por compatibilidad con versiones anteriores. Retorna NULL en caso de error. Para obtener una descripción más completa de los argumentos, consulte la documentación de la función `io.open()`.

Advertencia: Dado que las transmisiones (*streams*) de Python tienen su propia capa de almacenamiento en búfer, combinarlas con descriptores de archivos a nivel del sistema operativo puede producir varios problemas (como un pedido inesperado de datos).

Distinto en la versión 3.2: Ignora el atributo *name*.

int **PyObject_AsFileDescriptor** (*PyObject *p*)

Retorna el descriptor de archivo asociado con *p* como int. Si el objeto es un entero, se retorna su valor. Si no, se llama al método `fileno()` del objeto si existe; el método debe retornar un número entero, que se retorna como el valor del descriptor de archivo. Establece una excepción y retorna -1 en caso de error.

*PyObject** **PyFile_GetLine** (*PyObject *p*, int *n*)

Return value: New reference. Equivalente a `p.readline([n])`, esta función lee una línea del objeto *p*. *p* puede ser un objeto archivo o cualquier objeto con un método `readline()`. Si *n* es 0, se lee exactamente una línea, independientemente de la longitud de la línea. Si *n* es mayor que 0, no se leerán más de *n* bytes del archivo; se puede retornar una línea parcial. En ambos casos, se retorna una cadena de caracteres vacía si se llega al final del archivo de inmediato. Si *n* es menor que 0, sin embargo, se lee una línea independientemente de la longitud, pero `EOFError` se lanza si se llega al final del archivo de inmediato.

int **PyFile_SetOpenCodeHook** (Py_OpenCodeHookFunction *handler*)

Sobrescribe el comportamiento normal de `io.open_code()` para pasar su parámetro a través del controlador proporcionado.

El controlador es una función de tipo `PyObject * (*)(PyObject *path, void *userData)`, donde se garantiza que *path* sea `PyUnicodeObject`.

El puntero *userData* se pasa a la función de enlace. Dado que las funciones de enlace pueden llamarse desde diferentes tiempos de ejecución, este puntero no debe referirse directamente al estado de Python.

Como este *hook* se usa intencionalmente durante la importación, evite importar nuevos módulos durante su ejecución a menos que se sepa que están congelados o disponibles en `sys.modules`.

Una vez que se ha establecido un *hook*, no se puede quitar ni reemplazar, y luego llamadas a `PyFile_SetOpenCodeHook()` fallarán. En caso de error, la función retorna -1 y establece una excepción si el intérprete se ha inicializado.

Es seguro llamar a esta función antes de `Py_Initialize()`.

Genera un evento de auditoría `setopencodehook` sin argumentos.

Nuevo en la versión 3.8.

int **PyFile_WriteObject** (*PyObject* *obj, *PyObject* *p, int flags)

Escribe el objeto *obj* en el objeto archivo *p*. El único indicador admitido para *flags* es `Py_PRINT_RAW`; si se proporciona, se escribe el `str()` del objeto en lugar de `repr()`. Retorna 0 en caso de éxito o -1 en caso de error; se establecerá la excepción apropiada.

int **PyFile_WriteString** (const char *s, *PyObject* *p)

Escribe la cadena *s* en el objeto archivo *p*. Retorna 0 en caso de éxito o -1 en caso de error; se establecerá la excepción apropiada.

8.6.2 Objetos Modulo

PyObject **PyModule_Type**

Esta instancia de *PyObject* representa el tipo de módulo Python. Esto está expuesto a los programas de Python como `types.ModuleType`.

int **PyModule_Check** (*PyObject* *p)

Retorna verdadero si *p* es un objeto de módulo o un subtipo de un objeto de módulo. Esta función siempre finaliza con éxito.

int **PyModule_CheckExact** (*PyObject* *p)

Retorna verdadero si *p* es un objeto módulo, pero no un subtipo de *PyModule_Type*. Esta función siempre finaliza con éxito.

*PyObject** **PyModule_NewObject** (*PyObject* *name)

Return value: *New reference*. Retorna un nuevo objeto módulo con el atributo `__name__` establecido en *name*. Los atributos del módulo `__name__`, `__doc__`, `__package__`, y `__loader__` se completan (todos menos `__name__` están configurados en `None`); quien llama es responsable de proporcionar un atributo `__file__`.

Nuevo en la versión 3.3.

Distinto en la versión 3.4: `__package__` y `__loader__` están configurados en `None`.

*PyObject** **PyModule_New** (const char *name)

Return value: *New reference*. Similar a *PyModule_NewObject()*, pero el nombre es una cadena de caracteres codificada UTF-8 en lugar de un objeto Unicode.

*PyObject** **PyModule_GetDict** (*PyObject* *module)

Return value: *Borrowed reference*. Retorna el objeto del diccionario que implementa el espacio de nombres de *module*; este objeto es el mismo que el atributo `__dict__` del objeto módulo. Si *module* no es un objeto módulo (o un subtipo de un objeto de módulo), `SystemError` se genera y se retorna `NULL`.

Se recomienda que las extensiones utilicen otras funciones *PyModule_**() y *PyObject_**() en lugar de manipular directamente el módulo `__dict__`.

*PyObject** **PyModule_GetNameObject** (*PyObject* *module)

Return value: *New reference*. Retorna el valor `__name__` del *module*. Si el módulo no proporciona uno, o si no es una cadena de caracteres, `SystemError` se lanza y se retorna `NULL`.

Nuevo en la versión 3.3.

const char* **PyModule_GetName** (*PyObject* *module)

Similar a *PyModule_GetNameObject()* pero retorna el nombre codificado a 'utf-8'.

void* **PyModule_GetState** (*PyObject* *module)

Retorna el «estado» del módulo, es decir, un puntero al bloque de memoria asignado en el momento de la creación del módulo, o `NULL`. Ver *PyModuleDef.m_size*.

*PyModuleDef** **PyModule_GetDef** (*PyObject* *module)

Retorna un puntero a la estructura *PyModuleDef* a partir de la cual se creó el módulo, o NULL si el módulo no se creó a partir de una definición.

*PyObject** **PyModule_GetFilenameObject** (*PyObject* *module)

Return value: New reference. Retorna el nombre del archivo desde el cual *module* se cargó utilizando el atributo `__file__` del *module*. Si esto no está definido, o si no es una cadena de caracteres unicode, lanza `SystemError` y retornar NULL; de lo contrario, retorna una referencia a un objeto Unicode.

Nuevo en la versión 3.2.

const char* **PyModule_GetFilename** (*PyObject* *module)

Similar a *PyModule_GetFilenameObject* () pero retorna el nombre de archivo codificado a “utf-8”.

Obsoleto desde la versión 3.2: *PyModule_GetFilename* () lanza `UnicodeEncodeError` en nombres de archivo no codificables, use *PyModule_GetFilenameObject* () en su lugar.

Inicializando módulos en C

Los objetos módulos generalmente se crean a partir de módulos de extensión (bibliotecas compartidas que exportan una función de inicialización) o módulos compilados (donde la función de inicialización se agrega usando *PyImport_AppendInittab*()). Consulte *building* o extendiendo con incrustación para más detalles.

La función de inicialización puede pasar una instancia de definición de módulo a *PyModule_Create* (), y retornar el objeto módulo resultante, o solicitar una «inicialización de múltiples fases» retornando la estructura de definición.

PyModuleDef

La estructura de definición de módulo, que contiene toda la información necesaria para crear un objeto módulo. Por lo general, solo hay una variable estáticamente inicializada de este tipo para cada módulo.

PyModuleDef_Base **m_base**

Siempre inicialice este miembro a *PyModuleDef_HEAD_INIT*.

const char ***m_name**

Nombre para el nuevo módulo.

const char ***m_doc**

Docstring para el módulo; por lo general, se usa una variable docstring creada con *PyDoc_STRVAR*.

Py_ssize_t **m_size**

El estado del módulo se puede mantener en un área de memoria por módulo que se puede recuperar con *PyModule_GetState* (), en lugar de en globales estáticos. Esto hace que los módulos sean seguros para su uso en múltiples sub-interpretadores.

Esta área de memoria se asigna en base a *m_size* en la creación del módulo, y se libera cuando el objeto del módulo se desasigna, después de que se haya llamado a la función *m_free*, si está presente.

Establecer *m_size* en -1 significa que el módulo no admite sub-interpretadores, porque tiene un estado global.

Establecerlo en un valor no negativo significa que el módulo se puede reinicializar y especifica la cantidad adicional de memoria que requiere para su estado. Se requiere *m_size* no negativo para la inicialización de múltiples fases.

Ver **PEP 3121** para más detalles.

*PyMethodDef** **m_methods**

Un puntero a una tabla de funciones de nivel de módulo, descrito por valores *PyMethodDef*. Puede ser NULL si no hay funciones presentes.

***PyModuleDef_Slot** m_slots**

Un conjunto de definiciones de ranura para la inicialización de múltiples fases, terminadas por una entrada {0, NULL}. Cuando se utiliza la inicialización monofásica, *m_slots* debe ser NULL.

Distinto en la versión 3.5: Antes de la versión 3.5, este miembro siempre estaba configurado en NULL y se definía como:

inquiry **m_reload**

***traverseproc* m_traverse**

Una función transversal para llamar durante el recorrido GC del objeto del módulo, o NULL si no es necesario.

Esta función no se llama si se solicitó el estado del módulo pero aún no se asignó. Este es el caso inmediatamente después de que se crea el módulo y antes de que se ejecute (la función *Py_mod_exec*). Más precisamente, esta función no se llama si *m_size* es mayor que 0 y el estado del módulo (como lo retorna *PyModule_GetState()*) es NULL.

Distinto en la versión 3.9: Ya no se llama antes de que se asigne el estado del módulo.

***inquiry* m_clear**

Una función clara para llamar durante la limpieza GC del objeto del módulo, o NULL si no es necesario.

Esta función no se llama si se solicitó el estado del módulo pero aún no se asignó. Este es el caso inmediatamente después de que se crea el módulo y antes de que se ejecute (la función *Py_mod_exec*). Más precisamente, esta función no se llama si *m_size* es mayor que 0 y el estado del módulo (como lo retorna *PyModule_GetState()*) es NULL.

Like *PyTypeObject.tp_clear*, this function is not *always* called before a module is deallocated. For example, when reference counting is enough to determine that an object is no longer used, the cyclic garbage collector is not involved and *m_free* is called directly.

Distinto en la versión 3.9: Ya no se llama antes de que se asigne el estado del módulo.

***freefunc* m_free**

Una función para llamar durante la desasignación del objeto del módulo, o NULL si no es necesario.

Esta función no se llama si se solicitó el estado del módulo pero aún no se asignó. Este es el caso inmediatamente después de que se crea el módulo y antes de que se ejecute (la función *Py_mod_exec*). Más precisamente, esta función no se llama si *m_size* es mayor que 0 y el estado del módulo (como lo retorna *PyModule_GetState()*) es NULL.

Distinto en la versión 3.9: Ya no se llama antes de que se asigne el estado del módulo.

Inicialización monofásica

La función de inicialización del módulo puede crear y retornar el objeto módulo directamente. Esto se conoce como «inicialización monofásica» y utiliza una de las siguientes funciones de creación de dos módulos:

***PyObject** PyModule_Create(*PyModuleDef* *def)**

Return value: *New reference*. Crea un nuevo objeto módulo, dada la definición en *def*. Esto se comporta como *PyModule_Create2()* con *module_api_version* establecido en *PYTHON_API_VERSION*.

***PyObject** PyModule_Create2(*PyModuleDef* *def, int module_api_version)**

Return value: *New reference*. Crea un nuevo objeto de módulo, dada la definición en *def*, asumiendo la versión de API *module_api_version*. Si esa versión no coincide con la versión del intérprete en ejecución, se emite un *RuntimeWarning*.

Nota: La mayoría de los usos de esta función deberían usar `PyModule_Create()` en su lugar; solo use esto si está seguro de que lo necesita.

Antes de que se retorne desde la función de inicialización, el objeto del módulo resultante normalmente se llena utilizando funciones como `PyModule_AddObject()`.

Inicialización multifase

Una forma alternativa de especificar extensiones es solicitar una «inicialización de múltiples fases». Los módulos de extensión creados de esta manera se comportan más como los módulos de Python: la inicialización se divide entre la fase de creación (*creation phase*), cuando se crea el objeto módulo, y la fase de ejecución (*execution phase*), cuando se llena. La distinción es similar a los métodos de clases `__new__()` y `__init__()`.

A diferencia de los módulos creados con la inicialización monofásica, estos módulos no son singletons: si se elimina la entrada `sys.modules` y el módulo se vuelve a importar, se crea un nuevo objeto módulo y el módulo anterior está sujeto a la recolección normal de basura – Al igual que con los módulos de Python. Por defecto, los módulos múltiples creados a partir de la misma definición deberían ser independientes: los cambios en uno no deberían afectar a los demás. Esto significa que todo el estado debe ser específico para el objeto del módulo (usando, por ejemplo, usando `PyModule_GetState()`), o su contenido (como el módulo `__dict__` o clases individuales creadas con `PyType_FromSpec()`).

Se espera que todos los módulos creados mediante la inicialización de múltiples fases admitan *sub-interpretores*. Asegurándose de que varios módulos sean independientes suele ser suficiente para lograr esto.

Para solicitar la inicialización de múltiples fases, la función de inicialización (`PyInit_modulename`) retorna una instancia de `PyModuleDef` con un `m_slots` no vacío. Antes de que se retorne, la instancia `PyModuleDef` debe inicializarse con la siguiente función:

*PyObject** **PyModuleDef_Init** (*PyModuleDef* *def)

Return value: Borrowed reference. Asegura que la definición de un módulo sea un objeto Python correctamente inicializado que informe correctamente su tipo y conteo de referencias.

Retorna *def* convertido a `PyObject*` o `NULL` si se produjo un error.

Nuevo en la versión 3.5.

El miembro `m_slots` de la definición del módulo debe apuntar a un arreglo de estructuras `PyModuleDef_Slot`:

PyModuleDef_Slot

`int slot`

Una ranura ID, elegida entre los valores disponibles que se explican a continuación.

`void*` **value**

Valor de la ranura, cuyo significado depende de la ID de la ranura.

Nuevo en la versión 3.5.

El arreglo `m_slots` debe estar terminada por una ranura con id 0.

Los tipos de ranura disponibles son:

Py_mod_create

Especifica una función que se llama para crear el objeto del módulo en sí. El puntero *value* de este espacio debe apuntar a una función de la firma:

*PyObject** **create_module** (*PyObject* *spec, *PyModuleDef* *def)

La función recibe una instancia de `ModuleSpec`, como se define en [PEP 451](#), y la definición del módulo. Debería retornar un nuevo objeto de módulo, o establecer un error y retornar `NULL`.

Esta función debe mantenerse mínima. En particular, no debería llamar a código arbitrario de Python, ya que intentar importar el mismo módulo nuevamente puede dar como resultado un bucle infinito.

Múltiples ranuras `Py_mod_create` no pueden especificarse en una definición de módulo.

Si no se especifica `Py_mod_create`, la maquinaria de importación creará un objeto de módulo normal usando `PyModule_New()`. El nombre se toma de `spec`, no de la definición, para permitir que los módulos de extensión se ajusten dinámicamente a su lugar en la jerarquía de módulos y se importen bajo diferentes nombres a través de enlaces simbólicos, todo mientras se comparte una definición de módulo único.

No es necesario que el objeto retornado sea una instancia de `PyModule_Type`. Se puede usar cualquier tipo, siempre que admita la configuración y la obtención de atributos relacionados con la importación. Sin embargo, solo se pueden retornar instancias `PyModule_Type` si el `PyModuleDef` no tiene `NULL m_traverse`, `m_clear`, `m_free`; `m_size` distinto de cero; o ranuras que no sean `Py_mod_create`.

Py_mod_exec

Especifica una función que se llama para ejecutar (*execute*) el módulo. Esto es equivalente a ejecutar el código de un módulo Python: por lo general, esta función agrega clases y constantes al módulo. La firma de la función es:

```
int exec_module (PyObject* module)
```

Si se especifican varias ranuras `Py_mod_exec`, se procesan en el orden en que aparecen en el arreglo `m_slots`.

Ver [PEP 489](#) para más detalles sobre la inicialización de múltiples fases.

Funciones de creación de módulos de bajo nivel

Las siguientes funciones se invocan en segundo plano cuando se utiliza la inicialización de múltiples fases. Se pueden usar directamente, por ejemplo, al crear objetos de módulo de forma dinámica. Tenga en cuenta que tanto `PyModule_FromDefAndSpec` como `PyModule_ExecDef` deben llamarse para inicializar completamente un módulo.

*PyObject** **PyModule_FromDefAndSpec** (*PyModuleDef* *def, *PyObject* *spec)

Return value: New reference. Cree un nuevo objeto módulo, dada la definición en *module* y `ModuleSpec spec`. Esto se comporta como `PyModule_FromDefAndSpec2()` con `module_api_version` establecido en `PYTHON_API_VERSION`.

Nuevo en la versión 3.5.

*PyObject** **PyModule_FromDefAndSpec2** (*PyModuleDef* *def, *PyObject* *spec, int module_api_version)

Return value: New reference. Cree un nuevo objeto módulo, dada la definición en *module* y `ModuleSpec spec`, asumiendo la versión de API `module_api_version`. Si esa versión no coincide con la versión del intérprete en ejecución, se emite un `RuntimeWarning`.

Nota: La mayoría de los usos de esta función deberían usar `PyModule_FromDefAndSpec()` en su lugar; solo use esto si está seguro de que lo necesita.

Nuevo en la versión 3.5.

int **PyModule_ExecDef** (*PyObject* *module, *PyModuleDef* *def)

Procesa cualquier ranura de ejecución (`Py_mod_exec`) dado en *def*.

Nuevo en la versión 3.5.

int **PyModule_SetDocString** (*PyObject* *module, const char *docstring)

Establece la cadena de caracteres de documentación para *module* en *docstring*. Esta función se llama automáticamente cuando se crea un módulo desde `PyModuleDef`, usando `PyModule_Create` o `PyModule_FromDefAndSpec`.

Nuevo en la versión 3.5.

int **PyModule_AddFunctions** (*PyObject* *module, *PyMethodDef* *functions)

Agrega las funciones del arreglo *functions* terminadas en `NULL` a *module*. Consulte la documentación de *PyMethodDef* para obtener detalles sobre entradas individuales (debido a la falta de un espacio de nombres de módulo compartido, las «funciones» de nivel de módulo implementadas en C generalmente reciben el módulo como su primer parámetro, haciéndolos similares a la instancia métodos en clases de Python). Esta función se llama automáticamente cuando se crea un módulo desde `PyModuleDef`, usando `PyModule_Create` o `PyModule_FromDefAndSpec`.

Nuevo en la versión 3.5.

Funciones de soporte

La función de inicialización del módulo (si usa la inicialización de fase única) o una función llamada desde un intervalo de ejecución del módulo (si usa la inicialización de múltiples fases), puede usar las siguientes funciones para ayudar a inicializar el estado del módulo:

int **PyModule_AddObject** (*PyObject* *module, const char *name, *PyObject* *value)

Agrega un objeto a *module* como *name*. Esta es una función conveniente que se puede utilizar desde la función de inicialización del módulo. Esto roba una referencia al *value* en caso de éxito. Retorna `-1` en caso de error, `0` en caso de éxito.

Nota: A diferencia de otras funciones que roban referencias, `PyModule_AddObject()` solo disminuye el conteo de referencias de *value* en caso de éxito.

Esto significa que su valor de retorno debe ser verificado, y el código de llamada debe `Py_DECREF()` *value* manualmente en caso de error. Ejemplo de uso:

```
Py_INCREF(spam);
if (PyModule_AddObject(module, "spam", spam) < 0) {
    Py_DECREF(module);
    Py_DECREF(spam);
    return NULL;
}
```

int **PyModule_AddIntConstant** (*PyObject* *module, const char *name, long value)

Agrega una constante entera a *module* como *name*. Esta función de conveniencia se puede usar desde la función de inicialización del módulo. Retorna `-1` en caso de error, `0` en caso de éxito.

int **PyModule_AddStringConstant** (*PyObject* *module, const char *name, const char *value)

Agrega una constante de cadena a *module* como *name*. Esta función de conveniencia se puede usar desde la función de inicialización del módulo. La cadena de caracteres *value* debe estar terminada en `NULL`. Retorna `-1` en caso de error, `0` en caso de éxito.

int **PyModule_AddIntMacro** (*PyObject* *module, macro)

Agrega una constante `int` a *module*. El nombre y el valor se toman de *macro*. Por ejemplo, `PyModule_AddIntMacro(module, AF_INET)` agrega la constante `int AF_INET` con el valor de `AF_INET` a *module*. Retorna `-1` en caso de error, `0` en caso de éxito.

int **PyModule_AddStringMacro** (*PyObject* *module, macro)

Agrega una constante de cadena de caracteres a *module*.

int **PyModule_AddType** (*PyObject* *module, *PyTypeObject* *type)

Agrega un objeto tipo a *module*. El objeto tipo se finaliza llamando internamente *PyType_Ready()*. El nombre del objeto tipo se toma del último componente de *tp_name* después del punto. Retorna -1 en caso de error, 0 en caso de éxito.

Nuevo en la versión 3.9.

Búsqueda de módulos

La inicialización monofásica crea módulos singleton que se pueden buscar en el contexto del intérprete actual. Esto permite que el objeto módulo se recupere más tarde con solo una referencia a la definición del módulo.

Estas funciones no funcionarán en módulos creados mediante la inicialización de múltiples fases, ya que se pueden crear múltiples módulos de este tipo desde una sola definición.

*PyObject** **PyState_FindModule** (*PyModuleDef* *def)

Return value: Borrowed reference. Retorna el objeto módulo que se creó a partir de *def* para el intérprete actual. Este método requiere que el objeto módulo se haya adjuntado al estado del intérprete con *PyState_AddModule()* de antemano. En caso de que el objeto módulo correspondiente no se encuentre o no se haya adjuntado al estado del intérprete, retornará NULL.

int **PyState_AddModule** (*PyObject* *module, *PyModuleDef* *def)

Adjunta el objeto del módulo pasado a la función al estado del intérprete. Esto permite que se pueda acceder al objeto del módulo a través de *PyState_FindModule()*.

Solo es efectivo en módulos creados con la inicialización monofásica.

Python llama a *PyState_AddModule* automáticamente después de importar un módulo, por lo que es innecesario (pero inofensivo) llamarlo desde el código de inicialización del módulo. Solo se necesita una llamada explícita si el propio código de inicio del módulo llama posteriormente *PyState_FindModule*. La función está destinada principalmente a implementar mecanismos de importación alternativos (ya sea llamándolo directamente o refiriéndose a su implementación para obtener detalles de las actualizaciones de estado requeridas).

La persona que llama debe retener el GIL.

Retorna 0 en caso de éxito o -1 en caso de error.

Nuevo en la versión 3.3.

int **PyState_RemoveModule** (*PyModuleDef* *def)

Elimina el objeto del módulo creado a partir de *def* del estado del intérprete. Retorna 0 en caso de éxito o -1 en caso de error.

La persona que llama debe retener el GIL.

Nuevo en la versión 3.3.

8.6.3 Objetos iteradores

Python proporciona dos objetos iteradores de propósito general. El primero, un iterador de secuencia, funciona con una secuencia arbitraria que admite el método `__getitem__()`. El segundo funciona con un objeto invocable y un valor centinela, llamando al invocable para cada elemento de la secuencia y finalizando la iteración cuando se retorna el valor centinela.

PyObject **PySeqIter_Type**

Objeto tipo para objetos iteradores retornados por `PySeqIter_New()` y la forma de un argumento de la función incorporada `iter()` para los tipos de secuencia incorporados.

int **PySeqIter_Check** (op)

Retorna verdadero si el tipo de `op` es `PySeqIter_Type`. Esta función siempre finaliza con éxito.

*PyObject** **PySeqIter_New** (*PyObject* *seq)

Return value: *New reference.* Retorna un iterador que funciona con un objeto de secuencia general, `seq`. La iteración termina cuando la secuencia lanza `IndexError` para la operación de suscripción.

PyObject **PyCallIter_Type**

Objeto tipo para los objetos iteradores retornados por `PyCallIter_New()` y la forma de dos argumentos de la función incorporada `iter()`.

int **PyCallIter_Check** (op)

Retorna verdadero si el tipo de `op` es `PyCallIter_Type`. Esta función siempre finaliza con éxito.

*PyObject** **PyCallIter_New** (*PyObject* *callable, *PyObject* *sentinel)

Return value: *New reference.* Retorna un nuevo iterador. El primer parámetro, `callable`, puede ser cualquier objeto invocable de Python que se pueda invocar sin parámetros; cada llamada debe retornar el siguiente elemento en la iteración. Cuando `callable` retorna un valor igual a `sentinel`, la iteración finalizará.

8.6.4 Objetos descriptores

Los «descriptores» son objetos que describen algún atributo de un objeto. Se encuentran en el diccionario de objetos tipo.

PyObject **PyProperty_Type**

El objeto de tipo para los tipos de descriptor incorporado.

*PyObject** **PyDescr_NewGetSet** (*PyObject* *type, struct *PyGetSetDef* *getset)

Return value: *New reference.*

*PyObject** **PyDescr_NewMember** (*PyObject* *type, struct *PyMemberDef* *meth)

Return value: *New reference.*

*PyObject** **PyDescr_NewMethod** (*PyObject* *type, struct *PyMethodDef* *meth)

Return value: *New reference.*

*PyObject** **PyDescr_NewWrapper** (*PyObject* *type, struct wrapperbase *wrapper, void *wrapped)

Return value: *New reference.*

*PyObject** **PyDescr_NewClassMethod** (*PyObject* *type, *PyMethodDef* *method)

Return value: *New reference.*

int **PyDescr_IsData** (*PyObject* *descr)

Retorna verdadero si el descriptor objetos `descr` describe un atributo de datos, o falso si describe un método. `descr` debe ser un objeto descriptor; No hay comprobación de errores.

*PyObject** **PyWrapper_New** (*PyObject* *, *PyObject* *)

Return value: *New reference.*

8.6.5 Objeto rebanada (*slice*)

PyObject **PySlice_Type**

El objeto tipo para objetos rebanadas. Esto es lo mismo que *slice* en la capa de Python.

int **PySlice_Check** (*PyObject* **ob*)

Retorna verdadero si *ob* es un objeto rebanada; *ob* no debe ser NULL. Esta función siempre finaliza con éxito.

*PyObject** **PySlice_New** (*PyObject* **start*, *PyObject* **stop*, *PyObject* **step*)

Return value: New reference. Retorna un nuevo objeto rebanada con los valores dados. Los parámetros *start*, *stop* y *step* se utilizan como los valores de los atributos del objeto rebanada de los mismos nombres. Cualquiera de los valores puede ser NULL, en cuyo caso se usará None para el atributo correspondiente. Retorna NULL si no se puede asignar el nuevo objeto.

int **PySlice_GetIndices** (*PyObject* **slice*, *Py_ssize_t* *length*, *Py_ssize_t* **start*, *Py_ssize_t* **stop*, *Py_ssize_t* **step*)

Recupera los índices *start*, *stop* y *step* del objeto rebanada *slice*, suponiendo una secuencia de longitud *length*. Trata los índices mayores que *length* como errores.

Retorna 0 en caso de éxito y -1 en caso de error sin excepción establecida (a menos que uno de los índices no sea None y no se haya convertido a un entero, en cuyo caso “-1” se retorna con una excepción establecida).

Probablemente no quiera usar esta función.

Distinto en la versión 3.2: El tipo de parámetro para el parámetro *slice* era *PySliceObject** antes.

int **PySlice_GetIndicesEx** (*PyObject* **slice*, *Py_ssize_t* *length*, *Py_ssize_t* **start*, *Py_ssize_t* **stop*, *Py_ssize_t* **step*, *Py_ssize_t* **slicelength*)

Reemplazo utilizable para *PySlice_GetIndices()*. Recupera los índices de *start*, *stop*, y *step* del objeto rebanada *slice* asumiendo una secuencia de longitud *length*, y almacena la longitud de la rebanada en *slicelength*. Los índices fuera de los límites se recortan de manera coherente con el manejo de sectores normales.

Retorna 0 en caso de éxito y -1 en caso de error con excepción establecida.

Nota: Esta función se considera no segura para secuencias redimensionables. Su invocación debe ser reemplazada por una combinación de *PySlice_Unpack()* y *PySlice_AdjustIndices()* donde:

```
if (PySlice_GetIndicesEx(slice, length, &start, &stop, &step, &slicelength) < 0) {  
    // return error  
}
```

es reemplazado por:

```
if (PySlice_Unpack(slice, &start, &stop, &step) < 0) {  
    // return error  
}  
slicelength = PySlice_AdjustIndices(length, &start, &stop, step);
```

Distinto en la versión 3.2: El tipo de parámetro para el parámetro *slice* era *PySliceObject** antes.

Distinto en la versión 3.6.1: Si *Py_LIMITED_API* no se establece o establece el valor entre 0x03050400 y 0x03060000 (sin incluir) o 0x03060100 o un superior *PySlice_GetIndicesEx()* se implementa como un macro usando *PySlice_Unpack()* y *PySlice_AdjustIndices()*. Los argumentos *start*, *stop* y *step* se evalúan más de una vez.

Obsoleto desde la versión 3.6.1: Si *Py_LIMITED_API* se establece en un valor menor que 0x03050400 o entre 0x03060000 y 0x03060100 (sin incluir) *PySlice_GetIndicesEx()* es una función obsoleta.

int PySlice_Unpack (*PyObject* *slice, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step)

Extrae los miembros de datos *start*, *stop*, y *step* de un objeto rebanada como enteros en C. Reduce silenciosamente los valores mayores que PY_SSIZE_T_MAX a PY_SSIZE_T_MAX, aumenta silenciosamente los valores *start* y *stop* inferiores a PY_SSIZE_T_MIN a PY_SSIZE_T_MIN, y silenciosamente aumenta los valores de *step* a menos de -PY_SSE `` a `` -PY_SSIZE_T_MAX.

Retorna -1 en caso de error, 0 en caso de éxito.

Nuevo en la versión 3.6.1.

Py_ssize_t **PySlice_AdjustIndices** (*Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* step)

Ajusta los índices de corte de inicio/fin asumiendo una secuencia de la longitud especificada. Los índices fuera de los límites se recortan de manera coherente con el manejo de sectores normales.

Retorna la longitud de la rebanada. Siempre exitoso. No llama al código de Python.

Nuevo en la versión 3.6.1.

8.6.6 Objeto Ellipsis

PyObject ***Py_Ellipsis**

El objeto `Ellipsis` de Python. Este objeto no tiene métodos. Debe tratarse como cualquier otro objeto con respecto a los recuentos de referencia. Como `Py_None` es un objeto singleton.

8.6.7 Objetos de vista de memoria (*MemoryView*)

Un objeto `memoryview` expone la *interfaz de búfer* a nivel de C como un objeto Python que luego puede pasarse como cualquier otro objeto.

PyObject ***PyMemoryView_FromObject** (*PyObject* *obj)

Return value: *New reference.* Crea un objeto de vista de memoria *memoryview* a partir de un objeto que proporciona la interfaz del búfer. Si *obj* admite exportaciones de búfer de escritura, el objeto de vista de memoria será de lectura/escritura, de lo contrario puede ser de solo lectura o de lectura/escritura a discreción del exportador.

PyObject ***PyMemoryView_FromMemory** (char *mem, *Py_ssize_t* size, int flags)

Return value: *New reference.* Crea un objeto de vista de memoria usando *mem* como el búfer subyacente. *flags* pueden ser uno de `PyBUF_READ` o `PyBUF_WRITE`.

Nuevo en la versión 3.3.

PyObject ***PyMemoryView_FromBuffer** (*Py_buffer* *view)

Return value: *New reference.* Crea un objeto de vista de memoria que ajuste la estructura de búfer dada *view*. Para memorias intermedias de bytes simples, `PyMemoryView_FromMemory()` es la función preferida.

PyObject ***PyMemoryView_GetContiguous** (*PyObject* *obj, int buffertype, char order)

Return value: *New reference.* Crea un objeto de vista de memoria *memoryview* para un fragmento de memoria contiguo (*contiguous*, en *order* "C" o "F" de Fortran) desde un objeto que define la interfaz del búfer. Si la memoria es contigua, el objeto de vista de memoria apunta a la memoria original. De lo contrario, se realiza una copia y la vista de memoria apunta a un nuevo objeto de bytes.

int PyMemoryView_Check (*PyObject* *obj)

Retorna verdadero si el objeto *obj* es un objeto de vista de memoria. Actualmente no está permitido crear subclases de `memoryview`. Esta función siempre finaliza con éxito.

Py_buffer ***PyMemoryView_GET_BUFFER** (*PyObject* *mview)

Retorna un puntero a la copia privada de la vista de memoria del búfer del exportador. *mview* **debe** ser una instancia de *memoryview*; este macro no verifica su tipo, debe hacerlo usted mismo o correrá el riesgo de fallas.

Py_buffer ***PyMemoryView_GET_BASE** (*PyObject* *mview)

Retorna un puntero al objeto de exportación en el que se basa la vista de memoria o NULL si la vista de memoria ha sido creada por una de las funciones *PyMemoryView_FromMemory()* o *PyMemoryView_FromBuffer()*. *mview* **debe** ser una instancia de *memoryview*.

8.6.8 Objetos de referencia débil

Python soporta *referencias débiles* como objetos de primera clase. Hay dos tipos de objetos específicos que implementan directamente referencias débiles. El primero es un objeto con referencia simple, y el segundo actúa como un proxy del objeto original tanto como pueda.

int **PyWeakref_Check** (ob)

Retorna verdadero (true) si *ob* es una referencia o un objeto proxy. Esta función siempre finaliza con éxito.

int **PyWeakref_CheckRef** (ob)

Retorna verdadero (true) si *ob* es un objeto de referencia. Esta función siempre finaliza con éxito.

int **PyWeakref_CheckProxy** (ob)

Retorna verdadero (true) si *ob* es un objeto proxy. Esta función siempre finaliza con éxito.

*PyObject** **PyWeakref_NewRef** (*PyObject* *ob, *PyObject* *callback)

Return value: New reference. Retorna un objeto de referencia débil para el objeto *ob*. Esto siempre retornará una nueva referencia, pero no garantiza la creación de un objeto nuevo; un objeto de referencia ya existente puede ser retornado. El segundo parámetro, *callback*, puede ser un objeto invocable que recibe una notificación cuando *ob* es recolectado como basura; debe aceptar un solo parámetro, el cual será el mismo objeto de referencia débil. *callback* también puede ser None o NULL. Si *ob* no es un objeto que puede ser referido de forma débil, o si *callback* no es invocable, None, o NULL, esto retornará NULL y causará un *TypeError*.

*PyObject** **PyWeakref_NewProxy** (*PyObject* *ob, *PyObject* *callback)

Return value: New reference. Retorna un objeto proxy de referencia débil para el objeto *ob*. Esto siempre retornará una nueva referencia, pero no garantiza la creación de un objeto nuevo; un objeto proxy de referencia ya existente puede ser retornado. El segundo parámetro, *callback*, puede ser un objeto invocable que recibe una notificación cuando *ob* es recolectado como basura; debe aceptar un solo parámetro, el cual será el mismo objeto de referencia débil. *callback* también puede ser None o NULL. Si *ob* no es un objeto que puede ser referido de forma débil, o si *callback* no es invocable, None, o NULL, esto retornará NULL y causará un *TypeError*.

*PyObject** **PyWeakref_GetObject** (*PyObject* *ref)

Return value: Borrowed reference. Retorna el objeto referenciado desde una referencia débil, *ref*. Si el referente no está vivo, retornará *Py_None*.

Nota: Esta función retorna una *referencia prestada* al objeto referenciado. Esto significa que siempre debes llamar *Py_INCREF()* en el objeto excepto si sabes que no puede ser destruido mientras lo estés usando.

*PyObject** **PyWeakref_GET_OBJECT** (*PyObject* *ref)

Return value: Borrowed reference. Similar a *PyWeakref_GetObject()*, pero implementado como un macro que no verifica errores.

8.6.9 Cápsulas

Consulta `using-capsules` para obtener más información sobre el uso de estos objetos.

Nuevo en la versión 3.1.

PyCapsule

Este subtipo de *PyObject* representa un valor opaco, útil para los módulos de extensión C que necesitan pasar un valor opaco (como un puntero `void*`) a través del código Python a otro código C. A menudo se usa para hacer que un puntero de función C definido en un módulo esté disponible para otros módulos, por lo que el mecanismo de importación regular se puede usar para acceder a las API C definidas en módulos cargados dinámicamente.

PyCapsule_Destructor

El tipo de devolución de llamada de un destructor para una cápsula. Definido como:

```
typedef void (*PyCapsule_Destructor) (PyObject *);
```

Consulte *PyCapsule_New()* para conocer la semántica de las devoluciones de llamada de *PyCapsule_Destructor*.

`int PyCapsule_CheckExact (PyObject *p)`

Retorna verdadero si su argumento es a *PyCapsule*. Esta función siempre finaliza con éxito.

*PyObject** **PyCapsule_New** (void *pointer, const char *name, *PyCapsule_Destructor* destructor)

Return value: New reference. Crea un *PyCapsule* encapsulando el *pointer*. El argumento *pointer* puede no ser NULL.

En caso de falla, establece una excepción y retorna NULL.

La cadena de caracteres *name* puede ser NULL o un puntero a una cadena C válida. Si no es NULL, esta cadena de caracteres debe sobrevivir a la cápsula. (Aunque está permitido liberarlo dentro del *destructor*).

Si el argumento *destructor* no es NULL, se llamará con la cápsula como argumento cuando se destruya.

Si esta cápsula se almacenará como un atributo de un módulo, el nombre *name* debe especificarse como `modulename.attributename`. Esto permitirá que otros módulos importen la cápsula usando *PyCapsule_Import()*.

`void*` **PyCapsule_GetPointer** (*PyObject* *capsule, const char *name)

Recupera el *pointer* almacenado en la cápsula. En caso de falla, establece una excepción y retorna NULL.

El parámetro *name* debe compararse exactamente con el nombre almacenado en la cápsula. Si el nombre almacenado en la cápsula es NULL, el *name* pasado también debe ser NULL. Python usa la función C `strcmp()` para comparar nombres de cápsulas.

PyCapsule_Destructor **PyCapsule_GetDestructor** (*PyObject* *capsule)

Retorna el destructor actual almacenado en la cápsula. En caso de falla, establece una excepción y retorna NULL.

Es legal que una cápsula tenga un destructor NULL. Esto hace que un código de retorno NULL sea algo ambiguo; use *PyCapsule_IsValid()* o *PyErr_Occurred()* para desambiguar.

`void*` **PyCapsule_GetContext** (*PyObject* *capsule)

Retorna el contexto actual almacenado en la cápsula. En caso de falla, establece una excepción y retorna NULL.

Es legal que una cápsula tenga un contexto NULL. Esto hace que un código de retorno NULL sea algo ambiguo; use *PyCapsule_IsValid()* o *PyErr_Occurred()* para desambiguar.

`const char*` **PyCapsule_GetName** (*PyObject* *capsule)

Retorna el nombre actual almacenado en la cápsula. En caso de falla, establece una excepción y retorna NULL.

Es legal que una cápsula tenga un nombre NULL. Esto hace que un código de retorno NULL sea algo ambiguo; use *PyCapsule_IsValid()* o *PyErr_Occurred()* para desambiguar.

`void* PyCapsule_Import (const char *name, int no_block)`

Importa un puntero a un objeto C desde un atributo cápsula en un módulo. El parámetro *name* debe especificar el nombre completo del atributo, como en `module.attribute`. El nombre *name* almacenado en la cápsula debe coincidir exactamente con esta cadena de caracteres. Si *no_block* es verdadero, importa el módulo sin bloquear (usando `PyImport_ImportModuleNoBlock()`). Si *no_block* es falso, importa el módulo convencionalmente (usando `PyImport_ImportModule()`).

Retorna el puntero *pointer* interno de la cápsula en caso de éxito. En caso de falla, establece una excepción y retorna NULL.

`int PyCapsule_IsValid (PyObject *capsule, const char *name)`

Determina si *capsule* es o no una cápsula válida. Una cápsula válida no es NULL, pasa `PyCapsule_CheckExact()`, tiene un puntero no NULL almacenado y su nombre interno coincide con el parámetro *name*. (Consulte `PyCapsule_GetPointer()` para obtener información sobre cómo se comparan los nombres de las cápsulas).

En otras palabras, si `PyCapsule_IsValid()` retorna un valor verdadero, las llamadas a cualquiera de las funciones de acceso (cualquier función que comience con `PyCapsule_Get()`) tienen éxito.

Retorna un valor distinto de cero si el objeto es válido y coincide con el nombre pasado. Retorna 0 de lo contrario. Esta función no fallará.

`int PyCapsule_SetContext (PyObject *capsule, void *context)`

Establece el puntero de contexto dentro de *capsule* a *context*.

Retorna 0 en caso de éxito. Retorna distinto de cero y establece una excepción en caso de error.

`int PyCapsule_SetDestructor (PyObject *capsule, PyCapsule_Destructor destructor)`

Establece el destructor dentro de *capsule* en *destructor*.

Retorna 0 en caso de éxito. Retorna distinto de cero y establece una excepción en caso de error.

`int PyCapsule_SetName (PyObject *capsule, const char *name)`

Establece el nombre dentro de *capsule* a *name*. Si no es NULL, el nombre debe sobrevivir a la cápsula. Si el *name* anterior almacenado en la cápsula no era NULL, no se intenta liberarlo.

Retorna 0 en caso de éxito. Retorna distinto de cero y establece una excepción en caso de error.

`int PyCapsule_SetPointer (PyObject *capsule, void *pointer)`

Establece el puntero vacío dentro de *capsule* a *pointer*. El puntero puede no ser NULL.

Retorna 0 en caso de éxito. Retorna distinto de cero y establece una excepción en caso de error.

8.6.10 Objetos Generadores

Los objetos generadores son lo que Python usa para implementar iteradores generadores. Normalmente se crean iterando sobre una función que produce valores, en lugar de llamar explícitamente `PyGen_New()` o `PyGen_NewWithQualName()`.

PyGenObject

La estructura en C utilizada para los objetos generadores.

PyTypeObject PyGen_Type

El objeto tipo correspondiente a los objetos generadores.

`int PyGen_Check (PyObject *ob)`

Retorna verdadero si *ob* es un objeto generador; *ob* no debe ser NULL. Esta función siempre finaliza con éxito.

`int PyGen_CheckExact (PyObject *ob)`

Retorna verdadero si el tipo de *ob* es `PyGen_Type`; *ob* no debe ser NULL. Esta función siempre finaliza con éxito.

*PyObject** **PyGen_New** (*PyFrameObject* *frame)

Return value: *New reference.* Crea y retorna un nuevo objeto generador basado en el objeto *frame*. Una referencia a *frame* es robada por esta función. El argumento no debe ser NULL.

*PyObject** **PyGen_NewWithQualName** (*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

Return value: *New reference.* Crea y retorna un nuevo objeto generador basado en el objeto *frame*, con `__name__` y `__qualname__` establecido en *name* y *qualname*. Una referencia a *frame* es robada por esta función. El argumento *frame* no debe ser NULL.

8.6.11 Objetos corrutina

Nuevo en la versión 3.5.

Los objetos de corrutina son las funciones declaradas con un retorno de palabra clave `async`.

PyCoroObject

La estructura en C utilizada para objeto corrutina.

PyTypeObject **PyCoro_Type**

El tipo de objeto correspondiente a los objetos corrutina.

int **PyCoro_CheckExact** (*PyObject* *ob)

Retorna verdadero si el tipo de *ob* es *PyCoro_Type*; *ob* no debe ser NULL. Esta función siempre finaliza con éxito.

*PyObject** **PyCoro_New** (*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

Return value: *New reference.* Crea y retorna un nuevo objeto corrutina basado en el objeto *frame*, con `__name__` y `__qualname__` establecido en *name* y *qualname*. Una referencia a *frame* es robada por esta función. El argumento *frame* no debe ser NULL.

8.6.12 Objetos de variables de contexto

Nota: Distinto en la versión 3.7.1: En Python 3.7.1, las firmas de todas las variables de contexto C APIs fueron **cambiadas** para usar punteros *PyObject* en lugar de *PyContext*, *PyContextVar*, y *PyContextToken*, por ejemplo:

```
// in 3.7.0:
PyContext *PyContext_New(void);

// in 3.7.1+:
PyObject *PyContext_New(void);
```

Ver [bpo-34762](#) para más detalles.

Nuevo en la versión 3.7.

Esta sección detalla la API pública de C para el módulo `contextvars`.

PyContext

La estructura C utilizada para representar un objeto `contextvars.Context`.

PyContextVar

La estructura C utilizada para representar un objeto `contextvars.ContextVar`.

PyContextToken

La estructura C solía representar un objeto `contextvars.Token`.

***PyObject* PyContext_Type**

El objeto de tipo que representa el tipo *context*.

***PyObject* PyContextVar_Type**

El objeto tipo que representa el tipo *variable de contexto*.

***PyObject* PyContextToken_Type**

El tipo objeto que representa el tipo *token de variable de contexto*.

Macros de verificación de tipo:

int PyContext_CheckExact (*PyObject* *o)

Retorna verdadero si *o* es de tipo *PyContext_Type*. *o* no debe ser NULL. Esta función siempre finaliza con éxito.

int PyContextVar_CheckExact (*PyObject* *o)

Retorna verdadero si *o* es de tipo *PyContextVar_Type*. *o* no debe ser NULL. Esta función siempre finaliza con éxito.

int PyContextToken_CheckExact (*PyObject* *o)

Retorna verdadero si *o* es de tipo *PyContextToken_Type*. *o* no debe ser NULL. Esta función siempre finaliza con éxito.

Funciones de gestión de objetos de contexto:

***PyObject* *PyContext_New (void)**

Return value: New reference. Crea un nuevo objeto de contexto vacío. Retorna NULL si se ha producido un error.

***PyObject* *PyContext_Copy (*PyObject* *ctx)**

Return value: New reference. Crea una copia superficial del objeto de contexto *ctx* pasado. Retorna NULL si se ha producido un error.

***PyObject* *PyContext_CopyCurrent (void)**

Return value: New reference. Crea una copia superficial del contexto actual del hilo. Retorna NULL si se ha producido un error.

int PyContext_Enter (*PyObject* *ctx)

Establece *ctx* como el contexto actual para el hilo actual. Retorna 0 en caso de éxito y -1 en caso de error.

int PyContext_Exit (*PyObject* *ctx)

Desactiva el contexto *ctx* y restaura el contexto anterior como el contexto actual para el hilo actual. Retorna 0 en caso de éxito y -1 en caso de error.

Funciones variables de contexto:

***PyObject* *PyContextVar_New (const char *name, *PyObject* *def)**

Return value: New reference. Create a new ContextVar object. The *name* parameter is used for introspection and debug purposes. The *def* parameter specifies a default value for the context variable, or NULL for no default. If an error has occurred, this function returns NULL.

int PyContextVar_Get (*PyObject* *var, *PyObject* *default_value, *PyObject* **value)

Obtiene el valor de una variable de contexto. Retorna -1 si se produjo un error durante la búsqueda y 0 si no se produjo ningún error, se haya encontrado o no un valor.

Si se encontró la variable de contexto, *value* será un puntero a ella. Si la variable de contexto *not* se encontró, *value* apuntará a:

- *default_value*, si no es NULL;
- el valor predeterminado de *var*, si no es NULL;
- NULL

Except for NULL, the function returns a new reference.

PyObject *PyContextVar_Set (*PyObject* *var, *PyObject* *value)

Return value: New reference. Set the value of *var* to *value* in the current context. Returns a new token object for this change, or NULL if an error has occurred.

int PyContextVar_Reset (*PyObject* *var, *PyObject* *token)

Restablece el estado de la variable de contexto *var* a la que estaba antes *PyContextVar_Set* () que retornó el *token* fue llamado. Esta función retorna 0 en caso de éxito y -1 en caso de error.

8.6.13 Objetos *DateTime*

El módulo `datetime` proporciona varios objetos de fecha y hora. Antes de usar cualquiera de estas funciones, el archivo de encabezado `datetime.h` debe estar incluido en su fuente (tenga en cuenta que esto no está incluido en el archivo `Python.h`), y la macro `PyDateTime_IMPORT` debe llamarse, generalmente como parte de la función de inicialización del módulo. La macro coloca un puntero a una estructura C en una variable estática, `PyDateTimeAPI`, que utilizan las siguientes macros.

Macro para acceder al singleton UTC:

*PyObject** PyDateTime_TimeZone_UTC

Retorna la zona horaria singleton que representa UTC, el mismo objeto que `datetime.timezone.utc`.

Nuevo en la versión 3.7.

Macros de verificación de tipo:

int PyDate_Check (*PyObject* *ob)

Retorna verdadero si *ob* es de tipo `PyDateTime_DateType` o un subtipo de `PyDateTime_DateType`. *ob* no debe ser NULL. Esta función siempre finaliza con éxito.

int PyDate_CheckExact (*PyObject* *ob)

Retorna verdadero si *ob* es de tipo `PyDateTime_DateType`. *ob* no debe ser NULL. Esta función siempre finaliza con éxito.

int PyDateTime_Check (*PyObject* *ob)

Retorna verdadero si *ob* es de tipo `PyDateTime_DateTimeType` o un subtipo de `PyDateTime_DateTimeType`. *ob* no debe ser NULL. Esta función siempre finaliza con éxito.

int PyDateTime_CheckExact (*PyObject* *ob)

Retorna verdadero si *ob* es de tipo `PyDateTime_DateTimeType`. *ob* no debe ser NULL. Esta función siempre finaliza con éxito.

int PyTime_Check (*PyObject* *ob)

Retorna verdadero si *ob* es de tipo `PyDateTime_TimeType` o un subtipo de `PyDateTime_TimeType`. *ob* no debe ser NULL. Esta función siempre finaliza con éxito.

int PyTime_CheckExact (*PyObject* *ob)

Retorna verdadero si *ob* es de tipo `PyDateTime_TimeType`. *ob* no debe ser NULL. Esta función siempre finaliza con éxito.

int PyDelta_Check (*PyObject* *ob)

Retorna verdadero si *ob* es de tipo `PyDateTime_DeltaType` o un subtipo de `PyDateTime_DeltaType`. *ob* no debe ser NULL. Esta función siempre finaliza con éxito.

int PyDelta_CheckExact (*PyObject* *ob)

Retorna verdadero si *ob* es de tipo `PyDateTime_DeltaType`. *ob* no debe ser NULL. Esta función siempre finaliza con éxito.

int PyTZInfo_Check (*PyObject* *ob)

Retorna verdadero si *ob* es de tipo `PyDateTime_TZInfoType` o un subtipo de `PyDateTime_TZInfoType`. *ob* no debe ser NULL. Esta función siempre finaliza con éxito.

int **PyTZInfo_CheckExact** (*PyObject* **ob*)

Retorna verdadero si *ob* es de tipo `PyDateTime_TZInfoType`. *ob* no debe ser NULL. Esta función siempre finaliza con éxito.

Macros para crear objetos:

*PyObject** **PyDate_FromDate** (int *year*, int *month*, int *day*)

Return value: *New reference*. Retorna un objeto `datetime.date` con el año, mes y día especificados.

*PyObject** **PyDateTime_FromDateAndTime** (int *year*, int *month*, int *day*, int *hour*, int *minute*, int *second*,
int *usecond*)

Return value: *New reference*. Retorna un objeto `datetime.datetime` con el año, mes, día, hora, minuto, segundo y micro segundo especificados.

*PyObject** **PyDateTime_FromDateAndTimeAndFold** (int *year*, int *month*, int *day*, int *hour*, int *minute*,
int *second*, int *usecond*, int *fold*)

Return value: *New reference*. Retorna un objeto `datetime.datetime` con el año, mes, día, hora, minuto, segundo, micro segundo y doblez especificados.

Nuevo en la versión 3.6.

*PyObject** **PyTime_FromTime** (int *hour*, int *minute*, int *second*, int *usecond*)

Return value: *New reference*. Retorna un objeto `datetime.time` con la hora, minuto, segundo y micro segundo especificados.

*PyObject** **PyTime_FromTimeAndFold** (int *hour*, int *minute*, int *second*, int *usecond*, int *fold*)

Return value: *New reference*. Retorna un objeto `datetime.time` con la hora, minuto, segundo, micro segundo y doblez especificados.

Nuevo en la versión 3.6.

*PyObject** **PyDelta_FromDSU** (int *days*, int *seconds*, int *useconds*)

Return value: *New reference*. Retorna un objeto `datetime.timedelta` que representa el número dado de días, segundos y micro segundos. La normalización se realiza de modo que el número resultante de micro segundos y segundos se encuentre en los rangos documentados para los objetos `datetime.timedelta`.

*PyObject** **PyTimeZone_FromOffset** (*PyDateTime_DeltaType** *offset*)

Return value: *New reference*. Retorna un objeto `datetime.timezone` con un desplazamiento fijo sin nombre representado por el argumento *offset*.

Nuevo en la versión 3.7.

*PyObject** **PyTimeZone_FromOffsetAndName** (*PyDateTime_DeltaType** *offset*, *PyUnicode** *name*)

Return value: *New reference*. Retorna un objeto `datetime.timezone` con un desplazamiento fijo representado por el argumento *offset* y con *tzname name*.

Nuevo en la versión 3.7.

Macros para extraer campos de objetos de fecha. El argumento debe ser una instancia de `PyDateTime_Date`, incluidas las subclases (como `PyDateTime_DateTime`). El argumento no debe ser NULL y el tipo no está marcado:

int **PyDateTime_GET_YEAR** (*PyDateTime_Date* **o*)

Regrese el año, como un int positivo.

int **PyDateTime_GET_MONTH** (*PyDateTime_Date* **o*)

Regresa el mes, como int del 1 al 12.

int **PyDateTime_GET_DAY** (*PyDateTime_Date* **o*)

Retorna el día, como int del 1 al 31.

Macros para extraer campos de objetos de fecha y hora. El argumento debe ser una instancia de `PyDateTime_DateTime`, incluidas las subclases. El argumento no debe ser NULL y el tipo no es comprobado:

`int PyDateTime_DATE_GET_HOUR (PyDateTime_DateTime *o)`

Retorna la hora, como un int de 0 hasta 23.

`int PyDateTime_DATE_GET_MINUTE (PyDateTime_DateTime *o)`

Retorna el minuto, como un int de 0 hasta 59.

`int PyDateTime_DATE_GET_SECOND (PyDateTime_DateTime *o)`

Retorna el segundo, como un int de 0 hasta 59.

`int PyDateTime_DATE_GET_MICROSECOND (PyDateTime_DateTime *o)`

Retorna el micro segundo, como un int de 0 hasta 999999.

`int PyDateTime_DATE_GET_FOLD (PyDateTime_DateTime *o)`

Return the fold, as an int from 0 through 1.

Nuevo en la versión 3.6.

Macros para extraer campos de objetos de tiempo. El argumento debe ser una instancia de `PyDateTime_Time`, incluidas las subclases. El argumento no debe ser `NULL` y el tipo no está marcado:

`int PyDateTime_TIME_GET_HOUR (PyDateTime_Time *o)`

Retorna la hora, como un int de 0 hasta 23.

`int PyDateTime_TIME_GET_MINUTE (PyDateTime_Time *o)`

Retorna el minuto, como un int de 0 hasta 59.

`int PyDateTime_TIME_GET_SECOND (PyDateTime_Time *o)`

Retorna el segundo, como un int de 0 hasta 59.

`int PyDateTime_TIME_GET_MICROSECOND (PyDateTime_Time *o)`

Retorna el micro segundo, como un int de 0 hasta 999999.

`int PyDateTime_TIME_GET_FOLD (PyDateTime_Time *o)`

Return the fold, as an int from 0 through 1.

Nuevo en la versión 3.6.

Macros para extraer campos de objetos delta de tiempo. El argumento debe ser una instancia de `PyDateTime_Delta`, incluidas las subclases. El argumento no debe ser `NULL` y el tipo no está marcado:

`int PyDateTime_DELTA_GET_DAYS (PyDateTime_Delta *o)`

Retorna el número de días, como un int desde -999999999 a 999999999.

Nuevo en la versión 3.3.

`int PyDateTime_DELTA_GET_SECONDS (PyDateTime_Delta *o)`

Retorna el número de segundos, como un int de 0 a 86399.

Nuevo en la versión 3.3.

`int PyDateTime_DELTA_GET_MICROSECONDS (PyDateTime_Delta *o)`

Retorna el número de micro segundos, como un int de 0 a 999999.

Nuevo en la versión 3.3.

Macros para la conveniencia de módulos que implementan la API DB:

*PyObject** `PyDateTime_FromTimestamp (PyObject *args)`

Return value: New reference. Crea y retorna un nuevo objeto `datetime.datetime` dado una tupla de argumentos adecuada para pasar a `datetime.datetime.fromtimestamp()`.

*PyObject** `PyDate_FromTimestamp (PyObject *args)`

Return value: New reference. Crea y retorna un nuevo objeto `datetime.date` dado una tupla de argumentos adecuada para pasar a `datetime.date.fromtimestamp()`.

8.6.14 Objects for Type Hinting

Various built-in types for type hinting are provided. Only `GenericAlias` is exposed to C.

*PyObject** **Py_GenericAlias** (*PyObject* **origin*, *PyObject* **args*)

Create a `GenericAlias` object. Equivalent to calling the Python class `types.GenericAlias`. The *origin* and *args* arguments set the `GenericAlias`'s `__origin__` and `__args__` attributes respectively. *origin* should be a *PyTypeObject**, and *args* can be a *PyTupleObject** or any *PyObject**. If *args* passed is not a tuple, a 1-tuple is automatically constructed and `__args__` is set to `(args,)`. Minimal checking is done for the arguments, so the function will succeed even if *origin* is not a type. The `GenericAlias`'s `__parameters__` attribute is constructed lazily from `__args__`. On failure, an exception is raised and `NULL` is returned.

Here's an example of how to make an extension type generic:

```
...
static PyMethodDef my_obj_methods[] = {
    // Other methods.
    ...
    {"__class_getitem__", (PyCFunction)Py_GenericAlias, METH_O|METH_CLASS, "See_
↪ PEP 585"}
    ...
}
```

Ver también:

The data model method `__class_getitem__()`.

Nuevo en la versión 3.9.

PyTypeObject **Py_GenericAliasType**

The C type of the object returned by *Py_GenericAlias()*. Equivalent to `types.GenericAlias` in Python.

Nuevo en la versión 3.9.

Iniciación, Finalización e Hilos

Consulte también *Configuración de inicialización de Python*.

9.1 Antes de la inicialización de Python

En una aplicación que incorpora Python, se debe llamar a la función `Py_Initialize()` antes de usar cualquier otra función de API Python/C; con la excepción de algunas funciones y *variables de configuración global*.

Las siguientes funciones se pueden invocar de forma segura antes de que se inicializa Python:

- Funciones de configuración:

- `PyImport_AppendInittab()`
- `PyImport_ExtendInittab()`
- `PyInitFrozenExtensions()`
- `PyMem_SetAllocator()`
- `PyMem_SetupDebugHooks()`
- `PyObject_SetArenaAllocator()`
- `Py_SetPath()`
- `Py_SetProgramName()`
- `Py_SetPythonHome()`
- `Py_SetStandardStreamEncoding()`
- `PySys_AddWarnOption()`
- `PySys_AddXOption()`
- `PySys_ResetWarnOptions()`

- Funciones informativas:

- `Py_IsInitialized()`
- `PyMem_GetAllocator()`
- `PyObject_GetArenaAllocator()`
- `Py_GetBuildInfo()`
- `Py_GetCompiler()`
- `Py_GetCopyright()`
- `Py_GetPlatform()`
- `Py_GetVersion()`

- Utilidades:

- `Py_DecodeLocale()`

- Asignadores de memoria:

- `PyMem_RawMalloc()`
 - `PyMem_RawRealloc()`
 - `PyMem_RawCalloc()`
 - `PyMem_RawFree()`

Nota: Las siguientes funciones **no deben llamarse** antes de `Py_Initialize()`: `Py_EncodeLocale()`, `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, `Py_GetProgramFullPath()`, `Py_GetPythonHome()`, `Py_GetProgramName()` y `PyEval_InitThreads()`.

9.2 Variables de configuración global

Python tiene variables para la configuración global para controlar diferentes características y opciones. De forma predefinida, estos indicadores están controlados por opciones de línea de comando.

Cuando una opción establece un indicador, el valor del indicador es la cantidad de veces que se configuró la opción. Por ejemplo, `-b` establece `Py_BytesWarningFlag` en 1 y `-bb` establece `Py_BytesWarningFlag` en 2.

int `Py_BytesWarningFlag`

Emite una advertencia al comparar `bytes` o `bytearray` con `str` o `bytes` con `int`. Emite un error si es mayor o igual a 2.

Establecido por la opción `-b`.

int `Py_DebugFlag`

Activa la salida de depuración del analizador (solo para expertos, según las opciones de compilación).

Establecido por la opción `-d` y la variable de entorno `PYTHONDEBUG`.

int `Py_DontWriteBytecodeFlag`

Si se establece en un valor distinto de cero, Python no intentará escribir archivos `.pyc` en la importación de módulos fuente.

Establecido por la opción `-B` y la variable de entorno `PYTHONDONTWRITEBYTECODE`.

int `Py_FrozenFlag`

Suprime los mensajes de error al calcular la ruta de búsqueda del módulo en `Py_GetPath()`.

Indicador privado utilizado por los programas `_freeze_importlib` y `frozenmain`.

int `Py_HashRandomizationFlag`

Se establece en 1 si la variable de entorno `PYTHONHASHSEED` se establece en una cadena de caracteres no vacía.

Si el indicador no es cero, lee la variable de entorno `PYTHONHASHSEED` para inicializar la semilla de *hash* secreta.

int `Py_IgnoreEnvironmentFlag`

Ignorar todas las variables de entorno `PYTHON*`, por ejemplo `PYTHONPATH` y `PYTHONHOME`, eso podría establecerse.

Establecido por las opciones `-E` y `-I`.

int `Py_InspectFlag`

Cuando se pasa una secuencia de comandos (*script*) como primer argumento o se usa la opción `-c`, ingresa al modo interactivo después de ejecutar la secuencia de comandos o el comando, incluso cuando `sys.stdin` no parece ser un terminal.

Establecido por la opción `-i` y la variable de entorno `PYTHONINSPECT`.

int `Py_InteractiveFlag`

Establecido por la opción `-i`.

int `Py_IsolatedFlag`

Ejecuta Python en modo aislado. En modo aislado `sys.path` no contiene ni el directorio de la secuencia de comandos (*script*) ni el directorio de paquetes del sitio del usuario (*site-packages*).

Establecido por la opción `-I`.

Nuevo en la versión 3.4.

int `Py_LegacyWindowsFSEncodingFlag`

Si el indicador no es cero, use la codificación `mbcs` en lugar de la codificación UTF-8 para la codificación del sistema de archivos.

Establece en 1 si la variable de entorno `PYTHONLEGACYWINDOWSFSENCODING` está configurada en una cadena de caracteres no vacía.

Ver [PEP 529](#) para más detalles.

Disponibilidad: Windows.

int `Py_LegacyWindowsStdioFlag`

Si el indicador no es cero, use `io.FileIO` en lugar de `WindowsConsoleIO` para secuencias estándar `sys`.

Establece en 1 si la variable de entorno `PYTHONLEGACYWINDOWSSTDIO` está configurada en una cadena de caracteres no vacía.

Ver [PEP 528](#) para más detalles.

Disponibilidad: Windows.

int `Py_NoSiteFlag`

Deshabilita la importación del módulo `site` y las manipulaciones dependientes del sitio de `sys.path` que conlleva. También deshabilita estas manipulaciones si `site` se importa explícitamente más tarde (llama a `site.main()` si desea que se activen).

Establecido por la opción `-S`.

int `Py_NoUserSiteDirectory`

No agregue el directorio de paquetes de sitio del usuario (*site-packages*) a `sys.path`.

Establecido por las opciones `-s` y `-I`, y la variable de entorno `PYTHONNOUSERSITE`.

int `Py_OptimizeFlag`

Establecido por la opción `-O` y la variable de entorno `PYTHONOPTIMIZE`.

int `Py_QuietFlag`

No muestre los mensajes de *copyright* y de versión incluso en modo interactivo.

Establecido por la opción `-q`.

Nuevo en la versión 3.2.

int `Py_UnbufferedStdioFlag`

Obliga a las secuencias *stdout* y *stderr* a que no tengan búfer.

Establecido por la opción `-u` y la variable de entorno `PYTHONUNBUFFERED`.

int `Py_VerboseFlag`

Imprime un mensaje cada vez que se inicializa un módulo, mostrando el lugar (nombre de archivo o módulo incorporado) desde el que se carga. Si es mayor o igual a 2, imprime un mensaje para cada archivo que se verifica al buscar un módulo. También proporciona información sobre la limpieza del módulo a la salida.

Establecido por la opción `-v` y la variable de entorno `PYTHONVERBOSE`.

9.3 Inicializando y finalizando el intérprete

void `Py_Initialize()`

Inicializa el intérprete de Python. En una aplicación que incorpora Python, se debe llamar antes de usar cualquier otra función de API Python/C; vea *Antes de la inicialización de Python* para ver algunas excepciones.

Esto inicializa la tabla de módulos cargados (`sys.modules`) y crea los módulos fundamentales `builtins`, `__main__` y `sys`. También inicializa la ruta de búsqueda del módulo (`sys.path`). No establece `sys.argv`; use `PySys_SetArgvEx()` para eso. Este es un *no-op* cuando se llama por segunda vez (sin llamar primero a `Py_FinalizeEx()`). No hay valor de retorno; es un error fatal si falla la inicialización.

Nota: En Windows, cambia el modo de consola de `O_TEXT` a `O_BINARY`, lo que también afectará los usos de la consola que no sean de Python utilizando *C Runtime*.

void `Py_InitializeEx(int initsigs)`

Esta función funciona como `Py_Initialize()` si `initsigs` es 1. Si `initsigs` es 0, omite el registro de inicialización de los manejadores de señal, lo que podría ser útil cuando Python está incrustado.

int `Py_IsInitialized()`

Retorna verdadero (distinto de cero) cuando el intérprete de Python se ha inicializado, falso (cero) si no. Después de que se llama a `Py_FinalizeEx()`, esto retorna falso hasta que `Py_Initialize()` se llama de nuevo.

int `Py_FinalizeEx()`

Deshace todas las inicializaciones realizadas por `Py_Initialize()` y el uso posterior de las funciones de Python/C API, y destruye todos los sub-intérpretes (ver `Py_NewInterpreter()` a continuación) que se crearon y aún no se destruyeron desde el última llamada a `Py_Initialize()`. Idealmente, esto libera toda la memoria asignada por el intérprete de Python. Este es un *no-op* cuando se llama por segunda vez (sin llamar a `Py_Initialize()` nuevamente primero). Normalmente el valor de retorno es 0. Si hubo errores durante la finalización (lavado de datos almacenados en el búfer), se retorna -1.

Esta función se proporciona por varias razones. Una aplicación de incrustación puede querer reiniciar Python sin tener que reiniciar la aplicación misma. Una aplicación que ha cargado el intérprete de Python desde una biblioteca cargable dinámicamente (o DLL) puede querer liberar toda la memoria asignada por Python antes de descargar la

DLL. Durante una búsqueda de pérdidas de memoria en una aplicación, un desarrollador puede querer liberar toda la memoria asignada por Python antes de salir de la aplicación.

Errores y advertencias: La destrucción de módulos y objetos en módulos se realiza en orden aleatorio; esto puede causar que los destructores (métodos `__del__()`) fallen cuando dependen de otros objetos (incluso funciones) o módulos. Los módulos de extensión cargados dinámicamente cargados por Python no se descargan. Es posible que no se liberen pequeñas cantidades de memoria asignadas por el intérprete de Python (si encuentra una fuga, informe por favor). La memoria atada en referencias circulares entre objetos no se libera. Es posible que parte de la memoria asignada por los módulos de extensión no se libere. Algunas extensiones pueden no funcionar correctamente si su rutina de inicialización se llama más de una vez; Esto puede suceder si una aplicación llama a `Py_Initialize()` y `Py_FinalizeEx()` más de una vez.

Genera un evento de auditoría `cpython._PySys_ClearAuditHooks` sin argumentos.

Nuevo en la versión 3.6.

void **Py_Finalize()**

Esta es una versión compatible con versiones anteriores de `Py_FinalizeEx()` que ignora el valor de retorno.

9.4 Parámetros de todo el proceso

int **Py_SetStandardStreamEncoding**(const char *encoding, const char *errors)

Esta función debería llamarse antes de `Py_Initialize()`, si es que se llama. Especifica qué codificación y manejo de errores usar con IO estándar, con los mismos significados que en `str.encode()`.

Reemplaza los valores `PYTHONIOENCODING`, y permite incrustar código para controlar la codificación IO cuando la variable de entorno no funciona.

codificación o *errores* pueden ser `NULL` para usar `PYTHONIOENCODING` o valores predeterminados (dependiendo de otras configuraciones).

Tenga en cuenta que `sys.stderr` siempre usa el controlador de error «*backslashreplace*», independientemente de esta configuración (o cualquier otra).

Si se llama a `Py_FinalizeEx()`, será necesario volver a llamar a esta función para afectar las llamadas posteriores a `Py_Initialize()`.

Retorna 0 si tiene éxito, un valor distinto de cero en caso de error (por ejemplo, llamar después de que el intérprete ya se haya inicializado)

Nuevo en la versión 3.4.

void **Py_SetProgramName**(const wchar_t *name)

Esta función debería llamarse antes `Py_Initialize()` se llama por primera vez, si es que se llama. Le dice al intérprete el valor del argumento `argv[0]` para la función `main()` del programa (convertido a caracteres anchos). Esto es utilizado por `Py_GetPath()` y algunas otras funciones a continuación para encontrar las bibliotecas de tiempo de ejecución de Python relativas al ejecutable del intérprete. El valor predeterminado es 'python'. El argumento debe apuntar a una cadena de caracteres anchos terminada en cero en almacenamiento estático cuyo contenido no cambiará mientras dure la ejecución del programa. Ningún código en el intérprete de Python cambiará el contenido de este almacenamiento.

Use `Py_DecodeLocale()` para decodificar una cadena de bytes para obtener una cadena `wchar_*`.

wchar* **Py_GetProgramName**()

Retorna el nombre del programa establecido con `Py_SetProgramName()`, o el valor predeterminado. La cadena de caracteres retornada apunta al almacenamiento estático; la persona que llama no debe modificar su valor.

wchar_t* **Py_GetPrefix**()

Retorna el prefijo *prefix* para los archivos instalados independientes de la plataforma. Esto se deriva a través de

una serie de reglas complicadas del nombre del programa establecido con `Py_SetProgramName()` y algunas variables de entorno; por ejemplo, si el nombre del programa es `'/usr/local/bin/python'`, el prefijo es `'/usr/local'`. La cadena de caracteres retornada apunta al almacenamiento estático; la persona que llama no debe modificar su valor. Esto corresponde a la variable **prefix** en el archivo de nivel superior `Makefile` y el argumento `--prefix` a la secuencia de comandos (*script*) **configure** en tiempo de compilación. El valor está disponible para el código de Python como `sys.prefix`. Solo es útil en Unix. Ver también la siguiente función.

wchar_t* **Py_GetExecPrefix** ()

Retorna el *exec-prefix* para los archivos instalados *dependientes* de la plataforma. Esto se deriva a través de una serie de reglas complicadas del nombre del programa establecido con `Py_SetProgramName()` y algunas variables de entorno; por ejemplo, si el nombre del programa es `'/usr/local/bin/python'`, el prefijo *exec* es `'/usr/local'`. La cadena de caracteres retornada apunta al almacenamiento estático; la persona que llama no debe modificar su valor. Esto corresponde a la variable **exec_prefix** en el archivo de nivel superior `Makefile` y el argumento `--exec-prefix` a la secuencia de comandos (*script*) **configure** en tiempo de compilación. El valor está disponible para el código de Python como `sys.exec_prefix`. Solo es útil en Unix.

Antecedentes: el prefijo *exec* difiere del prefijo cuando los archivos dependientes de la plataforma (como ejecutables y bibliotecas compartidas) se instalan en un árbol de directorios diferente. En una instalación típica, los archivos dependientes de la plataforma pueden instalarse en el subárbol `/usr/local/plat` mientras que la plataforma independiente puede instalarse en `/usr/local`.

En términos generales, una plataforma es una combinación de familias de hardware y software, por ejemplo, las máquinas Sparc que ejecutan el sistema operativo Solaris 2.x se consideran la misma plataforma, pero las máquinas Intel que ejecutan Solaris 2.x son otra plataforma, y las máquinas Intel que ejecutan Linux son otra plataforma más. Las diferentes revisiones importantes del mismo sistema operativo generalmente también forman plataformas diferentes. Los sistemas operativos que no son Unix son una historia diferente; Las estrategias de instalación en esos sistemas son tan diferentes que el prefijo y el prefijo *exec* no tienen sentido y se configuran en la cadena vacía. Tenga en cuenta que los archivos de bytecode compilados de Python son independientes de la plataforma (¡pero no independientes de la versión de Python con la que fueron compilados!).

Los administradores de sistemas sabrán cómo configurar los programas **mount** o **automount** para compartir `/usr/local` entre plataformas mientras que `/usr/local/plat` sea un sistema de archivos diferente para cada plataforma.

wchar_t* **Py_GetProgramFullPath** ()

Retorna el nombre completo del programa del ejecutable de Python; esto se calcula como un efecto secundario de derivar la ruta de búsqueda predeterminada del módulo del nombre del programa (establecido por `Py_SetProgramName()` arriba). La cadena de caracteres retornada apunta al almacenamiento estático; la persona que llama no debe modificar su valor. El valor está disponible para el código de Python como `sys.executable`.

wchar_t* **Py_GetPath** ()

Return the default module search path; this is computed from the program name (set by `Py_SetProgramName()` above) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is `:` on Unix and macOS, `;` on Windows. The returned string points into static storage; the caller should not modify its value. The list `sys.path` is initialized with this value on interpreter startup; it can be (and usually is) modified later to change the search path for loading modules.

void **Py_SetPath** (const wchar_t *)

Set the default module search path. If this function is called before `Py_Initialize()`, then `Py_GetPath()` won't attempt to compute a default search path but uses the one provided instead. This is useful if Python is embedded by an application that has full knowledge of the location of all modules. The path components should be separated by the platform dependent delimiter character, which is `:` on Unix and macOS, `;` on Windows.

Esto también hace que `sys.executable` se configure en la ruta completa del programa (consulte `Py_GetProgramFullPath()`) y para `sys.prefix` y `sys.exec_prefix` a estar vacío. Depende de la persona que llama modificarlos si es necesario después de llamar `Py_Initialize()`.

Use `Py_DecodeLocale()` para decodificar una cadena de bytes para obtener una cadena `wchar_*`.

El argumento de ruta se copia internamente, por lo que la persona que llama puede liberarlo después de que se complete la llamada.

Distinto en la versión 3.8: La ruta completa del programa ahora se usa para `sys.executable`, en lugar del nombre del programa.

const char* **Py_GetVersion** ()

Retorna la versión de este intérprete de Python. Esta es una cadena de caracteres que se parece a

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[GCC 4.2.3]"
```

The first word (up to the first space character) is the current Python version; the first characters are the major and minor version separated by a period. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.version`.

const char* **Py_GetPlatform** ()

Return the platform identifier for the current platform. On Unix, this is formed from the «official» name of the operating system, converted to lower case, followed by the major revision number; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is 'sunos5'. On macOS, it is 'darwin'. On Windows, it is 'win'. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.platform`.

const char* **Py_GetCopyright** ()

Retorna la cadena de caracteres de copyright oficial para la versión actual de Python, por ejemplo

```
'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'
```

La cadena de caracteres retornada apunta al almacenamiento estático; la persona que llama no debe modificar su valor. El valor está disponible para el código de Python como `sys.copyright`.

const char* **Py_GetCompiler** ()

Retorna una indicación del compilador utilizado para construir la versión actual de Python, entre corchetes, por ejemplo:

```
"[GCC 2.7.2.2]"
```

La cadena de caracteres retornada apunta al almacenamiento estático; la persona que llama no debe modificar su valor. El valor está disponible para el código Python como parte de la variable `sys.version`.

const char* **Py_GetBuildInfo** ()

Retorna información sobre el número de secuencia y la fecha y hora de compilación de la instancia actual de intérprete de Python, por ejemplo:

```
"#67, Aug 1 1997, 22:34:28"
```

La cadena de caracteres retornada apunta al almacenamiento estático; la persona que llama no debe modificar su valor. El valor está disponible para el código Python como parte de la variable `sys.version`.

void **PySys_SetArgvEx** (int argc, wchar_t **argv, int updatepath)

Establece `sys.argv` basado en `argc` y `argv`. Estos parámetros son similares a los pasados a la función del programa `main()` con la diferencia de que la primera entrada debe referirse al archivo de la secuencia de comandos (*script*) que se ejecutará en lugar del ejecutable que aloja el intérprete de Python. Si no se ejecuta una secuencia de comandos (*script*), la primera entrada en `argv` puede ser una cadena de caracteres vacía. Si esta función no puede inicializar `sys.argv`, una condición fatal se señala usando `Py_FatalError()`.

Si `updatepath` es cero, esto es todo lo que hace la función. Si `updatepath` no es cero, la función también modifica `sys.path` de acuerdo con el siguiente algoritmo:

- Si el nombre de una secuencia de comandos (*script*) existente se pasa en `argv[0]`, la ruta absoluta del directorio donde se encuentra el *script* se antepone a `sys.path`.
- De lo contrario (es decir, si *argc* es 0 o `argv[0]` no apunta a un nombre de archivo existente), una cadena de caracteres vacía se antepone a `sys.path`, que es lo mismo que anteponer el directorio de trabajo actual (`"."`).

Use `Py_DecodeLocale()` para decodificar una cadena de bytes para obtener una cadena `wchar_*`.

Nota: Se recomienda que las aplicaciones que incorporan el intérprete de Python para otros fines que no sean ejecutar una sola secuencia de comandos (*script*) pasen 0 como *updatepath* y actualicen `sys.path` si lo desean. Ver CVE-2008-5983 <<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-5983>> _.

En las versiones anteriores a 3.1.3, puede lograr el mismo efecto quitando manualmente el primer elemento (*poping*) `sys.path` después de haber llamado `PySys_SetArgv()`, por ejemplo usando

```
PyRun_SimpleString("import sys; sys.path.pop(0)\n");
```

Nuevo en la versión 3.1.3.

void **PySys_SetArgv** (int *argc*, wchar_t ***argv*)

Esta función funciona como `PySys_SetArgvEx()` con *updatepath* establecido en 1 a menos que el intérprete **python** se haya iniciado con la opción `-I`.

Use `Py_DecodeLocale()` para decodificar una cadena de bytes para obtener una cadena `wchar_*`.

Distinto en la versión 3.4: El valor *updatepath* depende de la opción `-I`.

void **Py_SetPythonHome** (const wchar_t **home*)

Establece el directorio «inicio» (*home*) predeterminado, es decir, la ubicación de las bibliotecas estándar de Python. Ver `PYTHONHOME` para el significado de la cadena de caracteres de argumento.

El argumento debe apuntar a una cadena de caracteres terminada en cero en el almacenamiento estático cuyo contenido no cambiará mientras dure la ejecución del programa. Ningún código en el intérprete de Python cambiará el contenido de este almacenamiento.

Use `Py_DecodeLocale()` para decodificar una cadena de bytes para obtener una cadena `wchar_*`.

w_char* **Py_GetPythonHome** ()

Retorna el «inicio» (*home*) predeterminado, es decir, el valor establecido por una llamada anterior a `Py_SetPythonHome()`, o el valor de la variable de entorno `PYTHONHOME` si está configurado.

9.5 Estado del hilo y el bloqueo global del intérprete

El intérprete de Python no es completamente seguro para hilos (*thread-safe*). Para admitir programas Python multiproceso, hay un bloqueo global, denominado *global interpreter lock* o *GIL*, que debe mantener el hilo actual antes de que pueda acceder de forma segura a los objetos Python. Sin el bloqueo, incluso las operaciones más simples podrían causar problemas en un programa de hilos múltiples: por ejemplo, cuando dos hilos incrementan simultáneamente el conteo de referencias del mismo objeto, el conteo de referencias podría terminar incrementándose solo una vez en lugar de dos veces.

Por lo tanto, existe la regla de que solo el hilo que ha adquirido *GIL* puede operar en objetos Python o llamar a funciones API Python/C. Para emular la concurrencia de ejecución, el intérprete regularmente intenta cambiar los hilos (ver `sys.setswitchinterval()`). El bloqueo también se libera para bloquear potencialmente las operaciones de E/S, como leer o escribir un archivo, para que otros hilos de Python puedan ejecutarse mientras tanto.

El intérprete de Python mantiene cierta información de contabilidad específica de hilos dentro de una estructura de datos llamada `PyThreadState`. También hay una variable global que apunta a la actual `PyThreadState`: se puede recuperar usando `PyThreadState_Get()`.

9.5.1 Liberando el GIL del código de extensión

La mayoría del código de extensión que manipula el *GIL* tiene la siguiente estructura simple

```
Save the thread state in a local variable.
Release the global interpreter lock.
... Do some blocking I/O operation ...
Reacquire the global interpreter lock.
Restore the thread state from the local variable.
```

Esto es tan común que existen un par de macros para simplificarlo:

```
Py_BEGIN_ALLOW_THREADS
... Do some blocking I/O operation ...
Py_END_ALLOW_THREADS
```

La macro `Py_BEGIN_ALLOW_THREADS` abre un nuevo bloque y declara una variable local oculta; la macro `Py_END_ALLOW_THREADS` cierra el bloque.

El bloque anterior se expande al siguiente código:

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(_save);
```

Así es como funcionan estas funciones: el bloqueo global del intérprete se usa para proteger el puntero al estado actual del hilo. Al liberar el bloqueo y guardar el estado del hilo, el puntero del estado del hilo actual debe recuperarse antes de que se libere el bloqueo (ya que otro hilo podría adquirir inmediatamente el bloqueo y almacenar su propio estado de hilo en la variable global). Por el contrario, al adquirir el bloqueo y restaurar el estado del hilo, el bloqueo debe adquirirse antes de almacenar el puntero del estado del hilo.

Nota: Llamar a las funciones de E/S del sistema es el caso de uso más común para liberar el GIL, pero también puede ser útil antes de llamar a cálculos de larga duración que no necesitan acceso a objetos de Python, como las funciones de compresión o criptográficas que operan sobre memorias intermedias. Por ejemplo, los módulos estándar `zlib` y `hashlib` liberan el GIL al comprimir o mezclar datos.

9.5.2 Hilos creados sin Python

Cuando se crean hilos utilizando las API dedicadas de Python (como el módulo `threading`), se les asocia automáticamente un estado del hilo y, por lo tanto, el código que se muestra arriba es correcto. Sin embargo, cuando los hilos se crean desde C (por ejemplo, por una biblioteca de terceros con su propia administración de hilos), no contienen el GIL, ni existe una estructura de estado de hilos para ellos.

Si necesita llamar al código Python desde estos subprocesos (a menudo esto será parte de una API de devolución de llamada proporcionada por la biblioteca de terceros mencionada anteriormente), primero debe registrar estos subprocesos con el intérprete creando una estructura de datos de estado del subproceso, luego adquiriendo el GIL, y finalmente almacenando su puntero de estado de hilo, antes de que pueda comenzar a usar la API Python/C. Cuando haya terminado,

debe restablecer el puntero del estado del hilo, liberar el GIL y finalmente liberar la estructura de datos del estado del hilo.

Las funciones `PyGILState_Ensure()` y `PyGILState_Release()` hacen todo lo anterior automáticamente. El idioma típico para llamar a Python desde un hilo C es:

```
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();

/* Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Release the thread. No Python API allowed beyond this point. */
PyGILState_Release(gstate);
```

Tenga en cuenta que las funciones `PyGILState_*`() suponen que solo hay un intérprete global (creado automáticamente por `Py_Initialize()`). Python admite la creación de intérpretes adicionales (usando `Py_NewInterpreter()`), pero la mezcla de múltiples intérpretes y la API `PyGILState_*`() no son compatibles.

9.5.3 Precauciones sobre `fork()`

Otra cosa importante a tener en cuenta sobre los hilos es su comportamiento frente a la llamada `C fork()`. En la mayoría de los sistemas con `fork()`, después de que un proceso se bifurca, solo existirá el hilo que emitió el *fork*. Esto tiene un impacto concreto tanto en cómo se deben manejar las cerraduras como en todo el estado almacenado en el tiempo de ejecución de CPython.

El hecho de que solo permanezca al hilo «actual» significa que ningún bloqueo retenido por otros hilos nunca se liberará. Python resuelve esto para `os.fork()` adquiriendo los bloqueos que usa internamente antes de la bifurcación (*fork*) y soltándolos después. Además, restablece cualquier lock-objects en el elemento secundario. Al extender o incrustar Python, no hay forma de informar a Python de bloqueos adicionales (que no sean Python) que deben adquirirse antes o restablecerse después de una bifurcación. Las instalaciones del sistema operativo como `pthread_atfork()` tendrían que usarse para lograr lo mismo. Además, al extender o incrustar Python, llamando `fork()` directamente en lugar de a través de `os.fork()` (y retornar o llamar a Python) puede resultar en un punto muerto (*deadlock*) por uno de los bloqueos internos de Python, sostenido por un hilo que no funciona después del *fork*. `PyOS_AfterFork_Child()` intenta restablecer los bloqueos necesarios, pero no siempre puede hacerlo.

El hecho de que todos los otros hilos desaparezcan también significa que el estado de ejecución de CPython debe limpiarse correctamente, lo que `os.fork()` lo hace. Esto significa finalizar todos los demás objetos `PyThreadState` que pertenecen al intérprete actual y todos los demás objetos `PyInterpreterState`. Debido a esto y a la naturaleza especial del *intérprete «principal»*, `fork()` solo debería llamarse en el hilo «principal» de ese intérprete, donde el CPython global el tiempo de ejecución se inicializó originalmente. La única excepción es si `exec()` se llamará inmediatamente después.

9.5.4 API de alto nivel

Estos son los tipos y funciones más utilizados al escribir código de extensión C o al incrustar el intérprete de Python:

PyInterpreterState

Esta estructura de datos representa el estado compartido por varios subprocesos cooperantes. Los hilos que pertenecen al mismo intérprete comparten la administración de su módulo y algunos otros elementos internos. No hay miembros públicos en esta estructura.

Los hilos que pertenecen a diferentes intérpretes inicialmente no comparten nada, excepto el estado del proceso como memoria disponible, descriptores de archivos abiertos y demás. El bloqueo global del intérprete también es compartido por todos los hilos, independientemente de a qué intérprete pertenezcan.

PyThreadState

Esta estructura de datos representa el estado de un solo hilo. El único miembro de datos públicos es `interp` (`PyInterpreterState *`), que apunta al estado del intérprete de este hilo.

void PyEval_InitThreads ()

Función deprecada que no hace nada.

En Python 3.6 y versiones anteriores, esta función creaba el GIL si no existía.

Distinto en la versión 3.9: La función ahora no hace nada.

Distinto en la versión 3.7: Esta función ahora es llamada por `Py_Initialize()`, por lo que ya no tiene que llamarla usted mismo.

Distinto en la versión 3.2: Esta función ya no se puede llamar antes de `Py_Initialize()`.

Deprecated since version 3.9, will be removed in version 3.11.

int PyEval_ThreadsInitialized ()

Retorna un valor distinto de cero si se ha llamado a `PyEval_InitThreads()`. Esta función se puede invocar sin mantener el GIL y, por lo tanto, se puede utilizar para evitar llamadas a la API de bloqueo cuando se ejecuta un solo hilo.

Distinto en la versión 3.7: El término *GIL* ahora se inicializa con `Py_Initialize()`.

Deprecated since version 3.9, will be removed in version 3.11.

PyThreadState* PyEval_SaveThread ()

Libere el bloqueo global del intérprete (si se ha creado) y restablezca el estado del hilo a NULL, retornando el estado del hilo anterior (que no es NULL). Si se ha creado el bloqueo, el hilo actual debe haberlo adquirido.

void PyEval_RestoreThread (PyThreadState *tstate)

Adquiera el bloqueo global del intérprete (si se ha creado) y establezca el estado del hilo en `tstate`, que no debe ser NULL. Si se ha creado el bloqueo, el hilo actual no debe haberlo adquirido, de lo contrario se produce un *deadlock*.

Nota: Llamar a esta función desde un hilo cuando finalice el tiempo de ejecución terminará el hilo, incluso si Python no creó el hilo. Puede usar `_Py_IsFinalizing()` o `sys.is_finalizing()` para verificar si el intérprete está en proceso de finalización antes de llamar a esta función para evitar una terminación no deseada.

PyThreadState* PyThreadState_Get ()

Retorna el estado actual del hilo. Se debe mantener el bloqueo global del intérprete. Cuando el estado actual del hilo es NULL, esto genera un error fatal (por lo que la persona que llama no necesita verificar NULL).

PyThreadState* PyThreadState_Swap (PyThreadState *tstate)

Cambia el estado del hilo actual con el estado del hilo dado por el argumento `tstate`, que puede ser NULL. El bloqueo global del intérprete debe mantenerse y no se libera.

Las siguientes funciones utilizan almacenamiento local de hilos y no son compatibles con subintérpretes:

PyGILState_STATE PyGILState_Ensure ()

Asegúrese de que el subproceso actual esté listo para llamar a la API de Python C, independientemente del estado actual de Python o del bloqueo global del intérprete. Esto se puede invocar tantas veces como lo desee un subproceso siempre que cada llamada coincida con una llamada a `PyGILState_Release()`. En general, se pueden usar otras API relacionadas con subprocesos entre `PyGILState_Ensure()` y `PyGILState_Release()` invoca siempre que el estado del subproceso se restablezca a su estado anterior antes del `Release()`. Por ejemplo, el uso normal de las macros `Py_BEGIN_ALLOW_THREADS` y `Py_END_ALLOW_THREADS` es aceptable.

El valor de retorno es un «identificador» opaco al estado del hilo cuando `PyGILState_Ensure()` fue llamado, y debe pasarse a `PyGILState_Release()` para asegurar que Python se deje en el mismo estado.

Aunque las llamadas recursivas están permitidas, estos identificadores *no* pueden compartirse; cada llamada única a `PyGILState_Ensure()` debe guardar el identificador para su llamada a `PyGILState_Release()`.

Cuando la función regrese, el hilo actual contendrá el GIL y podrá llamar a código arbitrario de Python. El fracaso es un error fatal.

Nota: Llamar a esta función desde un hilo cuando finalice el tiempo de ejecución terminará el hilo, incluso si Python no creó el hilo. Puede usar `_Py_IsFinalizing()` o `sys.is_finalizing()` para verificar si el intérprete está en proceso de finalización antes de llamar a esta función para evitar una terminación no deseada.

void **PyGILState_Release** (PyGILState_STATE)

Libera cualquier recurso previamente adquirido. Después de esta llamada, el estado de Python será el mismo que antes de la llamada correspondiente `PyGILState_Ensure()` (pero en general este estado será desconocido para la persona que llama, de ahí el uso de la API `GILState`).

Cada llamada a `PyGILState_Ensure()` debe coincidir con una llamada a `PyGILState_Release()` en el mismo hilo.

*PyThreadState** **PyGILState_GetThisThreadState** ()

Obtenga el estado actual del hilo para este hilo. Puede retornar `NULL` si no se ha utilizado la API `GILState` en el hilo actual. Tenga en cuenta que el subproceso principal siempre tiene dicho estado de subproceso, incluso si no se ha realizado una llamada de estado de subproceso automático en el subproceso principal. Esta es principalmente una función auxiliar y de diagnóstico.

int **PyGILState_Check** ()

Retorna 1 si el hilo actual mantiene el GIL y 0 de lo contrario. Esta función se puede llamar desde cualquier hilo en cualquier momento. Solo si se ha inicializado el hilo de Python y actualmente mantiene el GIL, retornará 1. Esta es principalmente una función auxiliar y de diagnóstico. Puede ser útil, por ejemplo, en contextos de devolución de llamada o funciones de asignación de memoria cuando saber que el GIL está bloqueado puede permitir que la persona que llama realice acciones confidenciales o se comporte de otra manera de manera diferente.

Nuevo en la versión 3.4.

Las siguientes macros se usan normalmente sin punto y coma final; busque, por ejemplo, el uso en la distribución fuente de Python.

Py_BEGIN_ALLOW_THREADS

Esta macro se expande a `{PyThreadState *_save; _save = PyEval_SaveThread();`. Tenga en cuenta que contiene una llave de apertura; debe coincidir con la siguiente macro `Py_END_ALLOW_THREADS`. Ver arriba para una discusión más detallada de esta macro.

Py_END_ALLOW_THREADS

Esta macro se expande a `PyEval_RestoreThread(_save); }`. Tenga en cuenta que contiene una llave de cierre; debe coincidir con una macro anterior `Py_BEGIN_ALLOW_THREADS`. Ver arriba para una discusión más detallada de esta macro.

Py_BLOCK_THREADS

Esta macro se expande a `PyEval_RestoreThread(_save);`; es equivalente a `Py_END_ALLOW_THREADS` sin la llave de cierre.

Py_UNBLOCK_THREADS

Esta macro se expande a `_save = PyEval_SaveThread();`; es equivalente a `Py_BEGIN_ALLOW_THREADS` sin la llave de apertura y la declaración de variable.

9.5.5 API de bajo nivel

Todas las siguientes funciones deben llamarse después de `Py_Initialize()`.

Distinto en la versión 3.7: `Py_Initialize()` ahora inicializa el *GIL*.

*PyInterpreterState** **PyInterpreterState_New** ()

Crea un nuevo objeto de estado de intérprete. No es necesario retener el bloqueo global del intérprete, pero se puede retener si es necesario para serializar llamadas a esta función.

Genera un evento de auditoría `python.PyInterpreterState_New` sin argumentos.

void **PyInterpreterState_Clear** (*PyInterpreterState* *interp)

Restablece toda la información en un objeto de estado de intérprete. Se debe mantener el bloqueo global del intérprete.

Lanza una eventos de auditoría `python.PyInterpreterState_Clear` sin argumentos.

void **PyInterpreterState_Delete** (*PyInterpreterState* *interp)

Destruye un objeto de estado de intérprete. No es necesario mantener el bloqueo global del intérprete. El estado del intérprete debe haberse restablecido con una llamada previa a `PyInterpreterState_Clear()`.

*PyThreadState** **PyThreadState_New** (*PyInterpreterState* *interp)

Crea un nuevo objeto de estado de hilo que pertenece al objeto de intérprete dado. No es necesario retener el bloqueo global del intérprete, pero se puede retener si es necesario para serializar llamadas a esta función.

void **PyThreadState_Clear** (*PyThreadState* *tstate)

Restablece toda la información en un objeto de estado de hilo. Se debe mantener el bloqueo global del intérprete.

Distinto en la versión 3.9: Esta función ahora llama a la retrollamada `PyThreadState.on_delete`. Anteriormente, eso sucedía en `PyThreadState_Delete()`.

void **PyThreadState_Delete** (*PyThreadState* *tstate)

Destruye un objeto de estado de hilo. No es necesario mantener el bloqueo global del intérprete. El estado del hilo debe haberse restablecido con una llamada previa a `PyThreadState_Clear()`.

void **PyThreadState_DeleteCurrent** (void)

Destruye un objeto de estado de hilo y suelta el bloqueo del intérprete global. Como `PyThreadState_Delete()`, no es necesario mantener el bloqueo del intérprete global. El estado del hilo debe haberse restablecido con una llamada anterior a `PyThreadState_Clear()`.

*PyFrameObject** **PyThreadState_GetFrame** (*PyThreadState* *tstate)

Obtiene el marco actual del estado del hilo de Python *tstate*.

Retorna una referencia sólida. Retorna NULL si no se está ejecutando ningún marco.

Vea también `PyEval_GetFrame()`.

tstate no debe ser NULL.

Nuevo en la versión 3.9.

uint64_t **PyThreadState_GetID** (*PyThreadState* *tstate)

Obtiene el identificador de estado de subproceso único del estado del hilo de Python *tstate*.

tstate no debe ser NULL.

Nuevo en la versión 3.9.

*PyInterpreterState** **PyThreadState_GetInterpreter** (*PyThreadState* *tstate)

Obtiene el intérprete del estado del hilo de Python *tstate*.

tstate no debe ser NULL.

Nuevo en la versión 3.9.

*PyInterpreterState** **PyInterpreterState_Get** (void)

Obtiene el intérprete actual.

Emite un error fatal si no hay un estado actual del hilo de Python o no hay un intérprete actual. No puede retornar NULL.

La persona que llama debe retener el GIL.

Nuevo en la versión 3.9.

int64_t **PyInterpreterState_GetID** (*PyInterpreterState* *interp)

Retorna la identificación única del intérprete. Si hubo algún error al hacerlo, entonces se retorna -1 y se establece un error.

La persona que llama debe retener el GIL.

Nuevo en la versión 3.7.

*PyObject** **PyInterpreterState_GetDict** (*PyInterpreterState* *interp)

Retorna un diccionario en el que se pueden almacenar datos específicos del intérprete. Si esta función retorna NULL, no se ha producido ninguna excepción y la persona que llama debe suponer que no hay disponible una instrucción específica del intérprete.

Esto no reemplaza a *PyModule_GetState()*, que las extensiones deben usar para almacenar información de estado específica del intérprete.

Nuevo en la versión 3.8.

*PyObject** (***_PyFrameEvalFunction**) (*PyThreadState* *tstate, *PyFrameObject* *frame, int throwflag)

Tipo de función de evaluación de marcos.

El parámetro *throwflag* es usado por el método de generadores *throw()*: si no es cero, maneja la excepción actual.

Distinto en la versión 3.9: La función ahora recibe un parámetro *tstate*.

_PyFrameEvalFunction **_PyInterpreterState_GetEvalFrameFunc** (*PyInterpreterState* *interp)

Obtiene la función de evaluación de marcos.

Consulte [PEP 523](#) «Adición de una API de evaluación de marcos a CPython».

Nuevo en la versión 3.9.

void **_PyInterpreterState_SetEvalFrameFunc** (*PyInterpreterState* *interp, *_PyFrameEvalFunction* eval_frame)

Configura la función de evaluación del marco.

Consulte [PEP 523](#) «Adición de una API de evaluación de marcos a CPython».

Nuevo en la versión 3.9.

*PyObject** **PyThreadState_GetDict** ()

Return value: Borrowed reference. Retorna un diccionario en el que las extensiones pueden almacenar información de estado específica del hilo. Cada extensión debe usar una clave única para almacenar el estado en el diccionario. Está bien llamar a esta función cuando no hay un estado del hilo actual disponible. Si esta función retorna NULL, no se ha producido ninguna excepción y la persona que llama debe asumir que no hay disponible ningún estado del hilo actual.

int **PyThreadState_SetAsyncExc** (unsigned long id, *PyObject* *exc)

Asincrónicamente lanza una excepción en un hilo. El argumento *id* es el id del hilo de destino; *exc* es el objeto de excepción que se debe generar. Esta función no roba ninguna referencia a *exc*. Para evitar el uso indebido ingenuo, debe escribir su propia extensión C para llamar a esto. Debe llamarse con el GIL retenido. Retorna el número de estados de hilo modificados; normalmente es uno, pero será cero si no se encuentra la identificación del hilo. Si *exc* es NULL, se borra la excepción pendiente (si existe) para el hilo. Esto no lanza excepciones.

Distinto en la versión 3.7: El tipo del parámetro *id* cambia de *long* a *unsigned long*.

void **PyEval_AcquireThread** (*PyThreadState* *tstate)

Adquiere el bloqueo global del intérprete y establece el estado actual del hilo en *tstate*, que no debe ser NULL. El bloqueo debe haber sido creado anteriormente. Si este hilo ya tiene el bloqueo, se produce un deadlock.

Nota: Llamar a esta función desde un hilo cuando finalice el tiempo de ejecución terminará el hilo, incluso si Python no creó el hilo. Puede usar `_Py_IsFinalizing()` o `sys.is_finalizing()` para verificar si el intérprete está en proceso de finalización antes de llamar a esta función para evitar una terminación no deseada.

Distinto en la versión 3.8: Actualiza para ser coherente con `PyEval_RestoreThread()`, `Py_END_ALLOW_THREADS()`, y `PyGILState_Ensure()`, y termina el hilo actual si se llama mientras el intérprete está finalizando.

`PyEval_RestoreThread()` es una función de nivel superior que siempre está disponible (incluso cuando los subprocesos no se han inicializado).

void **PyEval_ReleaseThread** (*PyThreadState* *tstate)

Restablece el estado actual del hilo a NULL y libera el bloqueo global del intérprete. El bloqueo debe haberse creado antes y debe estar retenido por el hilo actual. El argumento *tstate*, que no debe ser NULL, solo se usa para verificar que representa el estado actual del hilo — si no lo es, se informa un error fatal.

`PyEval_SaveThread()` es una función de nivel superior que siempre está disponible (incluso cuando los hilos no se han inicializado).

void **PyEval_AcquireLock** ()

Adquiera el bloqueo global de intérprete. El bloqueo debe haber sido creado anteriormente. Si este hilo ya tiene el bloqueo, se produce un *deadlock*.

Obsoleto desde la versión 3.2: Esta función no actualiza el estado actual del hilo. Utilice `PyEval_RestoreThread()` o `PyEval_AcquireThread()` en su lugar.

Nota: Llamar a esta función desde un hilo cuando finalice el tiempo de ejecución terminará el hilo, incluso si Python no creó el hilo. Puede usar `_Py_IsFinalizing()` o `sys.is_finalizing()` para verificar si el intérprete está en proceso de finalización antes de llamar a esta función para evitar una terminación no deseada.

Distinto en la versión 3.8: Actualiza para ser coherente con `PyEval_RestoreThread()`, `Py_END_ALLOW_THREADS()`, y `PyGILState_Ensure()`, y termina el hilo actual si se llama mientras el intérprete está finalizando.

void **PyEval_ReleaseLock** ()

Libere el bloqueo global del intérprete. El bloqueo debe haber sido creado anteriormente.

Obsoleto desde la versión 3.2: Esta función no actualiza el estado actual del hilo. Utilice `PyEval_SaveThread()` o `PyEval_ReleaseThread()` en su lugar.

9.6 Soporte de subintérprete

Si bien en la mayoría de los usos, solo incrustará un solo intérprete de Python, hay casos en los que necesita crear varios intérpretes independientes en el mismo proceso y tal vez incluso en el mismo hilo. Los subintérpretes le permiten hacer eso.

El intérprete «principal» es el primero creado cuando se inicializa el tiempo de ejecución. Suele ser el único intérprete de Python en un proceso. A diferencia de los subintérpretes, el intérprete principal tiene responsabilidades globales de proceso únicas, como el manejo de señales. También es responsable de la ejecución durante la inicialización del

tiempo de ejecución y generalmente es el intérprete activo durante la finalización del tiempo de ejecución. La función `PyInterpreterState_Main()` retorna un puntero a su estado.

Puede cambiar entre subinterpretes utilizando la función `PyThreadState_Swap()`. Puede crearlos y destruirlos utilizando las siguientes funciones:

`PyThreadState* Py_NewInterpreter()`

Crea un nuevo subintérprete. Este es un entorno (casi) totalmente separado para la ejecución de código Python. En particular, el nuevo intérprete tiene versiones separadas e independientes de todos los módulos importados, incluidos los módulos fundamentales `builtins`, `__main__` y `sys`. La tabla de módulos cargados (`sys.modules`) y la ruta de búsqueda del módulo (`sys.path`) también están separados. El nuevo entorno no tiene variable `sys.argv`. Tiene nuevos objetos de archivo de flujo de E/S estándar `sys.stdin`, `sys.stdout` y `sys.stderr` (sin embargo, estos se refieren a los mismos descriptores de archivo subyacentes).

El valor de retorno apunta al primer estado del hilo creado en el nuevo subintérprete. Este estado de hilo se realiza en el estado de hilo actual. Tenga en cuenta que no se crea ningún hilo real; vea la discusión de los estados del hilo a continuación. Si la creación del nuevo intérprete no tiene éxito, se retorna `NULL`; no se establece ninguna excepción, ya que el estado de excepción se almacena en el estado actual del hilo y es posible que no haya un estado actual del hilo. (Al igual que todas las otras funciones de Python/C API, el bloqueo global del intérprete debe mantenerse antes de llamar a esta función y aún se mantiene cuando regresa; sin embargo, a diferencia de la mayoría de las otras funciones de Python/C API, no es necesario que haya un estado del hilo actual en entrada.)

Los módulos de extensión se comparten entre (sub) intérpretes de la siguiente manera:

- Para módulos que usan inicialización multifase, por ejemplo `PyModule_FromDefAndSpec()`, se crea e inicializa un objeto de módulo separado para cada intérprete. Solo las variables estáticas y globales de nivel C se comparten entre estos objetos de módulo.
- Para módulos que utilizan inicialización monofásica, por ejemplo `PyModule_Create()`, la primera vez que se importa una extensión en particular, se inicializa normalmente y una copia (superficial) del diccionario de su módulo se guarda. Cuando otro (sub) intérprete importa la misma extensión, se inicializa un nuevo módulo y se llena con el contenido de esta copia; no se llama a la función `init` de la extensión. Los objetos en el diccionario del módulo terminan compartidos entre (sub) intérpretes, lo que puede causar un comportamiento no deseado (ver Errores y advertencias (*Bugs and caveats*) a continuación).

Tenga en cuenta que esto es diferente de lo que sucede cuando se importa una extensión después de que el intérprete se haya reiniciado por completo llamando a `Py_FinalizeEx()` y `Py_Initialize()`; en ese caso, la función `initmodule` de la extensión es llamada nuevamente. Al igual que con la inicialización de múltiples fases, esto significa que solo se comparten variables estáticas y globales de nivel C entre estos módulos.

`void Py_EndInterpreter(PyThreadState *tstate)`

Destruye el (sub) intérprete representado por el estado del hilo dado. El estado del hilo dado debe ser el estado del hilo actual. Vea la discusión de los estados del hilo a continuación. Cuando la llamada regresa, el estado actual del hilo es `NULL`. Todos los estados de hilo asociados con este intérprete se destruyen. (El bloqueo global del intérprete debe mantenerse antes de llamar a esta función y aún se mantiene cuando vuelve). `Py_FinalizeEx()` destruirá todos los subinterpretes que no se hayan destruido explícitamente en ese punto.

9.6.1 Errores y advertencias

Debido a que los subinterpretes (y el intérprete principal) son parte del mismo proceso, el aislamiento entre ellos no es perfecto — por ejemplo, usando operaciones de archivos de bajo nivel como `os.close()` pueden (accidentalmente o maliciosamente) afectar los archivos abiertos del otro. Debido a la forma en que las extensiones se comparten entre (sub) intérpretes, algunas extensiones pueden no funcionar correctamente; esto es especialmente probable cuando se utiliza la inicialización monofásica o las variables globales (estáticas). Es posible insertar objetos creados en un subintérprete en un espacio de nombres de otro (sub) intérprete; Esto debe evitarse si es posible.

Se debe tener especial cuidado para evitar compartir funciones, métodos, instancias o clases definidas por el usuario entre los subinterpretes, ya que las operaciones de importación ejecutadas por dichos objetos pueden afectar el diccionario (sub-) intérprete incorrecto de los módulos cargados. Es igualmente importante evitar compartir objetos desde los que se pueda acceder a lo anterior.

También tenga en cuenta que la combinación de esta funcionalidad con `PyGILState_*()` API es delicada, porque estas API suponen una biyección entre los estados de hilo de Python e hilos a nivel del sistema operativo, una suposición rota por la presencia de subinterpretes. Se recomienda encarecidamente que no cambie los subinterpretes entre un par de llamadas coincidentes `PyGILState_Ensure()` y `PyGILState_Release()`. Además, las extensiones (como `ctypes`) que usan estas API para permitir la llamada de código Python desde hilos no creados por Python probablemente se rompan cuando se usan subinterpretes.

9.7 Notificaciones asincrónicas

Se proporciona un mecanismo para hacer notificaciones asincrónicas al hilo principal del intérprete. Estas notificaciones toman la forma de un puntero de función y un argumento de puntero nulo.

int **Py_AddPendingCall** (int (**func*)(void *), void **arg*)

Programa una función para que se llame desde el hilo principal del intérprete. En caso de éxito, se retorna 0 y se pone en cola *func* para ser llamado en el hilo principal. En caso de fallo, se retorna -1 sin establecer ninguna excepción.

Cuando se puso en cola con éxito, *func* será *eventualmente* invocado desde el hilo principal del intérprete con el argumento *arg*. Se llamará de forma asincrónica con respecto al código Python que se ejecuta normalmente, pero con ambas condiciones cumplidas:

- en un límite *bytecode*;
- con el hilo principal que contiene el *global interpreter lock* (*func*, por lo tanto, puede usar la API C completa).

func debe retornar 0 en caso de éxito o -1 en caso de error con una excepción establecida. *func* no se interrumpirá para realizar otra notificación asíncrona de forma recursiva, pero aún se puede interrumpir para cambiar hilos si se libera el bloqueo global del intérprete.

Esta función no necesita un estado de hilo actual para ejecutarse y no necesita el bloqueo global del intérprete.

Para llamar a esta función en un subintérprete, quien llama debe mantener el GIL. De lo contrario, la función *func* se puede programar para que se llame desde el intérprete incorrecto.

Advertencia: Esta es una función de bajo nivel, solo útil para casos muy especiales. No hay garantía de que *func* se llame lo más rápido posible. Si el hilo principal está ocupado ejecutando una llamada al sistema, no se llamará *func* antes de que vuelva la llamada del sistema. Esta función generalmente **no** es adecuada para llamar a código Python desde hilos C arbitrarios. En su lugar, use *PyGILState API*.

Distinto en la versión 3.9: Si esta función se llama en un subintérprete, la función *func* ahora está programada para ser llamada desde el subintérprete, en lugar de ser llamada desde el intérprete principal. Cada subintérprete ahora

tiene su propia lista de llamadas programadas.

Nuevo en la versión 3.1.

9.8 Perfilado y Rastreo

El intérprete de Python proporciona soporte de bajo nivel para adjuntar funciones de creación de perfiles y seguimiento de ejecución. Estos se utilizan para herramientas de análisis de perfiles, depuración y cobertura.

Esta interfaz C permite que el código de perfilado o rastreo evite la sobrecarga de llamar a través de objetos invocables a nivel de Python, haciendo una llamada directa a la función C en su lugar. Los atributos esenciales de la instalación no han cambiado; la interfaz permite instalar funciones de rastreo por hilos, y los eventos básicos informados a la función de rastreo son los mismos que se informaron a las funciones de rastreo a nivel de Python en versiones anteriores.

int (***Py_tracefunc**) (*PyObject *obj*, *PyFrameObject *frame*, int *what*, *PyObject *arg*)

El tipo de la función de rastreo registrada usando `PyEval_SetProfile()` y `PyEval_SetTrace()`. El primer parámetro es el objeto pasado a la función de registro como *obj*, *frame* es el objeto de marco al que pertenece el evento, *what* es una de las constantes `PyTrace_CALL`, `PyTrace_EXCEPTION`, `PyTrace_LINE`, `PyTrace_RETURN`, `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION`, `PyTrace_C_RETURN`, o `PyTrace_OPCODE`, y *arg* depende de el valor de *what*:

Valor de <i>what</i>	Significado de <i>arg</i>
<code>PyTrace_CALL</code>	Siempre <code>Py_None</code> .
<code>PyTrace_EXCEPTION</code>	Información de excepción retornada por <code>sys.exc_info()</code> .
<code>PyTrace_LINE</code>	Siempre <code>Py_None</code> .
<code>PyTrace_RETURN</code>	Valor retornado al que llama, o NULL si es causado por una excepción.
<code>PyTrace_C_CALL</code>	Objeto función que se llaman.
<code>PyTrace_C_EXCEPTION</code>	Objeto función que se llaman.
<code>PyTrace_C_RETURN</code>	Objeto función que se llaman.
<code>PyTrace_OPCODE</code>	Siempre <code>Py_None</code> .

int **PyTrace_CALL**

El valor del parámetro *what* para una función `Py_tracefunc` cuando se informa una nueva llamada a una función o método, o una nueva entrada en un generador. Tenga en cuenta que la creación del iterador para una función de generador no se informa ya que no hay transferencia de control al código de bytes de Python en la marco correspondiente.

int **PyTrace_EXCEPTION**

El valor del parámetro *what* para una función `Py_tracefunc` cuando se ha producido una excepción. La función de devolución de llamada se llama con este valor para *what* cuando después de que se procese cualquier bytecode, después de lo cual la excepción se establece dentro del marco que se está ejecutando. El efecto de esto es que a medida que la propagación de la excepción hace que la pila de Python se desenrolle, el retorno de llamada se llama al retornar a cada marco a medida que se propaga la excepción. Solo las funciones de rastreo reciben estos eventos; el perfilador (*profiler*) no los necesita.

int **PyTrace_LINE**

El valor pasado como parámetro *what* a una función `Py_tracefunc` (pero no una función de creación de perfiles) cuando se informa un evento de número de línea. Puede deshabilitarse para un marco configurando `f_trace_lines` en 0 en ese marco.

int **PyTrace_RETURN**

El valor para el parámetro *what* para `Py_tracefunc` funciona cuando una llamada está por regresar.

int **PyTrace_C_CALL**

El valor del parámetro *what* para `Py_tracefunc` funciona cuando una función C está a punto de ser invocada.

int **PyTrace_C_EXCEPTION**

El valor del parámetro *what* para funciones *Py_tracefunc* cuando una función C ha lanzado una excepción.

int **PyTrace_C_RETURN**

El valor del parámetro *what* para *Py_tracefunc* funciona cuando una función C ha retornado.

int **PyTrace_OPCODE**

El valor del parámetro *what* para funciones *Py_tracefunc* (pero no funciones de creación de perfiles) cuando un nuevo código de operación está a punto de ejecutarse. Este evento no se emite de forma predeterminada: debe solicitarse explícitamente estableciendo *f_trace_opcodes* en 1 en el marco.

void **PyEval_SetProfile** (*Py_tracefunc func*, *PyObject *obj*)

Establece la función del generador de perfiles en *func*. El parámetro *obj* se pasa a la función como su primer parámetro, y puede ser cualquier objeto de Python o NULL. Si la función de perfilado necesita mantener el estado, el uso de un valor diferente para *obj* para cada hilo proporciona un lugar conveniente y seguro para guardarlo. Se llama a la función de perfilado para todos los eventos supervisados, excepto *PyTrace_LINE* *PyTrace_OPCODE* y *PyTrace_EXCEPTION*.

La persona que llama debe mantener el *GIL*.

void **PyEval_SetTrace** (*Py_tracefunc func*, *PyObject *obj*)

Establece la función de rastreo en *func*. Esto es similar a *PyEval_SetProfile()*, excepto que la función de rastreo recibe eventos de número de línea y eventos por código de operación, pero no recibe ningún evento relacionado con los objetos de la función C. Cualquier función de rastreo registrada con *PyEval_SetTrace()* no recibirá *PyTrace_C_CALL*, *PyTrace_C_EXCEPTION* o *PyTrace_C_RETURN* como valor para el parámetro *what*.

La persona que llama debe mantener el *GIL*.

9.9 Soporte avanzado del depurador

Estas funciones solo están destinadas a ser utilizadas por herramientas de depuración avanzadas.

*PyInterpreterState** **PyInterpreterState_Head** ()

Retorna el objeto de estado del intérprete al principio de la lista de todos esos objetos.

*PyInterpreterState** **PyInterpreterState_Main** ()

Retorna el objeto de estado del intérprete principal.

*PyInterpreterState** **PyInterpreterState_Next** (*PyInterpreterState *interp*)

Retorna el siguiente objeto de estado de intérprete después de *interp* de la lista de todos esos objetos.

*PyThreadState** **PyInterpreterState_ThreadHead** (*PyInterpreterState *interp*)

Retorna el puntero al primer objeto *PyThreadState* en la lista de hilos asociados con el intérprete *interp*.

*PyThreadState** **PyThreadState_Next** (*PyThreadState *tstate*)

Retorna el siguiente objeto de estado del hilo después de *tstate* de la lista de todos los objetos que pertenecen al mismo objeto *PyInterpreterState*.

9.10 Soporte de almacenamiento local de hilo

El intérprete de Python proporciona soporte de bajo nivel para el almacenamiento local de hilos (TLS) que envuelve la implementación de TLS nativa subyacente para admitir la API de almacenamiento local de hilos de nivel Python (`threading.local`). Las API de nivel CPython C son similares a las ofrecidas por pthreads y Windows: use una clave de hilo y funciones para asociar un valor de `void*` por hilo.

El GIL *no* necesita ser retenido al llamar a estas funciones; proporcionan su propio bloqueo.

Tenga en cuenta que `Python.h` no incluye la declaración de las API de TLS, debe incluir `pthread.h` para usar el almacenamiento local de hilos.

Nota: Ninguna de estas funciones API maneja la administración de memoria en nombre de los valores `void*`. Debe asignarlos y desasignarlos usted mismo. Si los valores `void*` son `PyObject*`, estas funciones tampoco realizan operaciones de conteo de referencias en ellos.

9.10.1 API de almacenamiento específico de hilo (TSS, *Thread Specific Storage*)

La API de TSS se introduce para reemplazar el uso de la API TLS existente dentro del intérprete de CPython. Esta API utiliza un nuevo tipo `Py_tss_t` en lugar de `int` para representar las claves del hilo.

Nuevo en la versión 3.7.

Ver también:

«Una nueva C-API para *Thread-Local Storage* en CPython» ([PEP 539](#))

`Py_tss_t`

Esta estructura de datos representa el estado de una clave del hilo, cuya definición puede depender de la implementación de TLS subyacente, y tiene un campo interno que representa el estado de inicialización de la clave. No hay miembros públicos en esta estructura.

Cuando `Py_LIMITED_API` no está definido, la asignación estática de este tipo por `Py_tss_NEEDS_INIT` está permitida.

`Py_tss_NEEDS_INIT`

Esta macro se expande al inicializador para variables `Py_tss_t`. Tenga en cuenta que esta macro no se definirá con `Py_LIMITED_API`.

Asignación dinámica

Asignación dinámica de `Py_tss_t`, requerida en los módulos de extensión construidos con `Py_LIMITED_API`, donde la asignación estática de este tipo no es posible debido a que su implementación es opaca en el momento de la compilación.

`Py_tss_t*` `PyThread_tss_alloc()`

Retorna un valor que es el mismo estado que un valor inicializado con `Py_tss_NEEDS_INIT`, o `NULL` en caso de falla de asignación dinámica.

`void` `PyThread_tss_free(Py_tss_t *key)`

Libera la clave `key` asignada por `PyThread_tss_alloc()`, después de llamar por primera vez `PyThread_tss_delete()` para asegurarse de que los hilos locales asociados no hayan sido asignados. Esto es un *no-op* si el argumento `key` es `NULL`.

Nota: A freed key becomes a dangling pointer. You should reset the key to `NULL`.

Métodos

El parámetro *key* de estas funciones no debe ser NULL. Además, los comportamientos de `PyThread_tss_set()` y `PyThread_tss_get()` no están definidos si el `Py_tss_t` dado no ha sido inicializado por `PyThread_tss_create()`.

int **PyThread_tss_is_created** (`Py_tss_t *key`)

Retorna un valor distinto de cero si `Py_tss_t` ha sido inicializado por `PyThread_tss_create()`.

int **PyThread_tss_create** (`Py_tss_t *key`)

Retorna un valor cero en la inicialización exitosa de una clave TSS. El comportamiento no está definido si el valor señalado por el argumento *key* no se inicializa con `Py_tss_NEEDS_INIT`. Esta función se puede invocar repetidamente en la misma tecla: llamarla a una tecla ya inicializada es un *no-op* e inmediatamente retorna el éxito.

void **PyThread_tss_delete** (`Py_tss_t *key`)

Destruye una clave TSS para olvidar los valores asociados con la clave en todos los hilos y cambie el estado de inicialización de la clave a no inicializado. Una clave destruida se puede inicializar nuevamente mediante `PyThread_tss_create()`. Esta función se puede invocar repetidamente en la misma llave; llamarla en una llave ya destruida es un *no-op*.

int **PyThread_tss_set** (`Py_tss_t *key`, void **value*)

Retorna un valor cero para indicar la asociación exitosa de un valor a void* con una clave TSS en el hilo actual. Cada hilo tiene un mapeo distinto de la clave a un valor void*.

void* **PyThread_tss_get** (`Py_tss_t *key`)

Retorna el valor void* asociado con una clave TSS en el hilo actual. Esto retorna NULL si no hay ningún valor asociado con la clave en el hilo actual.

9.10.2 API de almacenamiento local de hilos (TLS, *Thread Local Storage*)

Obsoleto desde la versión 3.7: Esta API es reemplazada por *API de Almacenamiento Específico de Hilos (TSS, por sus significado en inglés *Thread Specific Storage*)*.

Nota: Esta versión de la API no es compatible con plataformas donde la clave TLS nativa se define de una manera que no se puede transmitir de forma segura a int. En tales plataformas, `PyThread_create_key()` regresará inmediatamente con un estado de falla, y las otras funciones TLS serán no operativas en tales plataformas.

Debido al problema de compatibilidad mencionado anteriormente, esta versión de la API no debe usarse en código nuevo.

int **PyThread_create_key** ()

void **PyThread_delete_key** (int *key*)

int **PyThread_set_key_value** (int *key*, void **value*)

void* **PyThread_get_key_value** (int *key*)

void **PyThread_delete_key_value** (int *key*)

void **PyThread_ReInitTLS** ()

Configuración de inicialización de Python

Nuevo en la versión 3.8.

Estructuras:

- *PyConfig*
- *PyPreConfig*
- *PyStatus*
- *PyWideStringList*

Funciones:

- *PyConfig_Clear()*
- *PyConfig_InitIsolatedConfig()*
- *PyConfig_InitPythonConfig()*
- *PyConfig_Read()*
- *PyConfig_SetArgv()*
- *PyConfig_SetBytesArgv()*
- *PyConfig_SetBytesString()*
- *PyConfig_SetString()*
- *PyConfig_SetWideStringList()*
- *PyPreConfig_InitIsolatedConfig()*
- *PyPreConfig_InitPythonConfig()*
- *PyStatus_Error()*
- *PyStatus_Error()*
- *PyStatus_Error()*
- *PyStatus_IsError()*

- `PyStatus_IsExit()`
- `PyStatus_NoMemory()`
- `PyStatus_Ok()`
- `PyWideStringList_Append()`
- `PyWideStringList_Insert()`
- `Py_ExitStatusException()`
- `Py_InitializeFromConfig()`
- `Py_PreInitialize()`
- `Py_PreInitializeFromArgs()`
- `Py_PreInitializeFromBytesArgs()`
- `Py_RunMain()`
- `Py_GetArgcArgv()`

La preconfiguración (tipo `PyPreConfig`) se almacena en `_PyRuntime.preconfig` y la configuración (tipo `PyConfig`) se almacena en `PyInterpreterState.config`.

Consulte también *Inicialización, finalización y subprocessos*.

Ver también:

PEP 587 «Configuración de inicialización de Python».

10.1 PyWideStringList

PyWideStringList

Lista de cadenas de caracteres `wchar_t*`.

Si *length* no es cero, *items* no deben ser `NULL` y todas las cadenas de caracteres deben ser no `NULL`.

Métodos:

PyStatus **PyWideStringList_Append** (*PyWideStringList* *list, const `wchar_t` *item)

Agregar *item* a *list*.

Python debe estar preinicializado para llamar a esta función.

PyStatus **PyWideStringList_Insert** (*PyWideStringList* *list, *Py_ssize_t* index, const `wchar_t` *item)

Inserta *item* en *list* en *index*.

Si *index* es mayor o igual que el largo de *list*, agrega *item* a *list*.

index debe ser mayor o igual que 0.

Python debe estar preinicializado para llamar a esta función.

Campos de estructura:

Py_ssize_t **length**

Longitud de la lista.

`wchar_t**` **items**

Elementos de la lista.

10.2 PyStatus

PyStatus

Estructura para almacenar el estado de una función de inicialización: éxito, error o salida.

Para un error, puede almacenar el nombre de la función C que creó el error.

Campos de estructura:

int **exitcode**

Código de salida El argumento pasó a `exit()`.

const char ***err_msg**

Mensaje de error.

const char ***func**

El nombre de la función que creó un error puede ser NULL.

Funciones para crear un estado:

PyStatus **PyStatus_Ok** (void)

Éxito.

PyStatus **PyStatus_Error** (const char **err_msg*)

Error de inicialización con un mensaje.

PyStatus **PyStatus_NoMemory** (void)

Error de asignación de memoria (sin memoria).

PyStatus **PyStatus_Exit** (int *exitcode*)

Salida de Python con el código de salida especificado.

Funciones para manejar un estado:

int **PyStatus_Exception** (*PyStatus status*)

¿Es el estado un error o una salida? Si es verdadero, la excepción debe ser manejada; por ejemplo llamando a `Py_ExitStatusException()`.

int **PyStatus_IsError** (*PyStatus status*)

¿Es el resultado un error?

int **PyStatus_IsExit** (*PyStatus status*)

¿El resultado es una salida?

void **Py_ExitStatusException** (*PyStatus status*)

Llama a `exit(exitcode)` si *status* es una salida. Imprime el mensaje de error y sale con un código de salida distinto de cero si *status* es un error. Solo se debe llamar si `PyStatus_Exception(status)` no es cero.

Nota: Internamente, Python usa macros que establecen `PyStatus.func`, mientras que las funciones para crear un estado establecen `func` en NULL.

Ejemplo:

```
PyStatus alloc(void **ptr, size_t size)
{
    *ptr = PyMem_RawMalloc(size);
    if (*ptr == NULL) {
        return PyStatus_NoMemory();
    }
}
```

(continué en la próxima página)

(proviene de la página anterior)

```

    return PyStatus_Ok();
}

int main(int argc, char **argv)
{
    void *ptr;
    PyStatus status = alloc(&ptr, 16);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
    PyMem_Free(ptr);
    return 0;
}

```

10.3 PyPreConfig

PyPreConfig

Estructura utilizada para preinicializar Python:

- Establece el asignador de memoria de Python
- Configure el entorno local LC_CTYPE
- Establece el modo UTF-8

Función para inicializar una preconfiguración:

void **PyPreConfig_InitPythonConfig** (*PyPreConfig *preconfig*)

Inicializa la preconfiguración con *Configuración de Python*.

void **PyPreConfig_InitIsolatedConfig** (*PyPreConfig *preconfig*)

Inicializa la preconfiguración con *Configuración aislada*.

Campos de estructura:

int **allocator**

Nombre del asignador de memoria:

- PYMEM_ALLOCATOR_NOT_SET (0): no cambie los asignadores de memoria (use los valores predeterminados)
- PYMEM_ALLOCATOR_DEFAULT (1): asignadores de memoria predeterminados
- PYMEM_ALLOCATOR_DEBUG (2): asignadores de memoria predeterminados con ganchos de depuración
- PYMEM_ALLOCATOR_MALLOC (3): fuerza el uso de `malloc()`
- PYMEM_ALLOCATOR_MALLOC_DEBUG (4): fuerza el uso de `malloc()` con ganchos de depuración
- PYMEM_ALLOCATOR_PYMALLOC (5): *Python pymalloc memory allocator*
- PYMEM_ALLOCATOR_PYMALLOC_DEBUG (6): *Python pymalloc memory allocator* con ganchos de depuración

PYMEM_ALLOCATOR_PYMALLOC y PYMEM_ALLOCATOR_PYMALLOC_DEBUG no son compatibles si Python está configurado con `--with-pymalloc`

Ver *Administración de memorias*.

int `configure_locale`
 ¿Establece la configuración regional LC_CTYPE en la configuración regional preferida por el usuario? Si es igual a 0, establece `coerce_c_locale` y `coerce_c_locale_warn` en 0.

int `coerce_c_locale`
 Si es igual a 2, coaccione la configuración regional C; si es igual a 1, lea la configuración regional LC_CTYPE para decidir si debe ser forzado.

int `coerce_c_locale_warn`
 Si no es cero, emita una advertencia si la configuración regional C está coaccionada.

int `dev_mode`
 Ver `PyConfig.dev_mode`.

int `isolated`
 Ver `PyConfig.isolated`.

int `legacy_windows_fs_encoding` (*Windows only*)
 Si no es cero, desactive el modo UTF-8, configure la codificación del sistema de archivos Python en `mbscs`, configure el controlador de errores del sistema de archivos en `replace`.

 Solo disponible en Windows. La macro `#ifdef MS_WINDOWS` se puede usar para el código específico de Windows.

int `parse_argv`
 Si no es cero, `Py_PreInitializeFromArgs()` y `Py_PreInitializeFromBytesArgs()` analizan su argumento `argv` de la misma manera que Python analiza los argumentos de la línea de comandos: ver Argumentos de línea de comandos.

int `use_environment`
 Ver `PyConfig.use_environment`.

int `utf8_mode`
 Si no es cero, habilita el modo UTF-8.

10.4 Preinicialización con PyPreConfig

Funciones para preinicializar Python:

***PyStatus* `Py_PreInitialize`** (*const* *PyPreConfig* **preconfig*)
 Preinicializa Python desde la preconfiguración *preconfig*.

***PyStatus* `Py_PreInitializeFromBytesArgs`** (*const* *PyPreConfig* **preconfig*, *int* *argc*, *char* * *const* **argv*)
 Preinicializa Python desde la preconfiguración *preconfig* y argumentos de línea de comando (cadenas de caracteres de bytes).

***PyStatus* `Py_PreInitializeFromArgs`** (*const* *PyPreConfig* **preconfig*, *int* *argc*, *wchar_t* * *const* **argv*)
 Preinicializa Python desde la preconfiguración *preconfig* y argumentos de línea de comando (cadenas de caracteres anchas).

La persona que llama es responsable de manejar las excepciones (error o salida) usando `PyStatus_Exception()` y `Py_ExitStatusException()`.

Para *Configuración de Python* (`PyPreConfig_InitPythonConfig()`), si Python se inicializa con argumentos de línea de comando, los argumentos de línea de comando también deben pasarse para preinicializar Python, ya que tienen un efecto en la preconfiguración como codificaciones. Por ejemplo, la opción de línea de comando `-X utf8` habilita el modo UTF-8.

`PyMem_SetAllocator()` se puede llamar después de `Py_PreInitialize()` y antes `Py_InitializeFromConfig()` para instalar un asignador de memoria personalizado. Se puede llamar antes `Py_PreInitialize()` si `PyPreConfig.allocator` está configurado en `PYMEM_ALLOCATOR_NOT_SET`.

Las funciones de asignación de memoria de Python como `PyMem_RawMalloc()` no deben usarse antes de la preinicialización de Python, mientras que llamar directamente a `malloc()` y `free()` siempre es seguro. `Py_DecodeLocale()` no debe llamarse antes de la preinicialización.

Ejemplo usando la preinicialización para habilitar el modo UTF-8:

```
PyStatus status;
PyPreConfig preconfig;
PyPreConfig_InitPythonConfig(&preconfig);

preconfig.utf8_mode = 1;

status = Py_PreInitialize(&preconfig);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}

/* at this point, Python will speak UTF-8 */

Py_Initialize();
/* ... use Python API here ... */
Py_Finalize();
```

10.5 PyConfig

PyConfig

Estructura que contiene la mayoría de los parámetros para configurar Python.

Métodos de estructura:

void **PyConfig_InitPythonConfig** (*PyConfig *config*)

Inicializa la configuración con *Configuración de Python*.

void **PyConfig_InitIsolatedConfig** (*PyConfig *config*)

Inicializa la configuración con *Configuración aislada*.

PyStatus **PyConfig_SetString** (*PyConfig *config*, *wchar_t * const *config_str*, *const wchar_t *str*)

Copia la cadena de caracteres anchos *str* en **config_str*.

Preinicializa Python si es necesario.

PyStatus **PyConfig_SetBytesString** (*PyConfig *config*, *wchar_t * const *config_str*, *const char *str*)

Decodifica *str* usando `Py_DecodeLocale()` y configure el resultado en **config_str*.

Preinicializa Python si es necesario.

PyStatus **PyConfig_SetArgv** (*PyConfig *config*, *int argc*, *wchar_t * const *argv*)

Establezca argumentos de línea de comando a partir de cadenas de caracteres anchas.

Preinicializa Python si es necesario.

PyStatus **PyConfig_SetBytesArgv** (*PyConfig *config*, *int argc*, *char * const *argv*)

Establezca argumentos de línea de comando: decodifique bytes usando `Py_DecodeLocale()`.

Preinicializa Python si es necesario.

PyStatus **PyConfig_SetWideStringList** (*PyConfig* *config, *PyWideStringList* *list, *Py_ssize_t* length, wchar_t **items)

Establece la lista de cadenas de caracteres anchas *list* a *length* y *items*.

Preinicializa Python si es necesario.

PyStatus **PyConfig_Read** (*PyConfig* *config)

Lee toda la configuración de Python.

Los campos que ya están inicializados no se modifican.

Preinicializa Python si es necesario.

void **PyConfig_Clear** (*PyConfig* *config)

Libera memoria de configuración.

La mayoría de los métodos *PyConfig* preinicializan Python si es necesario. En ese caso, la configuración de preinicialización de Python se basa en *PyConfig*. Si los campos de configuración que son comunes con *PyPreConfig* están ajustados, deben configurarse antes de llamar al método *PyConfig*:

- *dev_mode*
- *isolated*
- *parse_argv*
- *use_environment*

Además, si se utiliza *PyConfig_SetArgv()* o *PyConfig_SetBytesArgv()*, este método debe llamarse primero, antes que otros métodos, ya que la configuración de preinicialización depende de los argumentos de la línea de comandos (si *parse_argv* no es cero).

Quien llama de estos métodos es responsable de manejar las excepciones (error o salida) usando *PyStatus_Exception()* y *Py_ExitStatusException()*.

Campos de estructura:

PyWideStringList **argv**

Argumentos de línea de comando, *sys.argv*. Consulta *parse_argv* para analizar *argv* de la misma manera que Python normal analiza los argumentos de la línea de comandos de Python. Si *argv* está vacío, se agrega una cadena de caracteres vacía para garantizar que *sys.argv* siempre exista y nunca esté vacío.

wchar_t* **base_exec_prefix**
sys.base_exec_prefix.

wchar_t* **base_executable**
sys._base_executable: `__PYENV_LAUNCHER__` valor de la variable de entorno, o copia de *PyConfig.executable*.

wchar_t* **base_prefix**
sys.base_prefix.

wchar_t* **platlibdir**
sys.platlibdir: nombre del directorio de la biblioteca de la plataforma, establecido en el momento de la configuración por `--with-platlibdir`, anulable por la variable de entorno `PYTHONPLATLIBDIR`.

Nuevo en la versión 3.9.

int **buffered_stdio**

Si es igual a 0, habilite el modo sin búfer, haciendo que las secuencias *stdout* y *stderr* no tengan búfer.

stdin siempre se abre en modo de búfer.

int bytes_warning

Si es igual a 1, emita una advertencia cuando compare `bytes` o `bytearray` con `str`, o compare `bytes` con `int`. Si es igual o mayor a 2, lanza una excepción `BytesWarning`.

wchar_t* check_hash_pycs_mode

Controla el comportamiento de validación de los archivos `.pyc` basados en hash (consulte [PEP 552](#)): con el valor de la opción de línea de comando `--check-hash-based-pycs`.

Valores válidos: `always`, `never` y `default`.

El valor predeterminado es: `default`.

int configure_c_stdio

Si no es cero, configure las secuencias estándar C (`stdio`, `stdout`, `stderr`). Por ejemplo, configure su modo en `O_BINARY` en Windows.

int dev_mode

Si es distinto de cero, habilita Modo de desarrollo de Python.

int dump_refs

Si no es cero, volcar todos los objetos que aún están vivos en la salida.

El macro `Py_TRACE_REFS` debe ser definido en la construcción.

wchar_t* exec_prefix

`sys.exec_prefix`.

wchar_t* executable

`sys.executable`.

int faulthandler

Si no es cero, llama a `faulthandler.enable()` al inicio.

wchar_t* filesystem_encoding

Codificación del sistema de archivos, `sys.getfilesystemencoding()`.

wchar_t* filesystem_errors

Errores de codificación del sistema de archivos, `sys.getfilesystemencodeerrors()`.

unsigned long hash_seed**int use_hash_seed**

Funciones de semillas aleatorias hash.

Si `use_hash_seed` es cero, se elige una semilla aleatoriamente en `Pythonstartup`, y `hash_seed` se ignora.

wchar_t* home

Directorio de inicio de Python.

Inicializado desde valor de variable de entorno `PYTHONHOME` por defecto.

int import_time

Si no es cero, el tiempo de importación del perfil.

int inspect

Ingresa al modo interactivo después de ejecutar un script o un comando.

int install_signal_handlers

¿Instala manejadores de señal?

int interactive

Modo interactivo.

int `isolated`

Si es mayor que 0, habilite el modo aislado:

- `sys.path` no contiene ni el directorio del script (calculado a partir de `argv[0]` o el directorio actual) ni el directorio de paquetes del sitio del usuario.
- Python REPL no importa `readline` ni habilita la configuración predeterminada de `readline` en mensajes interactivos.
- Establece `use_environment` y `user_site_directory` en 0.

int `legacy_windows_stdio`

Si no es cero, usa `io.FileIO` en lugar de `io.WindowsConsoleIO` para `sys.stdin`, `sys.stdout` y `sys.stderr`.

Solo disponible en Windows. La macro `#ifdef MS_WINDOWS` se puede usar para el código específico de Windows.

int `malloc_stats`

Si no es cero, volcar las estadísticas en *Asignador de memoria Python `pymalloc`* en la salida.

La opción se ignora si Python se construye usando `--without-pymalloc`.

wchar_t* `pythonpath_env`

Módulo de rutas de búsqueda como una cadena separada por `DELIM` (`os.path.pathsep`).

Inicializado desde valor de variable de entorno `PYTHONPATH` por defecto.

PyWideStringList* `module_search_paths`*int `module_search_paths_set`**

`sys.path`. Si `module_search_paths_set` es igual a 0, el `module_search_paths` es anulado por la función que calcula *Configuración de ruta*.

int `optimization_level`

Nivel de optimización de compilación:

- 0: Optimizador de mirilla (y `__debug__` está configurado como `True`)
- 1: Elimina las aserciones, establece `__debug__` en `False`
- 2: *Strip* docstrings

int `parse_argv`

Si no es cero, analiza `argv` de la misma manera que los argumentos regulares de la línea de comandos de Python, y elimine los argumentos de Python de `argv`: vea Argumentos de línea de comando.

int `parser_debug`

Si no es cero, activa la salida de depuración del analizador (solo para expertos, dependiendo de las opciones de compilación).

int `pathconfig_warnings`

Si es igual a 0, suprime las advertencias al calcular *Configuración de ruta* (solo Unix, Windows no registra ninguna advertencia). De lo contrario, las advertencias se escriben en `stderr`.

wchar_t* `prefix`

`sys.prefix`.

wchar_t* `program_name`

Nombre del programa. Se usa para inicializar *executable*, y en los primeros mensajes de error.

wchar_t* `pycache_prefix`

`sys.pycache_prefix`: prefijo de caché `.pyc`.

Si `NULL`, `sys.pycache_prefix` es establecido a `None`.

int quiet

Modo silencioso. Por ejemplo, no muestra los mensajes de copyright y versión en modo interactivo.

wchar_t* run_command

Argumento `python3 -c COMMAND`. Utilizado por `Py_RunMain()`.

wchar_t* run_filename

Argumento `python3 FILENAME`. Utilizado por `Py_RunMain()`.

wchar_t* run_module

Argumento `python3 -m MODULE`. Utilizado por `Py_RunMain()`.

int show_ref_count

¿Mostrar el recuento de referencia total en la salida?

Establecido en 1 por la opción de línea de comandos `-X showrefcount`.

Necesita una compilación de depuración de Python (se debe definir la macro `Py_REF_DEBUG`).

int site_import

¿Importar el módulo `site` al inicio?

int skip_source_first_line

¿Saltar la primera línea de la fuente?

wchar_t* stdio_encoding

wchar_t* stdio_errors

Codificación y codificación de errores de `sys.stdin`, `sys.stdout` y `sys.stderr`.

int tracemalloc

Si no es cero, llama a `tracemalloc.start()` al inicio.

int use_environment

Si es mayor que 0, use variables de entorno.

int user_site_directory

Si no es cero, agrega el directorio del sitio del usuario a `sys.path`.

int verbose

Si no es cero, habilita el modo detallado.

PyWideStringList **warnoptions**

`sys.warnoptions`: opciones del módulo `warnings` para crear filtros de advertencia: de menor a mayor prioridad.

El módulo `warnings` agrega `sys.warnoptions` en el orden inverso: el último elemento `PyConfig.warnoptions` se convierte en el primer elemento de `warnings.filters` que es verificado primero (máxima prioridad).

int write_bytecode

Si no es cero, escribe los archivos `.pyc`.

`sys.dont_write_bytecode` se inicializa al valor invertido de `write_bytecode`.

PyWideStringList **xoptions**

`sys._xoptions`.

int _use_peg_parser

Habilitar parser PEG? Por defecto: 1.

Establecido en 0 por `-X oldparser` y `PYTHONOLDPARSER`.

Vea también **PEP 617**.

Deprecated since version 3.9, will be removed in version 3.10.

Si `parse_argv` no es cero, los argumentos `argv` se analizan de la misma manera que Python analiza los argumentos de línea de comando, y los argumentos de Python se eliminan de `argv`: ver Argumentos de línea de comando.

Las opciones `xoptions` se analizan para establecer otras opciones: ver la opción `-X`.

Distinto en la versión 3.9: El campo `show_alloc_count` fue removido.

10.6 Inicialización con PyConfig

Función para inicializar Python:

PyStatus Py_InitializeFromConfig (const *PyConfig* **config*)

Inicializa Python desde la configuración *config*.

La persona que llama es responsable de manejar las excepciones (error o salida) usando *PyStatus_Exception()* y *Py_ExitStatusException()*.

If *PyImport_FrozenModules()*, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* are used, they must be set or called after Python preinitialization and before the Python initialization. If Python is initialized multiple times, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* must be called before each Python initialization.

Ejemplo de configuración del nombre del programa:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name. Implicitly preinitialize Python. */
    status = PyConfig_SetString(&config, &config.program_name,
                               L"/path/to/my_program");
    if (PyStatus_Exception(status)) {
        goto fail;
    }

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

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

Ejemplo más completo que modifica la configuración predeterminada, lee la configuración y luego anula algunos parámetros

```
PyStatus init_python(const char *program_name)
{
    PyStatus status;
```

(continué en la próxima página)

(proviene de la página anterior)

```

PyConfig config;
PyConfig_InitPythonConfig(&config);

/* Set the program name before reading the configuration
   (decode byte string from the locale encoding).

   Implicitly preinitialize Python. */
status = PyConfig_SetBytesString(&config, &config.program_name,
                                program_name);
if (PyStatus_Exception(status)) {
    goto done;
}

/* Read all configuration at once */
status = PyConfig_Read(&config);
if (PyStatus_Exception(status)) {
    goto done;
}

/* Append our custom search path to sys.path */
status = PyWideStringList_Append(&config.module_search_paths,
                                L"/path/to/more/modules");
if (PyStatus_Exception(status)) {
    goto done;
}

/* Override executable computed by PyConfig_Read() */
status = PyConfig_SetString(&config, &config.executable,
                            L"/path/to/my_executable");
if (PyStatus_Exception(status)) {
    goto done;
}

status = Py_InitializeFromConfig(&config);

done:
PyConfig_Clear(&config);
return status;
}

```

10.7 Configuración aislada

`PyPreConfig_InitIsolatedConfig()` y las funciones `PyConfig_InitIsolatedConfig()` crean una configuración para aislar Python del sistema. Por ejemplo, para incrustar Python en una aplicación.

This configuration ignores global configuration variables, environment variables, command line arguments (`PyConfig.argv` is not parsed) and user site directory. The C standard streams (ex: `stdout`) and the `LC_CTYPE` locale are left unchanged. Signal handlers are not installed.

Los archivos de configuración todavía se usan con esta configuración. Configure *Configuración de ruta* («campos de salida») para ignorar estos archivos de configuración y evitar la función que calcula la configuración de ruta predeterminada.

10.8 Configuración de Python

`PyPreConfig_InitPythonConfig()` y las funciones `PyConfig_InitPythonConfig()` crean una configuración para construir un Python personalizado que se comporta como el Python normal.

Las variables de entorno y los argumentos de la línea de comandos se utilizan para configurar Python, mientras que las variables de configuración global se ignoran.

Esta función permite la coerción de configuración regional C (**PEP 538**) y el modo UTF-8 (**PEP 540**) dependiendo de la configuración regional LC_CTYPE, PYTHONUTF8 y variables de entorno PYTHONCOERCECLOCALE.

Ejemplo de Python personalizado que siempre se ejecuta en modo aislado:

```
int main(int argc, char **argv)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config.isolated = 1;

    /* Decode command line arguments.
       Implicitly preinitialize Python (in isolated mode). */
    status = PyConfig_SetBytesArgv(&config, argc, argv);
    if (PyStatus_Exception(status)) {
        goto fail;
    }

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

    return Py_RunMain();
fail:
    PyConfig_Clear(&config);
    if (PyStatus_IsExit(status)) {
        return status.exitcode;
    }
    /* Display the error message and exit the process with
       non-zero exit code */
    Py_ExitStatusException(status);
}
```

10.9 Configuración de ruta

`PyConfig` contiene múltiples campos para la configuración de ruta:

- Entradas de configuración de ruta:
 - `PyConfig.home`
 - `PyConfig.platlibdir`
 - `PyConfig.pathconfig_warnings`

- `PyConfig.program_name`
 - `PyConfig.pythonpath_env`
 - directorio de trabajo actual: para obtener rutas absolutas
 - Variable de entorno `PATH` para obtener la ruta completa del programa (de `PyConfig.program_name`)
 - Variable de entorno `__PYENV_LAUNCHER__`
 - (Solo Windows) Rutas de aplicación en el registro en «SoftwarePythonPythonCoreX.YPythonPath» de `HKEY_CURRENT_USER` y `HKEY_LOCAL_MACHINE` (donde X.Y es la versión de Python).
- Campos de salida de configuración de ruta:
 - `PyConfig.base_exec_prefix`
 - `PyConfig.base_executable`
 - `PyConfig.base_prefix`
 - `PyConfig.exec_prefix`
 - `PyConfig.executable`
 - `PyConfig.module_search_paths_set`, `PyConfig.module_search_paths`
 - `PyConfig.prefix`

Si no se establece al menos un «campo de salida», Python calcula la configuración de la ruta para completar los campos no definidos. Si `module_search_paths_set` es igual a 0, `module_search_paths` se reemplaza y `module_search_paths_set` se establece en 1.

Es posible ignorar por completo la función que calcula la configuración de ruta predeterminada al establecer explícitamente todos los campos de salida de configuración de ruta enumerados anteriormente. Una cadena de caracteres se considera como un conjunto, incluso si no está vacía. `module_search_paths` se considera como establecido si `module_search_paths_set` se establece en 1. En este caso, los campos de entrada de configuración de ruta también se ignoran.

Establezca `pathconfig_warnings` en 0 para suprimir las advertencias al calcular la configuración de la ruta (solo Unix, Windows no registra ninguna advertencia).

Si `base_prefix` o los campos `base_exec_prefix` no están establecidos, heredan su valor de `prefix` y `exec_prefix` respectivamente.

`Py_RunMain()` y `Py_Main()` modifican `sys.path`:

- Si `run_filename` está configurado y es un directorio que contiene un script `__main__.py`, anteponga `run_filename` a `sys.path`.
- Si `isolated` es cero:
 - Si `run_module` está configurado, anteponga el directorio actual a `sys.path`. No haga nada si el directorio actual no se puede leer.
 - Si `run_filename` está configurado, anteponga el directorio del nombre del archivo a `sys.path`.
 - De lo contrario, anteponga una cadena de caracteres vacía a `sys.path`.

Si `site_import` no es cero, `sys.path` puede ser modificado por el módulo `site`. Si `user_site_directory` no es cero y el directorio del paquete del sitio del usuario existe, el módulo `site` agrega el directorio del paquete del sitio del usuario a `sys.path`.

La configuración de ruta utiliza los siguientes archivos de configuración:

- `pyvenv.cfg`

- `python._pth` (sólo Windows)
- `pybuilddir.txt` (sólo Unix)

La variable de entorno `__PYENVV_LAUNCHER__` se usa para establecer `PyConfig.base_executable`

10.10 Py_RunMain()

`int Py_RunMain (void)`

Ejecuta el comando (`PyConfig.run_command`), el script (`PyConfig.run_filename`) o el módulo (`PyConfig.run_module`) especificado en la línea de comando o en la configuración.

Por defecto y cuando se usa la opción `-i`, ejecuta el REPL.

Finalmente, finaliza Python y retorna un estado de salida que se puede pasar a la función `exit()`.

Consulte *Configuración de Python* para ver un ejemplo de Python personalizado que siempre se ejecuta en modo aislado usando `Py_RunMain()`.

10.11 Py_GetArgcArgv()

`void Py_GetArgcArgv (int *argc, wchar_t ***argv)`

Obtiene los argumentos originales de la línea de comandos, antes de que Python los modificara.

10.12 API Provisional Privada de Inicialización Multifásica

This section is a private provisional API introducing multi-phase initialization, the core feature of **PEP 432**:

- Fase de inicialización «Core», «Python mínimo»:
 - Tipos incorporados;
 - Excepciones incorporadas;
 - Módulos incorporados y congelados;
 - El módulo `sys` solo se inicializa parcialmente (por ejemplo `sys.path` aún no existe).
- Fase de inicialización «principal», Python está completamente inicializado:
 - Instala y configura `importlib`;
 - Aplique la *Configuración de ruta*;
 - Instala manejadores de señal;
 - Finaliza la inicialización del módulo `sys` (por ejemplo: crea `sys.stdout` y `sys.path`);
 - Habilita características opcionales como `faulthandler` y `tracemalloc`;
 - Importe el módulo `site`;
 - etc.

API provisional privada:

- `PyConfig._init_main`: si se establece en 0, `Py_InitializeFromConfig()` se detiene en la fase de inicialización «Core».

- `PyConfig._isolated_interpreter`: si no es cero, no permite hilos, subprocessos y bifurcaciones.

`PyStatus_Py_InitializeMain` (void)

Vaya a la fase de inicialización «Principal», finalice la inicialización de Python.

No se importa ningún módulo durante la fase «Core» y el módulo `importlib` no está configurado: la *Configuración de ruta* solo se aplica durante la fase «Principal». Puede permitir personalizar Python en Python para anular o ajustar *Configuración de ruta*, tal vez instale un importador personalizado `sys.meta_path` o un enlace de importación, etc.

Puede ser posible calcular *Configuración de ruta* en Python, después de la fase Core y antes de la fase Main, que es una de las motivaciones [PEP 432](#).

La fase «Núcleo» no está definida correctamente: lo que debería estar y lo que no debería estar disponible en esta fase aún no se ha especificado. La API está marcada como privada y provisional: la API se puede modificar o incluso eliminar en cualquier momento hasta que se diseñe una API pública adecuada.

Ejemplo de ejecución de código Python entre las fases de inicialización «Core» y «Main»:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config._init_main = 0;

    /* ... customize 'config' configuration ... */

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

    /* Use sys.stderr because sys.stdout is only created
       by _Py_InitializeMain() */
    int res = PyRun_SimpleString(
        "import sys; "
        "print('Run Python code before _Py_InitializeMain', "
        "      'file=sys.stderr')");
    if (res < 0) {
        exit(1);
    }

    /* ... put more configuration code here ... */

    status = _Py_InitializeMain();
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
}
```

11.1 Visión general

La gestión de memoria en Python implica un montón privado que contiene todos los objetos de Python y estructuras de datos. El *administrador de memoria de Python* garantiza internamente la gestión de este montón privado. El administrador de memoria de Python tiene diferentes componentes que se ocupan de varios aspectos de la gestión dinámica del almacenamiento, como compartir, segmentación, asignación previa o almacenamiento en caché.

En el nivel más bajo, un asignador de memoria sin procesar asegura que haya suficiente espacio en el montón privado para almacenar todos los datos relacionados con Python al interactuar con el administrador de memoria del sistema operativo. Además del asignador de memoria sin procesar, varios asignadores específicos de objeto operan en el mismo montón e implementan políticas de administración de memoria distintas adaptadas a las peculiaridades de cada tipo de objeto. Por ejemplo, los objetos enteros se administran de manera diferente dentro del montón que las cadenas, tuplas o diccionarios porque los enteros implican diferentes requisitos de almacenamiento y compensaciones de velocidad / espacio. El administrador de memoria de Python delega parte del trabajo a los asignadores específicos de objeto, pero asegura que este último opere dentro de los límites del montón privado.

Es importante comprender que la gestión del montón de Python la realiza el propio intérprete y que el usuario no tiene control sobre él, incluso si manipulan regularmente punteros de objetos a bloques de memoria dentro de ese montón. El administrador de memoria de Python realiza la asignación de espacio de almacenamiento dinámico para los objetos de Python y otros búferes internos a pedido a través de las funciones de API de Python/C enumeradas en este documento.

Para evitar daños en la memoria, los escritores de extensiones nunca deberían intentar operar en objetos Python con las funciones exportadas por la biblioteca C: `malloc()`, `calloc()`, `realloc()` y `free()`. Esto dará como resultado llamadas mixtas entre el asignador de C y el administrador de memoria de Python con consecuencias fatales, ya que implementan diferentes algoritmos y operan en diferentes montones. Sin embargo, uno puede asignar y liberar de forma segura bloques de memoria con el asignador de la biblioteca C para fines individuales, como se muestra en el siguiente ejemplo:

```
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
```

(continué en la próxima página)

(proviene de la página anterior)

```

    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyBytes_FromString(buf);
free(buf); /* malloc'ed */
return res;

```

En este ejemplo, la solicitud de memoria para el búfer de E/S es manejada por el asignador de la biblioteca C. El administrador de memoria de Python solo participa en la asignación del objeto de bytes retornado como resultado.

Sin embargo, en la mayoría de las situaciones, se recomienda asignar memoria del montón de Python específicamente porque este último está bajo el control del administrador de memoria de Python. Por ejemplo, esto es necesario cuando el intérprete se amplía con nuevos tipos de objetos escritos en C. Otra razón para usar el montón de Python es el deseo de *informar* al administrador de memoria de Python sobre las necesidades de memoria del módulo de extensión. Incluso cuando la memoria solicitada se usa exclusivamente para fines internos y altamente específicos, delegar todas las solicitudes de memoria al administrador de memoria de Python hace que el intérprete tenga una imagen más precisa de su huella de memoria en su conjunto. En consecuencia, bajo ciertas circunstancias, el administrador de memoria de Python puede o no desencadenar acciones apropiadas, como recolección de basura, compactación de memoria u otros procedimientos preventivos. Tenga en cuenta que al usar el asignador de la biblioteca C como se muestra en el ejemplo anterior, la memoria asignada para el búfer de E/S escapa completamente al administrador de memoria Python.

Ver también:

La variable de entorno PYTHONMALLOC puede usarse para configurar los asignadores de memoria utilizados por Python.

La variable de entorno PYTHONMALLOCSTATS se puede utilizar para imprimir estadísticas de [asignador de memoria pymalloc](#) cada vez que se crea un nuevo escenario de objetos pymalloc, y en el apagado.

11.2 Interfaz de memoria sin procesar

Los siguientes conjuntos de funciones son envoltorios para el asignador del sistema. Estas funciones son seguras para subprocesos, no es necesario mantener el *GIL*.

El *asignador de memoria sin procesar predeterminado* usa las siguientes funciones: `malloc()`, `calloc()`, `realloc()` y `free()`; llame a `malloc(1)` (o `calloc(1, 1)`) cuando solicita cero bytes.

Nuevo en la versión 3.4.

`void* PyMem_RawMalloc (size_t n)`

Asigna *n* bytes y retorna un puntero de tipo `void*` a la memoria asignada, o `NULL` si la solicitud falla.

Solicitar cero bytes retorna un puntero distinto que no sea `NULL` si es posible, como si en su lugar se hubiera llamado a `PyMem_RawMalloc(1)`. La memoria no se habrá inicializado de ninguna manera.

`void* PyMem_RawCalloc (size_t nelem, size_t elsize)`

Asigna *nelem* elementos cada uno cuyo tamaño en bytes es *elsize* y retorna un puntero de tipo `void*` a la memoria asignada, o `NULL` si la solicitud falla. La memoria se inicializa a ceros.

Solicitar elementos cero o elementos de tamaño cero bytes retorna un puntero distinto `NULL` si es posible, como si en su lugar se hubiera llamado `PyMem_RawCalloc(1, 1)`.

Nuevo en la versión 3.5.

`void* PyMem_RawRealloc (void *p, size_t n)`

Cambia el tamaño del bloque de memoria señalado por *p* a *n* bytes. Los contenidos no se modificarán al mínimo de los tamaños antiguo y nuevo.

Si *p* es `NULL`, la llamada es equivalente a `PyMem_RawMalloc(n)`; de lo contrario, si *n* es igual a cero, el bloque de memoria cambia de tamaño pero no se libera, y el puntero retornado no es `NULL`.

A menos que *p* sea NULL, debe haber sido retornado por una llamada previa a `PyMem_RawMalloc()`, `PyMem_RawRealloc()` o `PyMem_RawCalloc()`.

Si la solicitud falla, `PyMem_RawRealloc()` retorna NULL y *p* sigue siendo un puntero válido al área de memoria anterior.

void **PyMem_RawFree** (void **p*)

Libera el bloque de memoria al que apunta *p*, que debe haber sido retornado por una llamada anterior a `PyMem_RawMalloc()`, `PyMem_RawRealloc()` o `PyMem_RawCalloc()`. De lo contrario, o si se ha llamado antes a `PyMem_RawFree(p)`, se produce un comportamiento indefinido.

Si *p* es NULL, no se realiza ninguna operación.

11.3 Interfaz de memoria

Los siguientes conjuntos de funciones, modelados según el estándar ANSI C, pero que especifican el comportamiento cuando se solicitan cero bytes, están disponibles para asignar y liberar memoria del montón de Python.

El *asignador de memoria predeterminado* usa el *asignador de memoria* `pymalloc`.

Advertencia: El *GIL* debe mantenerse cuando se utilizan estas funciones.

Distinto en la versión 3.6: El asignador predeterminado ahora es `pymalloc` en lugar del `malloc()` del sistema.

void* **PyMem_Malloc** (size_t *n*)

Asigna *n* bytes y retorna un puntero de tipo `void*` a la memoria asignada, o NULL si la solicitud falla.

Solicitar cero bytes retorna un puntero distinto que no sea NULL si es posible, como si en su lugar se hubiera llamado a `PyMem_Malloc(1)`. La memoria no se habrá inicializado de ninguna manera.

void* **PyMem_Calloc** (size_t *nelem*, size_t *elsize*)

Asigna *nelem* elementos cada uno cuyo tamaño en bytes es *elsize* y retorna un puntero de tipo `void*` a la memoria asignada, o NULL si la solicitud falla. La memoria se inicializa a ceros.

Solicitar elementos cero o elementos de tamaño cero bytes retorna un puntero distinto NULL si es posible, como si en su lugar se hubiera llamado `PyMem_Calloc(1, 1)`.

Nuevo en la versión 3.5.

void* **PyMem_Realloc** (void **p*, size_t *n*)

Cambia el tamaño del bloque de memoria señalado por *p* a *n* bytes. Los contenidos no se modificarán al mínimo de los tamaños antiguo y nuevo.

Si *p* es NULL, la llamada es equivalente a `PyMem_Malloc(n)`; de lo contrario, si *n* es igual a cero, el bloque de memoria cambia de tamaño pero no se libera, y el puntero retornado no es NULL.

A menos que *p* sea NULL, debe haber sido retornado por una llamada previa a `PyMem_Malloc()`, `PyMem_Realloc()` o `PyMem_Calloc()`.

Si la solicitud falla, `PyMem_Realloc()` retorna NULL y *p* sigue siendo un puntero válido al área de memoria anterior.

void **PyMem_Free** (void **p*)

Libera el bloque de memoria señalado por *p*, que debe haber sido retornado por una llamada anterior a `PyMem_Malloc()`, `PyMem_Realloc()` o `PyMem_Calloc()`. De lo contrario, o si se ha llamado antes a `PyMem_Free(p)`, se produce un comportamiento indefinido.

Si *p* es NULL, no se realiza ninguna operación.

Las siguientes macros orientadas a tipos se proporcionan por conveniencia. Tenga en cuenta que *TYPE* se refiere a cualquier tipo de C.

TYPE* **PyMem_New** (TYPE, size_t *n*)

Igual que *PyMem_Malloc()*, pero asigna ($n * \text{sizeof}(\text{TYPE})$) bytes de memoria. Retorna una conversión de puntero a TYPE*. La memoria no se habrá inicializado de ninguna manera.

TYPE* **PyMem_Resize** (void **p*, TYPE, size_t *n*)

Igual que *PyMem_Realloc()*, pero el bloque de memoria cambia de tamaño a ($n * \text{sizeof}(\text{TYPE})$) bytes. Retorna una conversión de puntero a TYPE*. Al retornar, *p* será un puntero a la nueva área de memoria, o NULL en caso de falla.

Esta es una macro de preprocesador C; *p* siempre se reasigna. Guarde el valor original de *p* para evitar perder memoria al manejar errores.

void **PyMem_Del** (void **p*)

La misma que *PyMem_Free()*.

Además, se proporcionan los siguientes conjuntos de macros para llamar al asignador de memoria de Python directamente, sin involucrar las funciones de API de C mencionadas anteriormente. Sin embargo, tenga en cuenta que su uso no conserva la compatibilidad binaria entre las versiones de Python y, por lo tanto, está en desuso en los módulos de extensión.

- *PyMem_MALLOC(size)*
- *PyMem_NEW(type, size)*
- *PyMem_REALLOC(ptr, size)*
- *PyMem_RESIZE(ptr, type, size)*
- *PyMem_FREE(ptr)*
- *PyMem_DEL(ptr)*

11.4 Asignadores de objetos

Los siguientes conjuntos de funciones, modelados según el estándar ANSI C, pero que especifican el comportamiento cuando se solicitan cero bytes, están disponibles para asignar y liberar memoria del montón de Python.

El *asignador predeterminado de objetos* usa el *asignador de memoria pymalloc*.

Advertencia: El *GIL* debe mantenerse cuando se utilizan estas funciones.

void* **PyObject_Malloc** (size_t *n*)

Asigna *n* bytes y retorna un puntero de tipo void* a la memoria asignada, o NULL si la solicitud falla.

Solicitar cero bytes retorna un puntero distinto que no sea NULL si es posible, como si en su lugar se hubiera llamado a *PyObject_Malloc(1)*. La memoria no se habrá inicializado de ninguna manera.

void* **PyObject_Calloc** (size_t *nelem*, size_t *elsize*)

Asigna *nelem* elementos cada uno cuyo tamaño en bytes es *elsize* y retorna un puntero de tipo void* a la memoria asignada, o NULL si la solicitud falla. La memoria se inicializa a ceros.

Solicitar elementos cero o elementos de tamaño cero bytes retorna un puntero distinto NULL si es posible, como si en su lugar se hubiera llamado *PyObject_Calloc(1, 1)*.

Nuevo en la versión 3.5.

void* **PyObject_Realloc** (void **p*, size_t *n*)

Cambia el tamaño del bloque de memoria señalado por *p* a *n* bytes. Los contenidos no se modificarán al mínimo de los tamaños antiguo y nuevo.

Si *p* es NULL, la llamada es equivalente a `PyObject_Malloc(n)`; de lo contrario, si *n* es igual a cero, el bloque de memoria cambia de tamaño pero no se libera, y el puntero retornado no es NULL.

A menos que *p* sea NULL, debe haber sido retornado por una llamada previa a `PyObject_Malloc()`, `PyObject_Realloc()` o `PyObject_Calloc()`.

Si la solicitud falla, `PyObject_Realloc()` retorna NULL y *p* sigue siendo un puntero válido al área de memoria anterior.

void **PyObject_Free** (void **p*)

Libera el bloque de memoria al que apunta *p*, que debe haber sido retornado por una llamada anterior a `PyObject_Malloc()`, `PyObject_Realloc()` o `PyObject_Calloc()`. De lo contrario, o si se ha llamado antes a `PyObject_Free(p)`, se produce un comportamiento indefinido.

Si *p* es NULL, no se realiza ninguna operación.

11.5 Asignadores de memoria predeterminados

Asignadores de memoria predeterminados:

Configuración	Nombre	Py-Mem_RawMalloc	Py-Mem_Malloc	PyObject_Malloc
Lanzamiento de compilación	"pymalloc"	malloc	malloc + debug	malloc + debug
Compilación de depuración	"pymalloc_debug"	malloc + debug	pymalloc + debug	pymalloc + debug
Lanzamiento de compilación, sin pymalloc	"malloc"	malloc	malloc	malloc
Compilación de depuración, sin pymalloc	"malloc_debug"	malloc + debug	malloc + debug	malloc + debug

Leyenda:

- Nombre: valor para variable de entorno PYTHONMALLOC
- malloc: asignadores del sistema de la biblioteca C estándar, funciones C: `malloc()`, `calloc()`, `realloc()` y `free()`
- pymalloc: *asignador de memoria pymalloc*
- «+ debug»: con ganchos de depuración instalados por `PyMem_SetupDebugHooks()`

11.6 Personalizar asignadores de memoria

Nuevo en la versión 3.4.

PyMemAllocatorEx

Structure used to describe a memory block allocator. The structure has the following fields:

Campo	Significado
<code>void *ctx</code>	contexto de usuario pasado como primer argumento
<code>void* malloc(void *ctx, size_t size)</code>	asignar un bloque de memoria
<code>void* calloc(void *ctx, size_t nelem, size_t elsize)</code>	asignar un bloque de memoria inicializado con ceros
<code>void* realloc(void *ctx, void *ptr, size_t new_size)</code>	asignar o cambiar el tamaño de un bloque de memoria
<code>void free(void *ctx, void *ptr)</code>	liberar un bloque de memoria

Distinto en la versión 3.5: La estructura `PyMemAllocator` se renombró a `PyMemAllocatorEx` y se agregó un nuevo campo `calloc`.

PyMemAllocatorDomain

Enum se utiliza para identificar un dominio asignador. Dominios:

PYMEM_DOMAIN_RAW

Funciones:

- `PyMem_RawMalloc()`
- `PyMem_RawRealloc()`
- `PyMem_RawCalloc()`
- `PyMem_RawFree()`

PYMEM_DOMAIN_MEM

Funciones:

- `PyMem_Malloc()`,
- `PyMem_Realloc()`
- `PyMem_Calloc()`
- `PyMem_Free()`

PYMEM_DOMAIN_OBJ

Funciones:

- `PyObject_Malloc()`
- `PyObject_Realloc()`
- `PyObject_Calloc()`
- `PyObject_Free()`

`void PyMem_GetAllocator(PyMemAllocatorDomain domain, PyMemAllocatorEx *allocator)`

Obtenga el asignador de bloque de memoria del dominio especificado.

`void PyMem_SetAllocator(PyMemAllocatorDomain domain, PyMemAllocatorEx *allocator)`

Establece el asignador de bloque de memoria del dominio especificado.

El nuevo asignador debe retornar un puntero distinto `NULL` al solicitar cero bytes.

Para el dominio `PYMEM_DOMAIN_RAW`, el asignador debe ser seguro para subprocessos: el *GIL* no se mantiene cuando se llama al asignador.

Si el nuevo asignador no es un enlace (no llama al asignador anterior), se debe llamar a la función `PyMem_SetupDebugHooks()` para reinstalar los enlaces de depuración en la parte superior del nuevo asignador.

void **PyMem_SetupDebugHooks** (void)

Configurar ganchos para detectar errores en las funciones del asignador de memoria de Python.

La memoria recién asignada se llena con el byte `0xCD` (`CLEANBYTE`), la memoria liberada se llena con el byte `0xDD` (`DEADBYTE`). Los bloques de memoria están rodeados por «bytes prohibidos» (`FORBIDDENBYTE`: byte `0xFD`).

Verificaciones de tiempo de ejecución:

- Detecte violaciones de API, por ejemplo: `PyObject_Free()` llamado en un búfer asignado por `PyMem_Malloc()`
- Detectar escritura antes del inicio del búfer (desbordamiento del búfer)
- Detectar escritura después del final del búfer (desbordamiento del búfer)
- Comprueba que *GIL* se mantiene cuando las funciones del asignador de `PYMEM_DOMAIN_OBJ` (ej: `PyObject_Malloc()`) y dominios `PYMEM_DOMAIN_MEM` (por ejemplo: `PyMem_Malloc()`) se llaman

En caso de error, los enlaces de depuración usan el módulo `tracemalloc` para obtener el rastreo donde se asignó un bloque de memoria. El rastreo solo se muestra si `tracemalloc` rastrea las asignaciones de memoria de Python y se rastrea el bloque de memoria.

Estos enlaces son *instalado por defecto* si Python se compila en modo de depuración. La variable de entorno `PYTHONMALLOC` puede usarse para instalar enlaces de depuración en un Python compilado en modo de lanzamiento.

Distinto en la versión 3.6: Esta función ahora también funciona en Python compilado en modo de lanzamiento. En caso de error, los enlaces de depuración ahora usan `tracemalloc` para obtener el rastreo donde se asignó un bloque de memoria. Los enlaces de depuración ahora también verifican si el *GIL* se mantiene cuando se llaman a las funciones de `PYMEM_DOMAIN_OBJ` y dominios `PYMEM_DOMAIN_MEM`.

Distinto en la versión 3.8: Los patrones de bytes `0xCB` (`CLEANBYTE`), `0xDB` (`DEADBYTE`) y `0xFB` (`FORBIDDENBYTE`) han sido reemplazados por `0xCD`, `0xDD` y `0xFD` para usar los mismos valores que la depuración CRT de Windows `malloc()` y `free()`.

11.7 El asignador pymalloc

Python tiene un asignador *pymalloc* optimizado para objetos pequeños (más pequeños o iguales a 512 bytes) con una vida útil corta. Utiliza asignaciones de memoria llamadas «arenas» con un tamaño fijo de 256 KiB. Vuelve a `PyMem_RawMalloc()` y `PyMem_RawRealloc()` para asignaciones de más de 512 bytes.

pymalloc es el *asignador por defecto* de `PYMEM_DOMAIN_MEM` (por ejemplo: `PyMem_Malloc()`) y `PYMEM_DOMAIN_OBJ` (por ejemplo: `PyObject_Malloc()`) dominios.

El asignador de arena utiliza las siguientes funciones:

- `VirtualAlloc()` y `VirtualFree()` en Windows,
- `mmap()` y `munmap()` si está disponible,
- `malloc()` y `free()` en caso contrario.

11.7.1 Personalizar asignador de arena de pymalloc

Nuevo en la versión 3.4.

PyObjectArenaAllocator

Estructura utilizada para describir un asignador de arena. La estructura tiene tres campos:

Campo	Significado
<code>void *ctx</code>	contexto de usuario pasado como primer argumento
<code>void* alloc(void *ctx, size_t size)</code>	asignar una arena de bytes de tamaño
<code>void free(void *ctx, void *ptr, size_t size)</code>	liberar la arena

void **PyObject_GetArenaAllocator** (*PyObjectArenaAllocator* *allocator)

Consigue el asignador de arena.

void **PyObject_SetArenaAllocator** (*PyObjectArenaAllocator* *allocator)

Establecer el asignador de arena.

11.8 tracemalloc C API

Nuevo en la versión 3.7.

int **PyTraceMalloc_Track** (unsigned int *domain*, uintptr_t *ptr*, size_t *size*)

Rastree un bloque de memoria asignado en el módulo `tracemalloc`.

Retorna 0 en caso de éxito, retorna -1 en caso de error (no se pudo asignar memoria para almacenar la traza). Retorna -2 si `tracemalloc` está deshabilitado.

Si el bloque de memoria ya está rastreado, actualice el rastreo existente.

int **PyTraceMalloc_Untrack** (unsigned int *domain*, uintptr_t *ptr*)

Descomprima un bloque de memoria asignado en el módulo `tracemalloc`. No haga nada si el bloque no fue rastreado.

Retorna -2 si `tracemalloc` está deshabilitado; de lo contrario, retorna 0.

11.9 Ejemplos

Aquí está el ejemplo de la sección *Visión general*, reescrito para que el búfer de E/S se asigne desde el montón de Python utilizando el primer conjunto de funciones:

```
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```

El mismo código que utiliza el conjunto de funciones orientado a tipos:

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_New */
return res;
```

Tenga en cuenta que en los dos ejemplos anteriores, el búfer siempre se manipula a través de funciones que pertenecen al mismo conjunto. De hecho, es necesario usar la misma familia de API de memoria para un bloque de memoria dado, de modo que el riesgo de mezclar diferentes asignadores se reduzca al mínimo. La siguiente secuencia de código contiene dos errores, uno de los cuales está etiquetado como *fatal* porque mezcla dos asignadores diferentes que operan en montones diferentes.:

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3); /* Wrong -- should be PyMem_Free() */
free(buf2);      /* Right -- allocated via malloc() */
free(buf1);      /* Fatal -- should be PyMem_Del() */
```

Además de las funciones destinadas a manejar bloques de memoria sin procesar del montón de Python, los objetos en Python se asignan y liberan con `PyObject_New()`, `PyObject_NewVar()` y `PyObject_Del()`.

Esto se explicará en el próximo capítulo sobre cómo definir e implementar nuevos tipos de objetos en C.

Soporte de implementación de objetos

Este capítulo describe las funciones, los tipos y las macros utilizados al definir nuevos tipos de objetos.

12.1 Asignación de objetos en el montículo

*PyObject** **_PyObject_New** (*PyTypeObject* *type)

Return value: New reference.

*PyVarObject** **_PyObject_NewVar** (*PyTypeObject* *type, *Py_ssize_t* size)

Return value: New reference.

*PyObject** **PyObject_Init** (*PyObject* *op, *PyTypeObject* *type)

Return value: Borrowed reference. Inicializa un objeto *op* recientemente asignado con su tipo y referencia inicial. Retorna el objeto inicializado. Si *type* indica que el objeto participa en el detector de basura cíclico, se agrega al conjunto de objetos observados del detector. Otros campos del objeto no se ven afectados.

*PyVarObject** **PyObject_InitVar** (*PyVarObject* *op, *PyTypeObject* *type, *Py_ssize_t* size)

Return value: Borrowed reference. Esto hace todo lo que *PyObject_Init()* hace, y también inicializa la información de longitud para un objeto de tamaño variable.

TYPE* **PyObject_New** (**TYPE**, *PyTypeObject* *type)

Return value: New reference. Asigna un nuevo objeto Python usando el tipo de estructura de C *TYPE* y el objeto tipo Python *type*. Los campos no definidos por el encabezado del objeto Python no se inicializan; el conteo de referencias del objeto será uno. El tamaño de la asignación de memoria se determina a partir del campo *tp_basicsize* del tipo de objeto.

TYPE* **PyObject_NewVar** (**TYPE**, *PyTypeObject* *type, *Py_ssize_t* size)

Return value: New reference. Asigna un nuevo objeto Python usando el tipo de estructura de C *TYPE* y el objeto tipo Python *type*. Los campos no definidos por el encabezado del objeto Python no se inicializan. La memoria asignada permite los campos de la estructura *TYPE* más los campos *size* del tamaño dado por el campo *tp_itemsize* de *type*. Esto es útil para implementar objetos como tuplas, que pueden determinar su tamaño en el momento de la construcción. Incrustar el arreglo de campos en la misma asignación disminuye el número de asignaciones, mejorando la eficiencia de la gestión de memoria.

void **PyObject_Del** (void *op)

Libera memoria asignada a un objeto usando `PyObject_New()` o `PyObject_NewVar()`. Esto normalmente se llama desde el manejador `tp_dealloc` especificado en el tipo de objeto. No se debe acceder a los campos del objeto después de esta llamada, ya que la memoria ya no es un objeto Python válido.

PyObject_Py_NoneStruct

Objeto que es visible en Python como `None`. Esto solo se debe acceder utilizando el macro `Py_None`, que se evalúa como un puntero a este objeto.

Ver también:

`PyModule_Create()` Para asignar y crear módulos de extensión.

12.2 Estructuras de objetos comunes

Hay un gran número de estructuras que se utilizan en la definición de los tipos de objetos de Python. Esta sección describe estas estructuras y la forma en que se utilizan.

12.2.1 Tipos objeto base y macros

En última instancia, todos los objetos de Python comparten un pequeño número de campos en el comienzo de la representación del objeto en la memoria. Estos están representados por la `PyObject` y `PyVarObject`, que se definen, a su vez, por las expansiones de algunos macros también se utilizan, ya sea directa o indirectamente, en la definición de todos otros objetos de Python.

PyObject

Todos los tipos de objetos son extensiones de este tipo. Este es un tipo que contiene la información que Python necesita para tratar un puntero a un objeto como un objeto. En una construcción «*release*» normal, que contiene solo contador de referencia del objeto y un puntero al objeto de tipo correspondiente. En realidad nada es declarado como un `PyObject`, pero cada puntero a un objeto de Python se puede convertir en una `PyObject*`. El acceso a los miembros debe hacerse mediante el uso de las macros `Py_REFCNT` y `Py_TYPE`.

PyVarObject

Esta es una extensión de `PyObject` que se suma el campo `ob_size`. Esto sólo se utiliza para objetos que tienen cierta noción de longitud (*length*). Este tipo no suele aparecer en la API Python/C. El acceso a los miembros debe hacerse mediante el uso de las macros `Py_REFCNT`, `Py_TYPE`, y `Py_SIZE`.

PyObject_HEAD

Esta es una macro utilizado cuando se declara nuevos tipos que representan objetos sin una longitud variable. La macro `PyObject_HEAD` se expande a:

```
PyObject ob_base;
```

Consulte la documentación de `PyObject` en secciones anteriores.

PyObject_VAR_HEAD

Esta es una macro utilizado cuando se declara nuevos tipos que representan objetos con una longitud que varía de una instancia a otra instancia. La macro `PyObject_VAR_HEAD` se expande a:

```
PyVarObject ob_base;
```

Consulte la documentación de `PyVarObject` anteriormente.

Py_TYPE(o)

Esta macro se utiliza para acceder al miembro `ob_type` de un objeto Python. Se expande a:

```
((PyObject*) (o))->ob_type)
```

int **Py_IS_TYPE** (*PyObject* *o, *PyTypeObject* *type)

Retorna un valor distinto de cero si el objeto *o* tipo es *type*. Retorna cero en caso contrario. Equivalente a:
`Py_TYPE(o) == type.`

Nuevo en la versión 3.9.

void **Py_SET_TYPE** (*PyObject* *o, *PyTypeObject* *type)

Establece el tipo del objeto *o* a *type*.

Nuevo en la versión 3.9.

Py_REFCNT (o)

Esta macro se utiliza para acceder al miembro `ob_refcnt` de un objeto Python. Se expande a:

```
((PyObject*) (o))->ob_refcnt)
```

void **Py_SET_REFCNT** (*PyObject* *o, *Py_ssize_t* refcnt)

Establece el conteo de referencia del objeto *o* a *refcnt*.

Nuevo en la versión 3.9.

Py_SIZE (o)

Esta macro se utiliza para acceder al miembro `ob_size` de un objeto Python. Se expande a:

```
((PyVarObject*) (o))->ob_size)
```

void **Py_SET_SIZE** (*PyVarObject* *o, *Py_ssize_t* size)

Establece el tamaño del objeto *o* a *size*.

Nuevo en la versión 3.9.

PyObject_HEAD_INIT (type)

Esta es una macro que se expande para valores de inicialización para un nuevo tipo *PyObject*. Esta macro expande:

```
_PyObject_EXTRA_INIT
1, type,
```

PyVarObject_HEAD_INIT (type, size)

Esta es una macro que se expande para valores de inicialización para un nuevo tipo *PyVarObject*, incluyendo el campo `ob_size`. Esta macro se expande a:

```
_PyObject_EXTRA_INIT
1, type, size,
```

12.2.2 Implementando funciones y métodos

PyCFunction

Tipo de las funciones usadas para implementar la mayoría de invocables Python en C. Funciones de este tipo toman dos parámetros *PyObject* * y retorna un valor de ese tipo. Si el valor de retorno es `NULL`, una excepción fue establecida. Si no es `NULL`, el valor retornado se interpreta como el valor de retorno de la función que se expone en Python. La función debe retornar una nueva referencia.

La firma de la función es:

```
PyObject *PyCFunction(PyObject *self,
                      PyObject *args);
```

PyCFunctionWithKeywords

Tipo de las funciones que se utilizan para implementar invocables Python en C con la firma `METH_VARARGS` | `METH_KEYWORDS`. La firma de la función es:

```
PyObject *PyCFunctionWithKeywords(PyObject *self,
                                  PyObject *args,
                                  PyObject *kwargs);
```

_PyCFunctionFast

Tipo de las funciones que se utilizan para implementar invocables Python en C con la firma `METH_FASTCALL`. La firma de la función es:

```
PyObject *_PyCFunctionFast(PyObject *self,
                           PyObject *const *args,
                           Py_ssize_t nargs);
```

_PyCFunctionFastWithKeywords

Tipo de las funciones que se utilizan para implementar invocables Python en C con la firma `METH_FASTCALL` | `METH_KEYWORDS`. La firma de la función es:

```
PyObject *_PyCFunctionFastWithKeywords(PyObject *self,
                                       PyObject *const *args,
                                       Py_ssize_t nargs,
                                       PyObject *kwnames);
```

PyCMethod

Tipo de las funciones que se utilizan para implementar invocables Python en C con la firma `METH_METHOD` | `METH_FASTCALL` | `METH_KEYWORDS`. La firma de la función es:

```
PyObject *PyCMethod(PyObject *self,
                    PyTypeObject *defining_class,
                    PyObject *const *args,
                    Py_ssize_t nargs,
                    PyObject *kwnames)
```

Nuevo en la versión 3.9.

PyMethodDef

Estructura utilizada para describir un método de un tipo de extensión. Esta estructura tiene cuatro campos:

Campo	Tipo C	Significado
<code>ml_name</code>	<code>const char *</code>	nombre del método
<code>ml_meth</code>	<code>PyCFunction</code>	puntero a la implementación en C
<code>ml_flags</code>	<code>int</code>	<i>flag</i> bits que indican cómo debe ser construida la llamada
<code>ml_doc</code>	<code>const char *</code>	puntero a los contenidos del docstring

El `ml_meth` es un puntero de función C. Las funciones pueden ser de diferentes tipos, pero siempre retornan `PyObject *`. Si la función no es la de `PyCFunction`, el compilador requiere una conversión de tipo en la tabla de métodos. A pesar de que `PyCFunction` define el primer parámetro como `PyObject *`, es común que la implementación del método utilice el tipo específico C del objeto *self*.

El campo `ml_flags` es un campo de bits que puede incluir las siguientes *flags*. Las *flags* individuales indican o bien una convención de llamada o una convención vinculante.

Existen estas convenciones de llamada:

METH_VARARGS

Esta es la convención de llamada típica, donde los métodos tienen el tipo *PyCFunction*. La función espera dos valores *PyObject**. El primero es objeto *self* para los métodos; para las funciones del módulo, que es el objeto módulo. El segundo parámetro (a menudo llamado *args*) es un objeto tupla que representa todos los argumentos. Este parámetro se procesa típicamente usando *PyArg_ParseTuple()* o *PyArg_UnpackTuple()*.

METH_VARARGS | METH_KEYWORDS

Los métodos con estas *flags* deben ser del tipo *PyCFunctionWithKeywords*. La función espera tres parámetros: *self*, *args*, *kwargs* donde *kwargs* es un diccionario de todos los argumentos de palabras clave o, posiblemente, NULL si no hay argumentos de palabra clave. Los parámetros se procesan típicamente usando *PyArg_ParseTupleAndKeywords()*.

METH_FASTCALL

Convención de llamando rápido que soporta sólo argumentos posicionales. Los métodos tienen el tipo *_PyCFunctionFast*. El primer parámetro es *self*, el segundo parámetro es un arreglo C de valores *PyObject** que indican los argumentos y el tercer parámetro es el número de argumentos (la longitud del arreglo).

Esto no es parte de la *API limitada*.

Nuevo en la versión 3.7.

METH_FASTCALL | METH_KEYWORDS

Extensión de *METH_FASTCALL* que admite también argumentos de palabra clave, con los métodos de tipo *_PyCFunctionFastWithKeywords*. argumentos de palabra clave se transmiten de la misma manera como en el *vectorcall protocol*: hay un cuarto parámetro *PyObject** adicional que es una tupla que representa los nombres de los argumentos de palabra clave o posiblemente NULL si no hay palabras clave. Los valores de los argumentos de palabras clave se almacenan en el arreglo *args*, después de los argumentos posicionales.

Esto no es parte de la *API limitada*.

Nuevo en la versión 3.7.

METH_METHOD | METH_FASTCALL | METH_KEYWORDS

Extensión de *METH_FASTCALL | METH_KEYWORDS* que admite la *clase definitoria*, es decir, la clase que contiene el método en cuestión. La clase definitoria podría ser una superclase de *Py_TYPE(self)*.

El método debe ser de tipo *PyCMethod*, lo mismo que para *METH_FASTCALL | METH_KEYWORDS* con el argumento *defining_class* añadido después de *self*.

Nuevo en la versión 3.9.

METH_NOARGS

Métodos sin parámetros no tienen que comprobar si los argumentos se dan si están registrados con el *flag METH_NOARGS*. Tienen que ser de tipo *PyCFunction*. El primer parámetro normalmente se denomina *self* y llevará a cabo una referencia a la instancia módulo u objeto. En todos los casos el segundo parámetro será NULL.

METH_O

Los métodos con un solo argumento objeto pueden ser listados con el *flag METH_O*, en lugar de invocar *PyArg_ParseTuple()* con un argumento "O". Tienen el tipo *PyCFunction*, con el parámetro *self*, y un parámetro *PyObject** que representa el único argumento.

Estas dos constantes no se utilizan para indicar la convención de llamada si no la vinculación cuando su usan con métodos de las clases. Estos no se pueden usar para funciones definidas para módulos. A lo sumo uno de estos *flags* puede establecerse en un método dado.

METH_CLASS

Al método se le pasará el objeto tipo como primer parámetro, en lugar de una instancia del tipo. Esto se utiliza para

crear métodos de clase (*class methods*), similar a lo que se crea cuando se utiliza la función `classmethod()` incorporada.

METH_STATIC

El método pasará `NULL` como el primer parámetro en lugar de una instancia del tipo. Esto se utiliza para crear métodos estáticos (*static methods*), similar a lo que se crea cuando se utiliza la función `staticmethod()` incorporada.

En otros controles constantes dependiendo si se carga un método en su lugar (*in place*) de otra definición con el mismo nombre del método.

METH_COEXIST

El método se cargará en lugar de las definiciones existentes. Sin *METH_COEXIST*, el comportamiento predeterminado es saltarse las definiciones repetidas. Desde envolturas de ranura se cargan antes de la tabla de métodos, la existencia de una ranura *sq_contains*, por ejemplo, generaría un método envuelto llamado `__contains__()` e impediría la carga de una `PyCFunction` correspondiente con el mismo nombre. Con el *flag* definido, la `PyCFunction` se cargará en lugar del objeto envoltorio y coexistirá con la ranura. Esto es útil porque las llamadas a `PyCFunctions` se optimizan más que las llamadas a objetos envoltorio.

12.2.3 Acceder a atributos de tipos de extensión

PyMemberDef

Estructura que describe un atributo de un tipo que corresponde a un miembro de la estructura de C. Sus campos son:

Campo	Tipo C	Significado
<code>name</code>	<code>const char *</code>	nombre del miembro
<code>type</code>	<code>int</code>	el tipo de miembro en la estructura de C
<code>offset</code>	<code>Py_ssize_t</code>	el desplazamiento en bytes que el miembro se encuentra en la estructura de objetos tipo
<code>flags</code>	<code>int</code>	<i>flags</i> bits que indican si el campo debe ser de sólo lectura o de escritura
<code>doc</code>	<code>const char *</code>	puntos a los contenidos del docstring

`type` puede ser uno de muchos macros `T_` correspondientes a diversos tipos C. Cuando se accede al miembro en Python, será convertida al tipo Python equivalente.

Nombre de la macro	Tipo C
T_SHORT	short
T_INT	int
T_LONG	long
T_FLOAT	float
T_DOUBLE	double
T_STRING	const char *
T_OBJECT	PyObject *
T_OBJECT_EX	PyObject *
T_CHAR	char
T_BYTE	char
T_UBYTE	unsigned char
T_UINT	unsigned int
T_USHORT	unsigned short
T_ULONG	unsigned long
T_BOOL	char
T_LONGLONG	long long
T_ULONGLONG	unsigned long long
T_PYSSIZET	Py_ssize_t

T_OBJECT y T_OBJECT_EX se diferencian en que T_OBJECT retorna None si el miembro es NULL y T_OBJECT_EX lanza un AttributeError. Trate de usar T_OBJECT_EX sobre T_OBJECT porque T_OBJECT_EX maneja el uso de la declaración del en ese atributo más correctamente que T_OBJECT.

flags puede ser 0 para el acceso de escritura y lectura o READONLY para el acceso de sólo lectura. El uso de T_STRING para type implica READONLY. Los datos T_STRING se interpretan como UTF-8. Sólo se pueden eliminar T_OBJECT y miembros T_OBJECT_EX. (Se establecen a NULL).

Los tipos asignados al heap (creados usando `PyType_FromSpec()` o similar), PyMemberDef pueden contener definiciones para los miembros especiales `__dictoffset__`, `__weaklistoffset__` y `__vectorcalloffset__`, correspondientes a `tp_dictoffset`, `tp_weaklistoffset` y `tp_vectorcall_offset` en objetos de tipo. Estos deben definirse con T_PYSSIZET y READONLY, por ejemplo:

```
static PyMemberDef spam_type_members[] = {
    {"__dictoffset__", T_PYSSIZET, offsetof(Spam_object, dict), READONLY},
    {NULL} /* Sentinel */
};
```

*PyObject** **PyMember_GetOne** (const char *obj_addr, struct *PyMemberDef* *m)

Get an attribute belonging to the object at address *obj_addr*. The attribute is described by *PyMemberDef* *m*. Returns NULL on error.

int **PyMember_SetOne** (char *obj_addr, struct *PyMemberDef* *m, *PyObject* *o)

Set an attribute belonging to the object at address *obj_addr* to object *o*. The attribute to set is described by *PyMemberDef* *m*. Returns 0 if successful and a negative value on failure.

PyGetSetDef

Estructura para definir el acceso para un tipo como el de una propiedad. Véase también la descripción de la ranura `PyTypeObject.tp_getset`.

Campo	Tipo C	Significado
nombre	const char *	Nombre del atributo
get	getter	C function to get the attribute
set	setter	función opcional C para establecer o eliminar el atributo, si se omite el atributo es de sólo lectura
doc	const char *	docstring opcional
clausura (<i>closure</i>)	void *	puntero de función opcional, proporcionar datos adicionales para <i>getter</i> y <i>setter</i>

La función *get* toma un parámetro *PyObject ** (la instancia) y un puntero de función (el *closure* asociado):

```
typedef PyObject *(*getter)(PyObject *, void *);
```

Debe retornar una nueva referencia en caso de éxito o NULL con una excepción establecida en caso de error.

Las funciones *set* toman dos parámetros *PyObject ** (la instancia y el valor a ser establecido) y un puntero de función (el *closure* asociado):

```
typedef int (*setter)(PyObject *, PyObject *, void *);
```

En caso de que el atributo deba suprimirse el segundo parámetro es NULL. Debe retornar 0 en caso de éxito o -1 con una excepción explícita en caso de fallo.

12.3 Objetos Tipo

Quizás una de las estructuras más importantes del sistema de objetos Python es la estructura que define un nuevo tipo: la estructura *PyTypeObject*. Los objetos tipo se pueden manejar utilizando cualquiera de las funciones *PyObject_*()* o *PyType_*()*, pero no ofrecen mucho que sea interesante para la mayoría de las aplicaciones de Python. Estos objetos son fundamentales para el comportamiento de los objetos, por lo que son muy importantes para el propio intérprete y para cualquier módulo de extensión que implemente nuevos tipos.

Los objetos de tipo son bastante grandes en comparación con la mayoría de los tipos estándar. La razón del tamaño es que cada objeto de tipo almacena una gran cantidad de valores, principalmente punteros de función C, cada uno de los cuales implementa una pequeña parte de la funcionalidad del tipo. Los campos del objeto tipo se examinan en detalle en esta sección. Los campos se describirán en el orden en que aparecen en la estructura.

Además de la siguiente referencia rápida, la sección *Ejemplos* proporciona una visión rápida del significado y uso de *PyTypeObject*.

12.3.1 Referencia rápida

«ranuras *tp*» (*tp slots*)

Ranura <i>PyTypeObject</i> ¹	<i>Type</i>	métodos/atributos especiales	Información ²			
			O	T	D	I
<R> <i>tp_name</i>	const char *	__name__	X	X		
<i>tp_basicsize</i>	<i>Py_ssize_t</i>		X	X		X
<i>tp_itemsize</i>	<i>Py_ssize_t</i>			X		X

Continúa en la página siguiente

Tabla 1 – proviene de la página anterior

Ranura PyObject ¹	Type	métodos/atributos especiales	Información ²				
			O	T	D	I	
<code>tp_dealloc</code>	<code>destructor</code>		X	X			X
<code>tp_vectorcall_offset</code>	<code>Py_ssize_t</code>			X			X
<code>(tp_getattr)</code>	<code>getattrfunc</code>	<code>__getattr__</code> , <code>__getattr__</code>					G
<code>(tp_setattr)</code>	<code>setattrfunc</code>	<code>__setattr__</code> , <code>__delattr__</code>					G
<code>tp_as_async</code>	<code>PyAsyncMethods *</code>	<i>sub-ranuras (sub-slots)</i>					%
<code>tp_repr</code>	<code>reprfunc</code>	<code>__repr__</code>	X	X			X
<code>tp_as_number</code>	<code>PyNumberMethods *</code>	<i>sub-ranuras (sub-slots)</i>					%
<code>tp_as_sequence</code>	<code>PySequenceMethods *</code>	<i>sub-ranuras (sub-slots)</i>					%
<code>tp_as_mapping</code>	<code>PyMappingMethods *</code>	<i>sub-ranuras (sub-slots)</i>					%
<code>tp_hash</code>	<code>hashfunc</code>	<code>__hash__</code>	X				G
<code>tp_call</code>	<code>ternaryfunc</code>	<code>__call__</code>		X			X
<code>tp_str</code>	<code>reprfunc</code>	<code>__str__</code>	X				X
<code>tp_getattro</code>	<code>getattrofunc</code>	<code>__getattr__</code> , <code>__getattr__</code>	X	X			G
<code>tp_setattro</code>	<code>setattrofunc</code>	<code>__setattr__</code> , <code>__delattr__</code>	X	X			G
<code>tp_as_buffer</code>	<code>PyBufferProcs *</code>						%
<code>tp_flags</code>	<code>unsigned long</code>		X	X			?
<code>tp_doc</code>	<code>const char *</code>	<code>__doc__</code>	X	X			
<code>tp_traverse</code>	<code>traverseproc</code>			X			G
<code>tp_clear</code>	<code>inquiry</code>			X			G
<code>tp_richcompare</code>	<code>richcmpfunc</code>	<code>__lt__</code> , <code>__le__</code> , <code>__eq__</code> , <code>__ne__</code> , <code>__gt__</code> , <code>__ge__</code>	X				G
<code>tp_weaklistoffset</code>	<code>Py_ssize_t</code>			X			?
<code>tp_iter</code>	<code>getiterfunc</code>	<code>__iter__</code>					X
<code>tp_iternext</code>	<code>iternextfunc</code>	<code>__next__</code>					X
<code>tp_methods</code>	<code>PyMethodDef []</code>		X	X			
<code>tp_members</code>	<code>PyMemberDef []</code>			X			
<code>tp_getset</code>	<code>PyGetSetDef []</code>		X	X			
<code>tp_base</code>	<code>PyObject *</code>	<code>__base__</code>				X	
<code>tp_dict</code>	<code>PyObject *</code>	<code>__dict__</code>				?	
<code>tp_descr_get</code>	<code>descrgetfunc</code>	<code>__get__</code>					X
<code>tp_descr_set</code>	<code>descrsetfunc</code>	<code>__set__</code> , <code>__delete__</code>					X
<code>tp_dictoffset</code>	<code>Py_ssize_t</code>			X			?
<code>tp_init</code>	<code>initproc</code>	<code>__init__</code>	X	X			X
<code>tp_alloc</code>	<code>allocfunc</code>		X		?	?	
<code>tp_new</code>	<code>newfunc</code>	<code>__new__</code>	X	X	?	?	
<code>tp_free</code>	<code>freefunc</code>		X	X	?	?	
<code>tp_is_gc</code>	<code>inquiry</code>			X			X
<code><tp_bases></code>	<code>PyObject *</code>	<code>__bases__</code>				~	
<code><tp_mro></code>	<code>PyObject *</code>	<code>__mro__</code>				~	
<code>[tp_cache]</code>	<code>PyObject *</code>						
<code>[tp_subclasses]</code>	<code>PyObject *</code>	<code>__subclasses__</code>					
<code>[tp_weaklist]</code>	<code>PyObject *</code>						
<code>(tp_del)</code>	<code>destructor</code>						
<code>[tp_version_tag]</code>	<code>unsigned int</code>						
<code>tp_finalize</code>	<code>destructor</code>	<code>__del__</code>					X
<code>tp_vectorcall</code>	<code>vectorcallfunc</code>						

¹ Un nombre de ranura entre paréntesis indica que está (efectivamente) en desuso. Los nombres entre paréntesis angulares deben tratarse como de solo lectura. Los nombres entre corchetes son solo para uso interno. «<R>» (como prefijo) significa que el campo es obligatorio (no debe ser NULL).

² Columnas:

sub-ranuras (*sub-slots*)

Ranuras (<i>Slot</i>)	Type	métodos especiales
<code>am_await</code>	<code>unaryfunc</code>	<code>__await__</code>
<code>am_aiter</code>	<code>unaryfunc</code>	<code>__aiter__</code>
<code>am_anext</code>	<code>unaryfunc</code>	<code>__anext__</code>
<code>nb_add</code>	<code>binaryfunc</code>	<code>__add__</code> <code>__radd__</code>
<code>nb_inplace_add</code>	<code>binaryfunc</code>	<code>__iadd__</code>
<code>nb_subtract</code>	<code>binaryfunc</code>	<code>__sub__</code> <code>__rsub__</code>
<code>nb_inplace_subtract</code>	<code>binaryfunc</code>	<code>__isub__</code>
<code>nb_multiply</code>	<code>binaryfunc</code>	<code>__mul__</code> <code>__rmul__</code>
<code>nb_inplace_multiply</code>	<code>binaryfunc</code>	<code>__imul__</code>
<code>nb_remainder</code>	<code>binaryfunc</code>	<code>__mod__</code> <code>__rmod__</code>
<code>nb_inplace_remainder</code>	<code>binaryfunc</code>	<code>__imod__</code>
<code>nb_divmod</code>	<code>binaryfunc</code>	<code>__divmod__</code> <code>__rdivmod__</code>
<code>nb_power</code>	<code>ternaryfunc</code>	<code>__pow__</code> <code>__rpow__</code>
<code>nb_inplace_power</code>	<code>ternaryfunc</code>	<code>__ipow__</code>
<code>nb_negative</code>	<code>unaryfunc</code>	<code>__neg__</code>
<code>nb_positive</code>	<code>unaryfunc</code>	<code>__pos__</code>
<code>nb_absolute</code>	<code>unaryfunc</code>	<code>__abs__</code>
<code>nb_bool</code>	<code>inquiry</code>	<code>__bool__</code>
<code>nb_invert</code>	<code>unaryfunc</code>	<code>__invert__</code>
<code>nb_lshift</code>	<code>binaryfunc</code>	<code>__lshift__</code> <code>__rlshift__</code>
<code>nb_inplace_lshift</code>	<code>binaryfunc</code>	<code>__ilshift__</code>
<code>nb_rshift</code>	<code>binaryfunc</code>	<code>__rshift__</code> <code>__rrshift__</code>
<code>nb_inplace_rshift</code>	<code>binaryfunc</code>	<code>__irshift__</code>
<code>nb_and</code>	<code>binaryfunc</code>	<code>__and__</code> <code>__rand__</code>
<code>nb_inplace_and</code>	<code>binaryfunc</code>	<code>__iand__</code>
<code>nb_xor</code>	<code>binaryfunc</code>	<code>__xor__</code> <code>__rxor__</code>
<code>nb_inplace_xor</code>	<code>binaryfunc</code>	<code>__ixor__</code>
<code>nb_or</code>	<code>binaryfunc</code>	<code>__or__</code> <code>__ror__</code>
<code>nb_inplace_or</code>	<code>binaryfunc</code>	<code>__ior__</code>
<code>nb_int</code>	<code>unaryfunc</code>	<code>__int__</code>
<code>nb_reserved</code>	<code>void *</code>	
<code>nb_float</code>	<code>unaryfunc</code>	<code>__float__</code>
<code>nb_floor_divide</code>	<code>binaryfunc</code>	<code>__floordiv__</code>
<code>nb_inplace_floor_divide</code>	<code>binaryfunc</code>	<code>__ifloordiv__</code>
«O»: establecido en <code>PyBaseObject_Type</code>		
«I»: establecido en <code>PyType_Type</code>	<code>binaryfunc</code>	<code>__truediv__</code>
«D»: por defecto (si la ranura está establecida como NULL)	<code>binaryfunc</code>	<code>__itruediv__</code>

X - `PyType_Ready` sets this value if it is NULL

~ - `PyType_Ready` always sets this value (it should be NULL)

? - `PyType_Ready` may set this value depending on other slots

Also see the inheritance column ("I").

«I»: herencia	<code>lenfunc</code>	<code>__len__</code>
---------------	----------------------	----------------------

X - type slot is inherited via `*PyType_Ready*` if defined with a `*NULL*` value

% - the slots of the sub-struct are inherited individually

G - inherited, but only in combination with other slots; see the slot's description

? - it's complicated; see the slot's description

Tenga en cuenta que algunos espacios se heredan efectivamente a través de la cadena de búsqueda de atributos normal.

Tabla 2 – proviene de la página anterior

Ranuras (<i>Slot</i>)	<i>Type</i>	métodos especiales
<i>mp_ass_subscript</i>	<i>objobjargproc</i>	<code>__setitem__</code> , <code>__delitem__</code>
<i>sq_length</i>	<i>lenfunc</i>	<code>__len__</code>
<i>sq_concat</i>	<i>binaryfunc</i>	<code>__add__</code>
<i>sq_repeat</i>	<i>ssizeargfunc</i>	<code>__mul__</code>
<i>sq_item</i>	<i>ssizeargfunc</i>	<code>__getitem__</code>
<i>sq_ass_item</i>	<i>ssizeobjargproc</i>	<code>__setitem__</code> , <code>__delitem__</code>
<i>sq_contains</i>	<i>objobjproc</i>	<code>__contains__</code>
<i>sq_inplace_concat</i>	<i>binaryfunc</i>	<code>__iadd__</code>
<i>sq_inplace_repeat</i>	<i>ssizeargfunc</i>	<code>__imul__</code>
<i>bf_getbuffer</i>	<i>getbufferproc()</i>	
<i>bf_releasebuffer</i>	<i>releasebufferproc()</i>	

ranura de *typedefs*

typedef	Tipos Parámetros	Tipo de Retorno
<i>allocfunc</i>	<i>PyObject</i> * <i>Py_ssize_t</i>	<i>PyObject</i> *
<i>destructor</i>	void *	void
<i>freefunc</i>	void *	void
<i>traverseproc</i>	void * <i>visitproc</i> void *	int
<i>newfunc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
<i>initproc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	int
<i>reprfunc</i>	<i>PyObject</i> *	<i>PyObject</i> *
<i>getattrfunc</i>	<i>PyObject</i> * const char *	<i>PyObject</i> *
<i>setattrfunc</i>	<i>PyObject</i> * const char * <i>PyObject</i> *	int
<i>getattrofunc</i>	<i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
<i>setattrofunc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	int
<i>descrgetfunc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
212 <i>descrsetfunc</i>	<i>PyObject</i> * <i>PyObject</i> *	int

Vea *Tipo Ranura typedefs* abajo para más detalles.

12.3.2 Definición de PyObject

La definición de estructura para *PyObject* se puede encontrar en `Include/object.h`. Por conveniencia de referencia, esto repite la definición encontrada allí:

```
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;
    getattrofunc tp_getattr;
    setattrofunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                   or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

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

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

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

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

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

    const char *tp_doc; /* Documentation string */

    /* 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 */
}
```

(continué en la próxima página)

(proviene de la página anterior)

```

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;

```

12.3.3 Ranuras (Slots) PyObject

The type object structure extends the *PyVarObject* structure. The *ob_size* field is used for dynamic types (created by *type_new()*, usually called from a class statement). Note that *PyType_Type* (the metatype) initializes *tp_itemsize*, which means that its instances (i.e. type objects) *must* have the *ob_size* field.

*PyObject** **PyObject._ob_next**

*PyObject** **PyObject._ob_prev**

Estos campos solo están presentes cuando se define la macro *Py_TRACE_REFS*. Su inicialización a *NULL* se ocupa de la macro *PyObject_HEAD_INIT*. Para los objetos asignados estáticamente, estos campos siempre permanecen *NULL*. Para los objetos asignados dinámicamente, estos dos campos se utilizan para vincular el objeto en una lista doblemente vinculada de *todos* objetos vivos en el montón. Esto podría usarse para varios propósitos de depuración; Actualmente, el único uso es imprimir los objetos que aún están vivos al final de una ejecución cuando se establece la variable de entorno *PYTHONDUMPREFS*.

Herencia:

Estos campos no son heredados por subtipos.

Py_ssize_t **PyObject.ob_refcnt**

Este es el recuento de referencia del objeto tipo, inicializado a 1 por el macro *PyObject_HEAD_INIT*. Tenga en cuenta que para los objetos de tipo asignados estáticamente, las instancias del tipo (objetos cuyo *ob_type* apunta al tipo) *no* cuentan como referencias. Pero para los objetos de tipo asignados dinámicamente, las instancias *sí* cuentan como referencias.

Herencia:

Este campo no es heredado por los subtipos.

*PyTypeObject** **PyObject.ob_type**

Este es el tipo del tipo, en otras palabras, su metatipo. Se inicializa mediante el argumento de la macro `PyObject_HEAD_INIT`, y su valor normalmente debería ser `&PyType_Type`. Sin embargo, para los módulos de extensión cargables dinámicamente que deben ser utilizables en Windows (al menos), el compilador se queja de que este no es un inicializador válido. Por lo tanto, la convención es pasar `NULL` al macro `PyObject_HEAD_INIT` e inicializar este campo explícitamente al comienzo de la función de inicialización del módulo, antes de hacer cualquier otra cosa. Esto normalmente se hace así:

```
Foo_Type.ob_type = &PyType_Type;
```

Esto debe hacerse antes de que se creen instancias del tipo. `PyType_Ready()` comprueba si `ob_type` es `NULL`, y si es así, lo inicializa en el campo `ob_type` de la clase base. `PyType_Ready()` no cambiará este campo si no es cero.

Herencia:

Este campo es heredado por subtipos.

12.3.4 Ranuras PyVarObject

Py_ssize_t **PyVarObject.ob_size**

Para los objetos tipo asignados estáticamente, esto debe inicializarse a cero. Para los objetos tipo asignados dinámicamente, este campo tiene un significado interno especial.

Herencia:

Este campo no es heredado por los subtipos.

12.3.5 Ranuras PyTypeObject

Cada ranura tiene una sección que describe la herencia. Si `PyType_Ready()` puede establecer un valor cuando el campo se establece en `NULL`, entonces también habrá una sección «Predeterminada». (Tenga en cuenta que muchos campos establecidos en `PyBaseObject_Type` y `PyType_Type` actúan efectivamente como valores predeterminados).

const char* **PyTypeObject.tp_name**

Puntero a una cadena de caracteres terminada en `NULL` que contiene el nombre del tipo. Para los tipos que son accesibles como módulos globales, la cadena debe ser el nombre completo del módulo, seguido de un punto, seguido del nombre del tipo; para los tipos integrados, debe ser solo el nombre del tipo. Si el módulo es un submódulo de un paquete, el nombre completo del paquete es parte del nombre completo del módulo. Por ejemplo, un tipo llamado `T` definido en el módulo `M` en el subpaquete `Q` en el paquete `P` debe tener el inicializador `tp_name "PQMT"`.

Para los objetos tipo asignados dinámicamente, este debería ser solo el nombre del tipo, y el nombre del módulo almacenado explícitamente en el tipo diccionario (*dict*) como el valor para la clave `'__module__'`.

Para los objetos tipo asignados estáticamente, el campo `tp_name` debe contener un punto. Todo antes del último punto se hace accesible como el atributo `__module__`, y todo después del último punto se hace accesible como el atributo `__name__`.

Si no hay ningún punto, todo el campo `tp_name` se hace accesible como el atributo `__name__`, y el atributo `__module__` no está definido (a menos que sea explícitamente establecido en el diccionario, como se explicó anteriormente). Esto significa que su tipo será imposible de guardar como *pickle*. Además, no figurará en la documentación del módulo creado con *pydoc*.

Este campo no debe ser NULL. Es el único campo obligatorio en `PyObject()` (que no sea potencialmente `tp_itemsize`).

Herencia:

Este campo no es heredado por los subtipos.

Py_ssize_t `PyObject.tp_basicsize`

Py_ssize_t `PyObject.tp_itemsize`

Estos campos permiten calcular el tamaño en bytes de instancias del tipo.

Hay dos tipos de tipos: los tipos con instancias de longitud fija tienen un campo cero `tp_itemsize`, los tipos con instancias de longitud variable tienen un campo distinto de cero `tp_itemsize`. Para un tipo con instancias de longitud fija, todas las instancias tienen el mismo tamaño, dado en `tp_basicsize`.

Para un tipo con instancias de longitud variable, las instancias deben tener un campo `ob_size`, y el tamaño de la instancia es `tp_basicsize` más N veces `tp_itemsize`, donde N es la «longitud» del objeto. El valor de N generalmente se almacena en el campo `ob_size` de la instancia. Hay excepciones: por ejemplo, los `ints` usan un negativo `ob_size` para indicar un número negativo, y N es `abs(ob_size)` allí. Además, la presencia de un campo `ob_size` en el diseño de la instancia no significa que la estructura de la instancia sea de longitud variable (por ejemplo, la estructura para el tipo de lista tiene instancias de longitud fija, aunque esas instancias tienen un significativo campo `ob_size`).

El tamaño básico incluye los campos en la instancia declarada por el macro `PyObject_HEAD` o `PyObject_VAR_HEAD` (lo que se use para declarar la estructura de la instancia) y esto a su vez incluye campos `_ob_prev` y `_ob_next` si están presentes. Esto significa que la única forma correcta de obtener un inicializador para `tp_basicsize` es usar el operador `sizeof` en la estructura utilizada para declarar el diseño de la instancia. El tamaño básico no incluye el tamaño del encabezado del GC.

Una nota sobre la alineación: si los elementos variables requieren una alineación particular, esto debe ser atendido por el valor de `tp_basicsize`. Ejemplo: supongamos que un tipo implementa un arreglo de dobles (`double`). `tp_itemsize` es `sizeof(double)`. Es responsabilidad del programador que `tp_basicsize` es un múltiplo de `sizeof(double)` (suponiendo que este sea el requisito de alineación para `double`).

Para cualquier tipo con instancias de longitud variable, este campo no debe ser NULL.

Herencia:

Estos campos se heredan por separado por subtipos. Si el tipo base tiene un miembro distinto de cero `tp_itemsize`, generalmente no es seguro establecer `tp_itemsize` en un valor diferente de cero en un subtipo (aunque esto depende de la implementación del tipo base).

destructor `PyObject.tp_dealloc`

Un puntero a la función destructor de instancias. Esta función debe definirse a menos que el tipo garantice que sus instancias nunca se desasignarán (como es el caso de los singletons `None` y `Ellipsis`). La firma de la función es:

```
void tp_dealloc(PyObject *self);
```

La función destructor es llamada por las macros `Py_DECREF()` y `Py_XDECREF()` cuando el nuevo recuento de referencia es cero. En este punto, la instancia todavía existe, pero no hay referencias a ella. La función destructor debe liberar todas las referencias que posee la instancia, liberar todos los búferes de memoria que posee la instancia (utilizando la función de liberación correspondiente a la función de asignación utilizada para asignar el búfer) y llamar a los tipos función `tp_free`. Si el tipo no es subtipable (no tiene establecido el bit de indicador `Py_TPFLAGS_BASETYPE`), está permitido llamar al objeto desasignador directamente en lugar de a través de `tp_free`. El objeto desasignador debe ser el utilizado para asignar la instancia; normalmente es `PyObject_Del()` si la instancia se asignó usando `PyObject_New()` o `PyObject_VarNew()`, o `PyObject_GC_Del()` si la instancia se asignó usando `PyObject_GC_New()` o `PyObject_GC_NewVar()`.

If the type supports garbage collection (has the `Py_TPFLAGS_HAVE_GC` flag bit set), the destructor should call `PyObject_GC_UnTrack()` before clearing any member fields.

```
static void foo_dealloc(foo_object *self) {
    PyObject_GC_UnTrack(self);
    Py_CLEAR(self->ref);
    Py_TYPE(self)->tp_free((PyObject *)self);
}
```

Finalmente, si el tipo está asignado en el montón (`Py_TPFLAGS_HEAPTYPE`), el desasignador debería disminuir el conteo de referencia para su objeto tipo después de llamar al desasignador del tipo. Para evitar punteros colgantes, la forma recomendada de lograr esto es:

```
static void foo_dealloc(foo_object *self) {
    PyTypeObject *tp = Py_TYPE(self);
    // free references and buffers here
    tp->tp_free(self);
    Py_DECREF(tp);
}
```

Herencia:

Este campo es heredado por subtipos.

`Py_ssize_t PyTypeObject.tp_vectorcall_offset`

Un desplazamiento opcional a una función por instancia que implementa la llamada al objeto usando *vectorcall protocol*, una alternativa más eficiente del simple `tp_call`.

Este campo solo se usa si se establece el flag `Py_TPFLAGS_HAVE_VECTORCALL`. Si es así, debe ser un entero positivo que contenga el desplazamiento en la instancia de un puntero *vectorcallfunc*.

El puntero *vectorcallfunc* puede ser NULL, en cuyo caso la instancia se comporta como si `Py_TPFLAGS_HAVE_VECTORCALL` no estuviera configurado: llamar a la instancia vuelve a `tp_call`.

Cualquier clase que establezca `_Py_TPFLAGS_HAVE_VECTORCALL` también debe establecer `tp_call` y asegurarse de que su comportamiento sea coherente con la función *vectorcallfunc*. Esto se puede hacer configurando `tp_call` en `PyVectorcall_Call()`.

Advertencia: No se recomienda para *tipos de pila* para implementar el protocolo vectorcall. Cuando un usuario establece `__call__` en código Python, solo se actualiza `tp_call`, lo que probablemente lo haga inconsistente con la función vectorcall.

Nota: La semántica de la ranura `tp_vectorcall_offset` es provisional y se espera que finalice en Python 3.9. Si usa *vectorcall*, planifique actualizar su código para Python 3.9.

Distinto en la versión 3.8: Antes de la versión 3.8, este slot se llamaba `tp_print`. En Python 2.x, se usó para imprimir en un archivo. En Python 3.0 a 3.7, no se usó.

Herencia:

Este campo siempre se hereda. Sin embargo, el flag `Py_TPFLAGS_HAVE_VECTORCALL` no siempre se hereda. Si no es así, entonces la subclase no usará *vectorcall*, excepto cuando `PyVectorcall_Call()` se llame explícitamente. Este es en particular el caso de los *heap types* (incluidas las subclases definidas en Python).

getattrfunc `PyTypeObject.tp_getattr`

Un puntero opcional a la función «obtener atributo cadena de caracteres» (*get-attribute-string*).

Este campo está en desuso. Cuando se define, debe apuntar a una función que actúe igual que la función `tp_getattro`, pero tomando una cadena de caracteres C en lugar de un objeto de cadena Python para dar el nombre del atributo.

Herencia:

Grupo: `tp_getattr`, `tp_getattro`

Este campo es heredado por los subtipos junto con `tp_getattro`: un subtipo hereda ambos `tp_getattr` y `tp_getattro` de su base escriba cuando los subtipos `tp_getattr` y `tp_getattro` son ambos NULL.

setattrfunc **PyObject.tp_setattr**

Un puntero opcional a la función para configurar y eliminar atributos.

Este campo está en desuso. Cuando se define, debe apuntar a una función que actúe igual que la función `tp_setattro`, pero tomando una cadena de caracteres C en lugar de un objeto de cadena Python para dar el nombre del atributo.

Herencia:

Grupo: `tp_setattr`, `tp_setattro`

Este campo es heredado por los subtipos junto con `tp_setattro`: un subtipo hereda ambos `tp_setattr` y `tp_setattro` de su base escriba cuando los subtipos `tp_setattr` y `tp_setattro` son ambos NULL.

*PyAsyncMethods** **PyObject.tp_as_async**

Puntero a una estructura adicional que contiene campos relevantes solo para los objetos que implementan los protocolos «esperable» (*awaitable*) y «iterador asíncrono» (*asynchronous iterator*) en el nivel C. Ver *Estructuras de objetos asíncronos* para más detalles.

Nuevo en la versión 3.5: Anteriormente conocidos como `tp_compare` y `tp_reserved`.

Herencia:

El campo `tp_as_async` no se hereda, pero los campos contenidos se heredan individualmente.

reprfunc **PyObject.tp_repr**

Un puntero opcional a una función que implementa la función incorporada `repr()`.

La firma es la misma que para *PyObject_Repr()*:

```
PyObject *tp_repr(PyObject *self);
```

La función debe retornar una cadena de caracteres o un objeto Unicode. Idealmente, esta función debería retornar una cadena que, cuando se pasa a `eval()`, dado un entorno adecuado, retorna un objeto con el mismo valor. Si esto no es factible, debe retornar una cadena que comience con '`<`' y termine con '`>`' desde la cual se puede deducir tanto el tipo como el valor del objeto.

Herencia:

Este campo es heredado por subtipos.

Por defecto:

Cuando este campo no está configurado, se retorna una cadena de caracteres de la forma `<%s object at %p>`, donde `%s` se reemplaza por el nombre del tipo y `%p` por dirección de memoria del objeto.

*PyNumberMethods** **PyObject.tp_as_number**

Puntero a una estructura adicional que contiene campos relevantes solo para objetos que implementan el protocolo numérico. Estos campos están documentados en *Estructuras de Objetos de Números*.

Herencia:

El campo `tp_as_number` no se hereda, pero los campos contenidos se heredan individualmente.

***PySequenceMethods** PyObject.tp_as_sequence**

Puntero a una estructura adicional que contiene campos relevantes solo para objetos que implementan el protocolo de secuencia. Estos campos están documentados en *Estructuras de objetos secuencia*.

Herencia:

El campo `tp_as_sequence` no se hereda, pero los campos contenidos se heredan individualmente.

***PyMappingMethods** PyObject.tp_as_mapping**

Puntero a una estructura adicional que contiene campos relevantes solo para objetos que implementan el protocolo de mapeo. Estos campos están documentados en *Estructuras de Objetos Mapeo*.

Herencia:

El campo `tp_as_mapping` no se hereda, pero los campos contenidos se heredan individualmente.

***hashfunc* PyObject.tp_hash**

Un puntero opcional a una función que implementa la función incorporada `hash()`.

La firma es la misma que para `PyObject_Hash()`:

```
Py_hash_t tp_hash(PyObject *);
```

El valor `-1` no debe retornarse como un valor de retorno normal; Cuando se produce un error durante el cálculo del valor `hash`, la función debe establecer una excepción y retornar `-1`.

Cuando este campo no está establecido (y `tp_richcompare` no está establecido), se lanza `TypeError` cuando hay un intento de tomar el `hash` del objeto. Esto es lo mismo que establecerlo en `PyObject_HashNotImplemented()`.

Este campo se puede establecer explícitamente en `PyObject_HashNotImplemented()` para bloquear la herencia del método `hash` de un tipo primario. Esto se interpreta como el equivalente de `__hash__ = None` en el nivel de Python, lo que hace que `isinstance(o, collections.Hashable)` retorne correctamente `False`. Tenga en cuenta que lo contrario también es cierto: establecer `__hash__ = None` en una clase en el nivel de Python dará como resultado que la ranura `tp_hash` se establezca en `PyObject_HashNotImplemented()`.

Herencia:

Grupo: `tp_hash`, `tp_richcompare`

Este campo es heredado por subtipos junto con `tp_richcompare`: un subtipo hereda ambos `tp_richcompare` y `tp_hash`, cuando los subtipos `tp_richcompare` y `tp_hash` son ambos `NULL`.

***ternaryfunc* PyObject.tp_call**

Un puntero opcional a una función que implementa la llamada al objeto. Esto debería ser `NULL` si el objeto no es invocable. La firma es la misma que para `PyObject_Call()`:

```
PyObject *tp_call(PyObject *self, PyObject *args, PyObject *kwargs);
```

Herencia:

Este campo es heredado por subtipos.

***reprfunc* PyObject.tp_str**

Un puntero opcional a una función que implementa la operación integrada `str()`. (Tenga en cuenta que `str` es un tipo ahora, y `str()` llama al constructor para ese tipo. Este constructor llama a `PyObject_Str()` para hacer el trabajo real, y `PyObject_Str()` llamará a este controlador.)

La firma es la misma que para `PyObject_Str()`:

```
PyObject *tp_str(PyObject *self);
```

La función debe retornar una cadena de caracteres o un objeto Unicode. Debe ser una representación de cadena «amigable» del objeto, ya que esta es la representación que será utilizada, entre otras cosas, por la función `print()`.

Herencia:

Este campo es heredado por subtipos.

Por defecto:

Cuando este campo no está configurado, se llama a `PyObject_Repr()` para retornar una representación de cadena de caracteres.

getattrofunc **PyTypeObject.tp_getattro**

Un puntero opcional a la función «obtener atributo» (*get-attribute*).

La firma es la misma que para `PyObject_GetAttr()`:

```
PyObject *tp_getattro(PyObject *self, PyObject *attr);
```

Por lo general, es conveniente establecer este campo en `PyObject_GenericGetAttr()`, que implementa la forma normal de buscar atributos de objeto.

Herencia:

Grupo: `tp_getattr`, `tp_getattro`

Este campo es heredado por los subtipos junto con `tp_getattr`: un subtipo hereda ambos `tp_getattr` y `tp_getattro` de su base escriba cuando los subtipos `tp_getattr` y `tp_getattro` son ambos NULL.

Por defecto:

`PyBaseObject_Type` usa `PyObject_GenericGetAttr()`.

setattrofunc **PyTypeObject.tp_setattro**

Un puntero opcional a la función para configurar y eliminar atributos.

La firma es la misma que para `PyObject_SetAttr()`:

```
int tp_setattro(PyObject *self, PyObject *attr, PyObject *value);
```

Además, se debe admitir la configuración de *value* en NULL para eliminar un atributo. Por lo general, es conveniente establecer este campo en `PyObject_GenericSetAttr()`, que implementa la forma normal de establecer los atributos del objeto.

Herencia:

Grupo: `tp_setattr`, `tp_setattro`

Los subtipos heredan este campo junto con `tp_setattr`: un subtipo hereda ambos `tp_setattr` y `tp_setattro` de su base escriba cuando los subtipos `tp_setattr` y `tp_setattro` son ambos NULL.

Por defecto:

`PyBaseObject_Type` usa `PyObject_GenericSetAttr()`.

*PyBufferProcs** **PyTypeObject.tp_as_buffer**

Puntero a una estructura adicional que contiene campos relevantes solo para objetos que implementan la interfaz del búfer. Estos campos están documentados en *Estructuras de Objetos Búfer*.

Herencia:

El campo `tp_as_buffer` no se hereda, pero los campos contenidos se heredan individualmente.

unsigned long `PyObject.tp_flags`

Este campo es una máscara de bits de varias banderas. Algunas banderas indican semántica variante para ciertas situaciones; otros se utilizan para indicar que ciertos campos en el tipo de objeto (o en las estructuras de extensión a las que se hace referencia a través de `tp_as_number`, `tp_as_sequence`, `tp_as_mapping`, y `tp_as_buffer`) que históricamente no siempre estuvieron presentes son válidos; si dicho bit de bandera está claro, no se debe acceder a los campos de tipo que protege y se debe considerar que tienen un valor cero o NULL.

Herencia:

La herencia de este campo es complicada. La mayoría de los bits de bandera se heredan individualmente, es decir, si el tipo base tiene un conjunto de bits de bandera, el subtipo hereda este bit de bandera. Los bits de bandera que pertenecen a las estructuras de extensión se heredan estrictamente si la estructura de extensión se hereda, es decir, el valor del tipo base del bit de bandera se copia en el subtipo junto con un puntero a la estructura de extensión. El bit de bandera `Py_TPFLAGS_HAVE_GC` se hereda junto con `tp_traverse` y `tp_clear`, es decir, si el bit de bandera `Py_TPFLAGS_HAVE_GC` está claro en el subtipo y los campos `tp_traverse` y `tp_clear` en el subtipo existen y tienen valores NULL.

Por defecto:

`PyBaseObject_Type` usa `Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE`.

Máscaras de bits:

Las siguientes máscaras de bits están definidas actualmente; estos se pueden unir por *OR* usando el operador `|` para formar el valor del campo `tp_flags`. El macro `PyType_HasFeature()` toma un tipo y un valor de banderas, `tp` y `f`, y comprueba si `tp->tp_flags & f` no es cero.

`Py_TPFLAGS_HEAPTYPE`

Este bit se establece cuando el objeto de tipo se asigna en el montón, por ejemplo, los tipos creados dinámicamente usando `PyType_FromSpec()`. En este caso, el campo `ob_type` de sus instancias se considera una referencia al tipo, y el objeto de tipo se llama *INCREf* cuando se crea una nueva instancia, y *DECREf* cuando se destruye una instancia (esto hace no se aplica a instancias de subtipos; solo el tipo al que hace referencia el `ob_type` de la instancia obtiene *INCREf* o *DECREf*).

Herencia:

???

`Py_TPFLAGS_BASETYPE`

Este bit se establece cuando el tipo se puede usar como el tipo base de otro tipo. Si este bit es claro, el tipo no puede subtiparse (similar a una clase «final» en Java).

Herencia:

???

`Py_TPFLAGS_READY`

Este bit se establece cuando el objeto tipo ha sido completamente inicializado por `PyType_Ready()`.

Herencia:

???

`Py_TPFLAGS_READYING`

Este bit se establece mientras `PyType_Ready()` está en el proceso de inicialización del objeto tipo.

Herencia:

???

`Py_TPFLAGS_HAVE_GC`

Este bit se establece cuando el objeto admite la recolección de elementos no utilizados. Si se establece este bit, las instancias deben crearse usando `PyObject_GC_New()` y destruirse usando `PyObject_GC_Del()`.

Más información en la sección *Apoyo a la recolección de basura cíclica*. Este bit también implica que los campos relacionados con GC `tp_traverse` y `tp_clear` están presentes en el objeto de tipo.

Herencia:

Grupo: `Py_TPFLAGS_HAVE_GC`, `tp_traverse`, `tp_clear`

El bit de indicador `Py_TPFLAGS_HAVE_GC` se hereda junto con los campos `tp_traverse` y `tp_clear`, es decir, si el bit de indicador `Py_TPFLAGS_HAVE_GC` está claro en el subtipo y los campos `tp_traverse` y `tp_clear` en el subtipo existen y tienen valores NULL.

Py_TPFLAGS_DEFAULT

Esta es una máscara de bits de todos los bits que pertenecen a la existencia de ciertos campos en el objeto tipo y sus estructuras de extensión. Actualmente, incluye los siguientes bits: `Py_TPFLAGS_HAVE_STACKLESS_EXTENSION`, `Py_TPFLAGS_HAVE_VERSION_TAG`.

Herencia:

???

Py_TPFLAGS_METHOD_DESCRIPTOR

Este bit indica que los objetos se comportan como métodos independientes.

Si este indicador está configurado para `type(meth)`, entonces:

- `meth.__get__(obj, cls)(*args, **kwds)` (con `obj` no `None`) debe ser equivalente a `meth(obj, *args, **kwds)`.
- `meth.__get__(None, cls)(*args, **kwds)` debe ser equivalente a `meth(*args, **kwds)`.

Este indicador (*flag*) permite una optimización para llamadas a métodos típicos como `obj.meth()`: evita crear un objeto temporal de «método vinculado» para `obj.meth`.

Nuevo en la versión 3.8.

Herencia:

Este indicador (*flag*) nunca es heredada por los tipos de montón. Para los tipos de extensión, se hereda siempre que `tp_descr_get` se hereda.

Py_TPFLAGS_LONG_SUBCLASS**Py_TPFLAGS_LIST_SUBCLASS****Py_TPFLAGS_TUPLE_SUBCLASS****Py_TPFLAGS_BYTES_SUBCLASS****Py_TPFLAGS_UNICODE_SUBCLASS****Py_TPFLAGS_DICT_SUBCLASS****Py_TPFLAGS_BASE_EXC_SUBCLASS****Py_TPFLAGS_TYPE_SUBCLASS**

Estas marcas son utilizadas por funciones como `PyLong_Check()` para determinar rápidamente si un tipo es una subclase de un tipo incorporado; dichos controles específicos son más rápidos que un control genérico, como `PyObject_IsInstance()`. Los tipos personalizados que heredan de los elementos integrados deben tener su `tp_flags` configurado correctamente, o el código que interactúa con dichos tipos se comportará de manera diferente dependiendo del tipo de verificación que se use.

Py_TPFLAGS_HAVE_FINALIZE

Este bit se establece cuando la ranura `tp_finalize` está presente en la estructura de tipo.

Nuevo en la versión 3.4.

Obsoleto desde la versión 3.8: Este indicador ya no es necesario, ya que el intérprete asume que: el espacio `tp_finalize` siempre está presente en la estructura de tipos.

Py_TPFLAGS_HAVE_VECTORCALL

Este bit se establece cuando la clase implementa *protocolo vectorcall*. Consulte `tp_vectorcall_offset` para obtener más detalles.

Herencia:

Este bit se hereda para subtipos *static* si `tp_call` también se hereda. *Heap types* no heredan `Py_TPFLAGS_HAVE_VECTORCALL`.

Nuevo en la versión 3.9.

const char* PyObject.tp_doc

Un puntero opcional a una cadena de caracteres de C terminada en NULL que proporciona la cadena de documentación para este tipo de objeto. Esto se expone como el atributo `__doc__` en el tipo y las instancias del tipo.

Herencia:

Este campo es *no* heredado por los subtipos.

traverseproc **PyObject.tp_traverse**

Un puntero opcional a una función transversal para el recolector de basura. Esto solo se usa si se establece el bit de la bandera (*flag*) `Py_TPFLAGS_HAVE_GC`. La firma es:

```
int tp_traverse(PyObject *self, visitproc visit, void *arg);
```

Se puede encontrar más información sobre el esquema de recolección de basura de Python en la sección *Apoyo a la recolección de basura cíclica*.

El puntero `tp_traverse` es utilizado por el recolector de basura para detectar ciclos de referencia. Una implementación típica de un `tp_traverse` simplemente llama a `Py_VISIT()` en cada uno de los miembros de la instancia que son objetos de Python que posee la instancia. Por ejemplo, esta es la función `local_traverse()` del módulo de extensión `_thread`:

```
static int
local_traverse(localobject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->args);
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

Tenga en cuenta que `Py_VISIT()` solo se llama a aquellos miembros que pueden participar en los ciclos de referencia. Aunque también hay un miembro `self->key`, solo puede ser NULL o una cadena de caracteres de Python y, por lo tanto, no puede ser parte de un ciclo de referencia.

Por otro lado, incluso si sabe que un miembro nunca puede ser parte de un ciclo, como ayuda para la depuración puede visitarlo de todos modos solo para que la función `get_referents()` del módulo `gc` lo incluya.

Advertencia: Al implementar `tp_traverse`, solo se deben visitar los miembros que *posee* la instancia (al tener fuertes referencias a ellos). Por ejemplo, si un objeto admite referencias débiles a través de la ranura `tp_weaklist`, el puntero que admite la lista vinculada (a lo que `tp_weaklist` señala) **no** debe ser visitado como la instancia no posee directamente las referencias débiles a sí mismo (la lista de referencias débiles está ahí para admitir la maquinaria de referencia débil, pero la instancia no tiene una referencia fuerte a los elementos dentro de ella, ya que se puede eliminar incluso si la instancia aún está viva).

Tenga en cuenta que `Py_VISIT()` requiere los parámetros `visit` y `arg` para `local_traverse()` para tener estos nombres específicos; no les llames de ninguna manera.

Los tipos asignados al heap (`Py_TPFLAGS_HEAPTYPE`, como los creados con `PyType_FromSpec()` y API similares) contienen una referencia a su tipo. Por lo tanto, su función transversal debe visitar `Py_TYPE(self)`, o delegar esta responsabilidad llamando a `tp_traverse` de otro tipo asignado al heap (como una superclase asignada al heap). Si no es así, es posible que el objeto de tipo no se recolecte como basura.

Distinto en la versión 3.9: Se espera que los tipos asignados al montón visiten `Py_TYPE(self)` en `tp_traverse`. En versiones anteriores de Python, debido al [bug 40217](#), hacer esto puede provocar fallas en las subclases.

Herencia:

Grupo: `Py_TPFLAGS_HAVE_GC`, `tp_traverse`, `tp_clear`

Este campo es heredado por los subtipos junto con `tp_clear` y el `Py_TPFLAGS_HAVE_GC` bit de bandera: el bit de bandera, `tp_traverse`, y `tp_clear` se heredan todos del tipo base si todos son cero en el subtipo.

inquiry `PyTypeObject.tp_clear`

Un puntero opcional a una función de limpieza (*clear function*) para el recolector de basura. Esto solo se usa si se establece el bit de bandera `Py_TPFLAGS_HAVE_GC`. La firma es:

```
int tp_clear(PyObject *);
```

La función miembro `tp_clear` se usa para romper los ciclos de referencia en la basura cíclica detectada por el recolector de basura. En conjunto, todas las funciones `tp_clear` en el sistema deben combinarse para romper todos los ciclos de referencia. Esto es sutil y, en caso de duda, proporcione una función `tp_clear`. Por ejemplo, el tipo de tupla no implementa una función `tp_clear`, porque es posible demostrar que ningún ciclo de referencia puede estar compuesto completamente de tuplas. Por lo tanto, las funciones `tp_clear` de otros tipos deben ser suficientes para romper cualquier ciclo que contenga una tupla. Esto no es inmediatamente obvio, y rara vez hay una buena razón para evitar la implementación de `tp_clear`.

Las implementaciones de `tp_clear` deberían descartar las referencias de la instancia a las de sus miembros que pueden ser objetos de Python, y establecer sus punteros a esos miembros en NULL, como en el siguiente ejemplo:

```
static int
local_clear(localobject *self)
{
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```

Se debe utilizar el macro `Py_CLEAR()`, porque borrar las referencias es delicado: la referencia al objeto contenido no se debe disminuir hasta después de que el puntero al objeto contenido se establezca en NULL. Esto se debe a que la disminución del conteo de referencias puede hacer que el objeto contenido se convierta en basura, lo que desencadena una cadena de actividad de recuperación que puede incluir la invocación de código arbitrario de Python (debido a finalizadores o devoluciones de llamada de reflujo débil, asociadas con el objeto contenido). Si es posible que dicho código haga referencia a *self* nuevamente, es importante que el puntero al objeto contenido sea NULL en ese momento, de modo que *self* sepa que el objeto contenido ya no se puede usar. El macro `Py_CLEAR()` realiza las operaciones en un orden seguro.

Note that `tp_clear` is not *always* called before an instance is deallocated. For example, when reference counting is enough to determine that an object is no longer used, the cyclic garbage collector is not involved and `tp_dealloc` is called directly.

Debido a que el objetivo de `tp_clear` es romper los ciclos de referencia, no es necesario borrar objetos contenidos como cadenas de caracteres de Python o enteros de Python, que no pueden participar en los ciclos de referencia. Por otro lado, puede ser conveniente borrar todos los objetos Python contenidos y escribir la función `tp_dealloc` para invocar `tp_clear`.

Se puede encontrar más información sobre el esquema de recolección de basura de Python en la sección [Apoyo a la recolección de basura cíclica](#).

Herencia:

Grupo: `Py_TPFLAGS_HAVE_GC`, `tp_traverse`, `tp_clear`

Este campo es heredado por subtipos junto con `tp_traverse` y el `Py_TPFLAGS_HAVE_GC` bit de bandera: el bit de bandera, `tp_traverse`, y `tp_clear` se heredan todos del tipo base si todos son cero en el subtipo.

richcmpfunc `PyObject.tp_richcompare`

Un puntero opcional a la función de comparación enriquecida, cuya firma es:

```
PyObject *tp_richcompare(PyObject *self, PyObject *other, int op);
```

Se garantiza que el primer parámetro será una instancia del tipo definido por `PyObject`.

La función debería retornar el resultado de la comparación (generalmente `Py_True` o `Py_False`). Si la comparación no está definida, debe retornar `Py_NotImplemented`, si se produce otro error, debe retornar `NULL` y establecer una condición de excepción.

Las siguientes constantes se definen para ser utilizadas como el tercer argumento para `tp_richcompare` y para `PyObject_RichCompare()`:

Constante	Comparación
<code>Py_LT</code>	<
<code>Py_LE</code>	<=
<code>Py_EQ</code>	==
<code>Py_NE</code>	!=
<code>Py_GT</code>	>
<code>Py_GE</code>	>=

El siguiente macro está definido para facilitar la escritura de funciones de comparación enriquecidas:

`Py_RETURN_RICHCOMPARE` (VAL_A, VAL_B, op)

Retorna `Py_True` o `Py_False` de la función, según el resultado de una comparación. `VAL_A` y `VAL_B` deben ser ordenados por operadores de comparación C (por ejemplo, pueden ser enteros o punto flotantes de C). El tercer argumento especifica la operación solicitada, como por ejemplo `PyObject_RichCompare()`.

El conteo de referencia del valor de retorno se incrementa correctamente.

En caso de error, establece una excepción y retorna `NULL` de la función.

Nuevo en la versión 3.7.

Herencia:

Grupo: `tp_hash`, `tp_richcompare`

Este campo es heredado por subtipos junto con `tp_hash`: un subtipo hereda `tp_richcompare` y `tp_hash` cuando el subtipo `tp_richcompare` y `tp_hash` son ambos `NULL`.

Por defecto:

`PyBaseObject_Type` proporciona una implementación `tp_richcompare`, que puede ser heredada. Sin embargo, si solo se define `tp_hash`, ni siquiera se utiliza la función heredada y las instancias del tipo no podrán participar en ninguna comparación.

***Py_ssize_t* PyObject.tp_weaklistoffset**

Si las instancias de este tipo son débilmente referenciables, este campo es mayor que cero y contiene el desplazamiento en la estructura de instancia del encabezado de la lista de referencia débil (ignorando el encabezado GC, si está presente); este desplazamiento es utilizado por `PyObject_ClearWeakRefs()` y las funciones `PyWeakref_*()`. La estructura de la instancia debe incluir un campo de tipo `PyObject*` que se inicializa en `NULL`.

No confunda este campo con `tp_weaklist`; ese es el encabezado de la lista para referencias débiles al objeto de tipo en sí.

Herencia:

Este campo es heredado por subtipos, pero consulte las reglas que se enumeran a continuación. Un subtipo puede anular este desplazamiento; Esto significa que el subtipo utiliza un encabezado de lista de referencia débil diferente que el tipo base. Dado que el encabezado de la lista siempre se encuentra a través de `tp_weaklistoffset`, esto no debería ser un problema.

Cuando un tipo definido por una declaración de clase no tiene `__slots__` declaración, y ninguno de sus tipos base es débilmente referenciable, el tipo se hace débilmente referenciable al agregar una ranura de encabezado de lista de referencia débil al diseño de la instancia y configurando `tp_weaklistoffset` del desplazamiento de esa ranura.

Cuando la declaración de un tipo `__slots__` contiene un espacio llamado `__weakref__`, ese espacio se convierte en el encabezado de la lista de referencia débil para las instancias del tipo, y el desplazamiento del espacio se almacena en el tipo `tp_weaklistoffset`.

Cuando la declaración de un tipo `__slots__` no contiene un espacio llamado `__weakref__`, el tipo hereda su `tp_weaklistoffset` de su tipo base.

***getterfunc* PyObject.tp_iter**

Un puntero opcional a una función que retorna un iterador para el objeto. Su presencia normalmente indica que las instancias de este tipo son iterables (aunque las secuencias pueden ser iterables sin esta función).

Esta función tiene la misma firma que `PyObject_GetIter()`:

```
PyObject *tp_iter(PyObject *self);
```

Herencia:

Este campo es heredado por subtipos.

***iternextfunc* PyObject.tp_iternext**

Un puntero opcional a una función que retorna el siguiente elemento en un iterador. La firma es:

```
PyObject *tp_iternext(PyObject *self);
```

Cuando el iterador está agotado, debe retornar `NULL`; a la excepción `StopIteration` puede o no establecerse. Cuando se produce otro error, también debe retornar `NULL`. Su presencia indica que las instancias de este tipo son iteradores.

Los tipos de iterador también deberían definir la función `tp_iter`, y esa función debería retornar la instancia de iterador en sí (no una nueva instancia de iterador).

Esta función tiene la misma firma que `PyIter_Next()`.

Herencia:

Este campo es heredado por subtipos.

struct *PyMethodDef PyObject.tp_methods**

Un puntero opcional a un arreglo estático terminado en `NULL` de estructuras `PyMethodDef`, declarando métodos regulares de este tipo.

Para cada entrada en el arreglo, se agrega una entrada al diccionario del tipo (ver *tp_dict* a continuación) que contiene un descriptor *method*.

Herencia:

Los subtipos no heredan este campo (los métodos se heredan mediante un mecanismo diferente).

struct *PyMemberDef** **PyObject.tp_members**

Un puntero opcional a un arreglo estático terminado en NULL de estructuras *PyMemberDef*, declarando miembros de datos regulares (campos o ranuras) de instancias de este tipo.

Para cada entrada en el arreglo, se agrega una entrada al diccionario del tipo (ver *tp_dict* a continuación) que contiene un descriptor *member*.

Herencia:

Los subtipos no heredan este campo (los miembros se heredan mediante un mecanismo diferente).

struct *PyGetSetDef** **PyObject.tp_getset**

Un puntero opcional a un arreglo estático terminado en NULL de estructuras *PyGetSetDef*, declarando atributos calculados de instancias de este tipo.

Para cada entrada en el arreglo, se agrega una entrada al diccionario del tipo (ver *tp_dict* a continuación) que contiene un descriptor *getset*.

Herencia:

Este campo no es heredado por los subtipos (los atributos computados se heredan a través de un mecanismo diferente).

*PyObject** **PyObject.tp_base**

Un puntero opcional a un tipo base del que se heredan las propiedades de tipo. En este nivel, solo se admite una herencia única; La herencia múltiple requiere la creación dinámica de un objeto tipo llamando al metatipo.

Nota: La inicialización de ranuras está sujeta a las reglas de inicialización de globales. C99 requiere que los inicializadores sean «constantes de dirección». Los designadores de funciones como *PyType_GenericNew()*, con conversión implícita a un puntero, son constantes de dirección C99 válidas.

Sin embargo, el operador unario “&” aplicado a una variable no estática como *PyBaseObject_Type()* no es necesario para producir una dirección constante. Los compiladores pueden admitir esto (gcc lo hace), MSVC no. Ambos compiladores son estrictamente estándar conforme a este comportamiento particular.

En consecuencia, *tp_base* debe establecerse en la función *init* del módulo de extensión.

Herencia:

Este campo no es heredado por los subtipos (obviamente).

Por defecto:

Este campo predeterminado es *&PyBaseObject_Type* (que para los programadores de Python se conoce como el tipo objeto).

*PyObject** **PyObject.tp_dict**

El diccionario del tipo se almacena aquí por *PyType_Ready()*.

Este campo normalmente debe inicializarse a NULL antes de llamar a *PyType_Ready*; también se puede inicializar en un diccionario que contiene atributos iniciales para el tipo. Una vez *PyType_Ready()* ha inicializado el tipo, los atributos adicionales para el tipo pueden agregarse a este diccionario solo si no corresponden a operaciones sobrecargadas (como *__add__()*).

Herencia:

Este campo no es heredado por los subtipos (aunque los atributos definidos aquí se heredan a través de un mecanismo diferente).

Por defecto:

Si este campo es NULL, `PyType_Ready()` le asignará un nuevo diccionario.

Advertencia: No es seguro usar `PyDict_SetItem()` en o modificar de otra manera a `tp_dict` con el diccionario C-API.

descrgetfunc **PyTypeObject.tp_descr_get**

Un puntero opcional a una función «obtener descriptor» (*descriptor ger*).

La firma de la función es:

```
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject *type);
```

Herencia:

Este campo es heredado por subtipos.

descrsetfunc **PyTypeObject.tp_descr_set**

Un puntero opcional a una función para configurar y eliminar el valor de un descriptor.

La firma de la función es:

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

El argumento *value* se establece a NULL para borrar el valor.

Herencia:

Este campo es heredado por subtipos.

Py_ssize_t **PyTypeObject.tp_dictoffset**

Si las instancias de este tipo tienen un diccionario que contiene variables de instancia, este campo no es cero y contiene el desplazamiento en las instancias del tipo del diccionario de variables de instancia; este desplazamiento es utilizado por `PyObject_GenericGetAttr()`.

No confunda este campo con `tp_dict`; ese es el diccionario para los atributos del tipo de objeto en sí.

Si el valor de este campo es mayor que cero, especifica el desplazamiento desde el inicio de la estructura de la instancia. Si el valor es menor que cero, especifica el desplazamiento desde el *end* de la estructura de la instancia. Un desplazamiento negativo es más costoso de usar y solo debe usarse cuando la estructura de la instancia contiene una parte de longitud variable. Esto se utiliza, por ejemplo, para agregar un diccionario de variables de instancia a los subtipos de `str` o `tuple`. Tenga en cuenta que el campo `tp_basicsize` debe tener en cuenta el diccionario agregado al final en ese caso, aunque el diccionario no esté incluido en el diseño básico del objeto. En un sistema con un tamaño de puntero de 4 bytes, `tp_dictoffset` debe establecerse en -4 para indicar que el diccionario está al final de la estructura.

El desplazamiento real del diccionario en una instancia se puede calcular a partir de un elemento negativo `tp_dictoffset` de la siguiente manera:

```
dictoffset = tp_basicsize + abs(ob_size)*tp_itemsize + tp_dictoffset
if dictoffset is not aligned on sizeof(void*):
    round up to sizeof(void*)
```

donde `tp_basicsize`, `tp_itemsize` y `tp_dictoffset` se toman del objeto *type*, y `ob_size` está tomado de la instancia. Se toma el valor absoluto porque *ints* usa el signo de `ob_size` para almace-

nar el signo del número. (Nunca es necesario hacer este cálculo usted mismo; lo hace por usted la función `_PyObject_GetDictPtr()`.)

Herencia:

Este campo es heredado por subtipos, pero consulte las reglas que se enumeran a continuación. Un subtipo puede anular este desplazamiento; Esto significa que las instancias de subtipo almacenan el diccionario en un desplazamiento de diferencia que el tipo base. Dado que el diccionario siempre se encuentra a través de `tp_dictoffset`, esto no debería ser un problema.

Cuando un tipo definido por una declaración de clase no tiene `__slots__` declaración, y ninguno de sus tipos base tiene un diccionario de variable de instancia, se agrega un espacio de diccionario al diseño de la instancia y el `tp_dictoffset` está configurado para el desplazamiento de esa ranura.

Cuando un tipo definido por una declaración de clase tiene una declaración `__slots__`, el tipo hereda su `tp_dictoffset` de su tipo base.

(Agrega un espacio llamado `__dict__` a la declaración `__slots__` no tiene el efecto esperado, solo causa confusión. Quizás esto debería agregarse como una característica como `__weakref__` aunque.)

Por defecto:

Esta ranura no tiene valor predeterminado. Para los tipos estáticos, si el campo es NULL, entonces no `__dict__` se crea para las instancias.

initproc `PyTypeObject.tp_init`

Un puntero opcional a una función de inicialización de instancia.

Esta función corresponde al método de clases `__init__()`. Como `__init__()`, es posible crear una instancia sin llamar a `__init__()`, y es posible reinicializar una instancia llamando de nuevo a su método `__init__()`.

La firma de la función es:

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwargs);
```

El argumento propio es la instancia que se debe inicializar; los argumentos `args` y `kwargs` representan argumentos posicionales y de palabras clave de la llamada a `__init__()`.

La función `tp_init`, si no es NULL, se llama cuando una instancia se crea normalmente llamando a su tipo, después de la función `tp_new` del tipo ha retornado una instancia del tipo. Si la función `tp_new` retorna una instancia de otro tipo que no es un subtipo del tipo original, no se llama la función `tp_init`; if `tp_new` retorna una instancia de un subtipo del tipo original, se llama al subtipo `tp_init`.

Retorna 0 en caso de éxito, -1 y establece una excepción en caso de error.

Herencia:

Este campo es heredado por subtipos.

Por defecto:

Para los tipos estáticos, este campo no tiene un valor predeterminado.

allocfunc `PyTypeObject.tp_alloc`

Un puntero opcional a una función de asignación de instancia.

La firma de la función es:

```
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems);
```

Herencia:

Este campo es heredado por subtipos estáticos, pero no por subtipos dinámicos (subtipos creados por una declaración de clase).

Por defecto:

Para subtipos dinámicos, este campo siempre se establece en `PyType_GenericAlloc()`, para forzar una estrategia de asignación de montón estándar.

Para subtipos estáticos, `PyBaseObject_Type` utiliza `PyType_GenericAlloc()`. Ese es el valor recomendado para todos los tipos definidos estáticamente.

***newfunc* `PyTypeObject.tp_new`**

Un puntero opcional a una función de creación de instancias.

La firma de la función es:

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwargs);
```

El argumento *subtype* es el tipo de objeto que se está creando; los argumentos *args* y *kwargs* representan argumentos posicionales y de palabras clave de la llamada al tipo. Tenga en cuenta que *subtype* no tiene que ser igual al tipo cuya función *tp_new* es llamada; puede ser un subtipo de ese tipo (pero no un tipo no relacionado).

La función *tp_new* debería llamar a `subtype->tp_alloc(subtype, nitems)` para asignar espacio para el objeto, y luego hacer solo la inicialización adicional que sea absolutamente necesaria. La inicialización que se puede ignorar o repetir de forma segura debe colocarse en el controlador *tp_init*. Una buena regla general es que para los tipos inmutables, toda la inicialización debe tener lugar en *tp_new*, mientras que para los tipos mutables, la mayoría de las inicializaciones se deben diferir a *tp_init*.

Herencia:

Este campo es heredado por subtipos, excepto que no es heredado por tipos estáticos cuyo *tp_base* es `NULL` o `&PyBaseObject_Type`.

Por defecto:

Para los tipos estáticos, este campo no tiene valor predeterminado. Esto significa que si el espacio se define como `NULL`, no se puede llamar al tipo para crear nuevas instancias; presumiblemente hay otra forma de crear instancias, como una función de fábrica.

***freefunc* `PyTypeObject.tp_free`**

Un puntero opcional a una función de desasignación de instancia. Su firma es:

```
void tp_free(void *self);
```

Un inicializador que es compatible con esta firma es `PyObject_Free()`.

Herencia:

Este campo es heredado por subtipos estáticos, pero no por subtipos dinámicos (subtipos creados por una declaración de clase)

Por defecto:

En los subtipos dinámicos, este campo se establece en un desasignador adecuado para que coincida con `PyType_GenericAlloc()` y el valor del bit de bandera `Py_TPFLAGS_HAVE_GC`.

Para subtipos estáticos, `PyBaseObject_Type` usa `PyObject_Del`.

***inquiry* `PyTypeObject.tp_is_gc`**

Un puntero opcional a una función llamada por el recolector de basura.

El recolector de basura necesita saber si un objeto en particular es coleccionable o no. Normalmente, es suficiente mirar el campo *tp_flags* del tipo objeto, y verificar el bit de bandera `Py_TPFLAGS_HAVE_GC`. Pero algunos tipos tienen una mezcla de instancias asignadas estáticamente y dinámicamente, y las instancias asignadas estáticamente no son coleccionables. Tales tipos deberían definir esta función; debería retornar 1 para una instancia coleccionable y 0 para una instancia no coleccionable. La firma es:

```
int tp_is_gc(PyObject *self);
```

(El único ejemplo de esto son los mismo tipos. El metatipo, `PyType_Type`, define esta función para distinguir entre tipos asignados estáticamente y dinámicamente.)

Herencia:

Este campo es heredado por subtipos.

Por defecto:

Esta ranura no tiene valor predeterminado. Si este campo es NULL, se utiliza `Py_TPFLAGS_HAVE_GC` como el equivalente funcional.

*PyObject** **PyTypeObject.tp_bases**

Tupla de tipos base.

Esto se establece para los tipos creados por una declaración de clase. Debería ser NULL para los tipos estáticamente definidos.

Herencia:

Este campo no se hereda.

*PyObject** **PyTypeObject.tp_mro**

Tupla que contiene el conjunto expandido de tipos base, comenzando con el tipo en sí y terminando con `object`, en orden de resolución de método.

Herencia:

Este campo no se hereda; se calcula fresco por `PyType_Ready()`.

*PyObject** **PyTypeObject.tp_cache**

No usado. Solo para uso interno.

Herencia:

Este campo no se hereda.

*PyObject** **PyTypeObject.tp_subclasses**

Lista de referencias débiles a subclases. Solo para uso interno.

Herencia:

Este campo no se hereda.

*PyObject** **PyTypeObject.tp_weaklist**

Cabecera de lista de referencia débil, para referencias débiles a este tipo de objeto. No heredado Solo para uso interno.

Herencia:

Este campo no se hereda.

destructor **PyTypeObject.tp_del**

Este campo está en desuso. Use `tp_finalize` en su lugar.

unsigned int **PyTypeObject.tp_version_tag**

Se usa para indexar en el caché de métodos. Solo para uso interno.

Herencia:

Este campo no se hereda.

destructor **PyTypeObject.tp_finalize**

Un puntero opcional a una función de finalización de instancia. Su firma es:

```
void tp_finalize(PyObject *self);
```

Si `tp_finalize` está configurado, el intérprete lo llama una vez cuando finaliza una instancia. Se llama desde el recolector de basura (si la instancia es parte de un ciclo de referencia aislado) o justo antes de que el objeto se desasigne. De cualquier manera, se garantiza que se invocará antes de intentar romper los ciclos de referencia, asegurando que encuentre el objeto en un estado sano.

`tp_finalize` no debe mutar el estado de excepción actual; por lo tanto, una forma recomendada de escribir un finalizador no trivial es:

```
static void
local_finalize(PyObject *self)
{
    PyObject *error_type, *error_value, *error_traceback;

    /* Save the current exception, if any. */
    PyErr_Fetch(&error_type, &error_value, &error_traceback);

    /* ... */

    /* Restore the saved exception. */
    PyErr_Restore(error_type, error_value, error_traceback);
}
```

Para que este campo se tenga en cuenta (incluso a través de la herencia), también debe establecer el bit de banderas `Py_TPFLAGS_HAVE_FINALIZE`.

Además, tenga en cuenta que, en un Python que ha recolectado basura, se puede llamar a `tp_dealloc` desde cualquier hilo de Python, no solo el hilo que creó el objeto (si el objeto se convierte en parte de un ciclo de conteo de referencias, ese ciclo puede ser recogido por una recolección de basura en cualquier hilo). Esto no es un problema para las llamadas a la API de Python, ya que el hilo en el que se llama `tp_dealloc` será el propietario del Bloqueo Global del Intérprete (GIL, por sus siglas en inglés *Global Interpreter Lock*). Sin embargo, si el objeto que se destruye a su vez destruye objetos de alguna otra biblioteca C o C++, se debe tener cuidado para garantizar que la destrucción de esos objetos en el hilo que se llama `tp_dealloc` no violará ningún supuesto de la biblioteca.

Herencia:

Este campo es heredado por subtipos.

Nuevo en la versión 3.4.

Ver también:

«Finalización segura de objetos» ([PEP 442](#))

vectorcallfunc `PyTypeObject.tp_vectorcall`

Función `Vectorcall` a utilizar para llamadas de este tipo de objeto. En otras palabras, se usa para implementar *vectorcall* para `type.__call__`. Si `tp_vectorcall` es `NULL`, se usa la implementación de llamada predefinida usando `__new__` y `__init__`.

Herencia:

Este campo nunca se hereda.

Nuevo en la versión 3.9: (el campo existe desde 3.8 pero solo se usa desde 3.9)

12.3.6 Tipos Montículos (*Heap Types*)

Tradicionalmente, los tipos definidos en el código C son *static*, es decir, una estructura estática `PyTypeObject` se define directamente en el código y se inicializa usando `PyType_Ready()`.

Esto da como resultado tipos que están limitados en relación con los tipos definidos en Python:

- Los tipos estáticos están limitados a una base, es decir, no pueden usar herencia múltiple.
- Los objetos de tipo estático (pero no necesariamente sus instancias) son inmutables. No es posible agregar o modificar los atributos del objeto tipo desde Python.
- Los objetos de tipo estático se comparten en *sub intérpretes*, por lo que no deben incluir ningún estado específico del sub intérprete.

Además, dado que `PyTypeObject` no forma parte de *stable ABI*, cualquier módulo de extensión que use tipos estáticos debe compilarse para una versión menor específica de Python.

Una alternativa a los tipos estáticos es *tipos asignados al montículo* (*heap-allocated types*), o *tipos montículo* (*heap types*) para abreviar, que corresponden estrechamente a las clases creadas por la declaración `class` de Python.

Esto se hace completando una estructura `PyType_Spec` y llamando a `PyType_FromSpecWithBases()`.

12.4 Estructuras de Objetos de Números

`PyNumberMethods`

Esta estructura contiene punteros a las funciones que utiliza un objeto para implementar el protocolo numérico. Cada función es utilizada por la función de un nombre similar documentado en la sección *Protocolo de números*.

Aquí está la definición de la estructura:

```
typedef struct {
    binaryfunc nb_add;
    binaryfunc nb_subtract;
    binaryfunc nb_multiply;
    binaryfunc nb_remainder;
    binaryfunc nb_divmod;
    ternaryfunc nb_power;
    unaryfunc nb_negative;
    unaryfunc nb_positive;
    unaryfunc nb_absolute;
    inquiry nb_bool;
    unaryfunc nb_invert;
    binaryfunc nb_lshift;
    binaryfunc nb_rshift;
    binaryfunc nb_and;
    binaryfunc nb_xor;
    binaryfunc nb_or;
    unaryfunc nb_int;
    void *nb_reserved;
    unaryfunc nb_float;

    binaryfunc nb_inplace_add;
    binaryfunc nb_inplace_subtract;
    binaryfunc nb_inplace_multiply;
    binaryfunc nb_inplace_remainder;
    ternaryfunc nb_inplace_power;
    binaryfunc nb_inplace_lshift;
```

(continué en la próxima página)

(proviene de la página anterior)

```
binaryfunc nb_inplace_rshift;
binaryfunc nb_inplace_and;
binaryfunc nb_inplace_xor;
binaryfunc nb_inplace_or;

binaryfunc nb_floor_divide;
binaryfunc nb_true_divide;
binaryfunc nb_inplace_floor_divide;
binaryfunc nb_inplace_true_divide;

unaryfunc nb_index;

binaryfunc nb_matrix_multiply;
binaryfunc nb_inplace_matrix_multiply;
} PyNumberMethods;
```

Nota: Las funciones binarias y ternarias deben verificar el tipo de todos sus operandos e implementar las conversiones necesarias (al menos uno de los operandos es una instancia del tipo definido). Si la operación no está definida para los operandos dados, las funciones binarias y ternarias deben retornar `Py_NotImplemented`, si se produce otro error, deben retornar `NULL` y establecer una excepción.

Nota: El campo `nb_reserved` siempre debe ser `NULL`. Anteriormente se llamaba `nb_long`, y se renombró en Python 3.0.1.

binaryfunc `PyNumberMethods.nb_add`
binaryfunc `PyNumberMethods.nb_subtract`
binaryfunc `PyNumberMethods.nb_multiply`
binaryfunc `PyNumberMethods.nb_remainder`
binaryfunc `PyNumberMethods.nb_divmod`
ternaryfunc `PyNumberMethods.nb_power`
unaryfunc `PyNumberMethods.nb_negative`
unaryfunc `PyNumberMethods.nb_positive`
unaryfunc `PyNumberMethods.nb_absolute`
inquiry `PyNumberMethods.nb_bool`
unaryfunc `PyNumberMethods.nb_invert`
binaryfunc `PyNumberMethods.nb_lshift`
binaryfunc `PyNumberMethods.nb_rshift`
binaryfunc `PyNumberMethods.nb_and`
binaryfunc `PyNumberMethods.nb_xor`
binaryfunc `PyNumberMethods.nb_or`
unaryfunc `PyNumberMethods.nb_int`
`void *``PyNumberMethods.nb_reserved`

unaryfunc **PyNumberMethods.nb_float**

binaryfunc **PyNumberMethods.nb_inplace_add**

binaryfunc **PyNumberMethods.nb_inplace_subtract**

binaryfunc **PyNumberMethods.nb_inplace_multiply**

binaryfunc **PyNumberMethods.nb_inplace_remainder**

ternaryfunc **PyNumberMethods.nb_inplace_power**

binaryfunc **PyNumberMethods.nb_inplace_lshift**

binaryfunc **PyNumberMethods.nb_inplace_rshift**

binaryfunc **PyNumberMethods.nb_inplace_and**

binaryfunc **PyNumberMethods.nb_inplace_xor**

binaryfunc **PyNumberMethods.nb_inplace_or**

binaryfunc **PyNumberMethods.nb_floor_divide**

binaryfunc **PyNumberMethods.nb_true_divide**

binaryfunc **PyNumberMethods.nb_inplace_floor_divide**

binaryfunc **PyNumberMethods.nb_inplace_true_divide**

unaryfunc **PyNumberMethods.nb_index**

binaryfunc **PyNumberMethods.nb_matrix_multiply**

binaryfunc **PyNumberMethods.nb_inplace_matrix_multiply**

12.5 Estructuras de Objetos Mapeo

PyMappingMethods

Esta estructura contiene punteros a las funciones que utiliza un objeto para implementar el protocolo de mapeo. Tiene tres miembros:

lenfunc **PyMappingMethods.mp_length**

Esta función es utilizada por *PyMapping_Size()* y *PyObject_Size()*, y tiene la misma firma. Esta ranura puede establecerse en NULL si el objeto no tiene una longitud definida.

binaryfunc **PyMappingMethods.mp_subscript**

Esta función es utilizada por *PyObject_GetItem()* y *PySequence_GetSlice()*, y tiene la misma firma que *PyObject_GetItem()*. Este espacio debe llenarse para que la función *PyMapping_Check()* retorne 1, de lo contrario puede ser NULL.

objobjargproc **PyMappingMethods.mp_ass_subscript**

Esta función es utilizada por *PyObject_SetItem()*, *PyObject_DelItem()*, *PyObject_SetSlice()* y *PyObject_DelSlice()*. Tiene la misma firma que *PyObject_SetItem()*, pero *v* también se puede establecer en NULL para eliminar un elemento. Si este espacio es NULL, el objeto no admite la asignación y eliminación de elementos.

12.6 Estructuras de objetos secuencia

PySequenceMethods

Esta estructura contiene punteros a las funciones que utiliza un objeto para implementar el protocolo de secuencia.

lenfunc **PySequenceMethods.sq_length**

Esta función es utilizada por *PySequence_Size()* y *PyObject_Size()*, y tiene la misma firma. También se usa para manejar índices negativos a través de los espacios *sq_item* y *sq_ass_item*.

binaryfunc **PySequenceMethods.sq_concat**

Esta función es utilizada por *PySequence_Concat()* y tiene la misma firma. También es utilizado por el operador +, después de intentar la suma numérica a través de la ranura *nb_add*.

ssizeargfunc **PySequenceMethods.sq_repeat**

Esta función es utilizada por *PySequence_Repeat()* y tiene la misma firma. También es utilizado por el operador *, después de intentar la multiplicación numérica a través de la ranura *nb_multiply*.

ssizeargfunc **PySequenceMethods.sq_item**

Esta función es utilizada por *PySequence_GetItem()* y tiene la misma firma. También es utilizado por *PyObject_GetItem()*, después de intentar la suscripción a través de la ranura *mp_subscript*. Este espacio debe llenarse para que la función *PySequence_Check()* retorne 1, de lo contrario puede ser NULL.

Los índices negativos se manejan de la siguiente manera: si se llena el espacio *sq_length*, se llama y la longitud de la secuencia se usa para calcular un índice positivo que se pasa a *sq_item*. Si *sq_length* es NULL, el índice se pasa como es a la función.

ssizeobjargproc **PySequenceMethods.sq_ass_item**

Esta función es utilizada por *PySequence_SetItem()* y tiene la misma firma. También lo usan *PyObject_SetItem()* y *PyObject_DelItem()*, después de intentar la asignación y eliminación del elemento a través de la ranura *mp_ass_subscript*. Este espacio puede dejarse en NULL si el objeto no admite la asignación y eliminación de elementos.

objobjproc **PySequenceMethods.sq_contains**

Esta función puede ser utilizada por *PySequence_Contains()* y tiene la misma firma. Este espacio puede dejarse en NULL, en este caso *PySequence_Contains()* simplemente atraviesa la secuencia hasta que encuentra una coincidencia.

binaryfunc **PySequenceMethods.sq_inplace_concat**

Esta función es utilizada por *PySequence_InPlaceConcat()* y tiene la misma firma. Debería modificar su primer operando y retornarlo. Este espacio puede dejarse en NULL, en este caso *PySequence_InPlaceConcat()* volverá a *PySequence_Concat()*. También es utilizado por la asignación aumentada +=, después de intentar la suma numérica en el lugar a través de la ranura *nb_inplace_add*.

ssizeargfunc **PySequenceMethods.sq_inplace_repeat**

Esta función es utilizada por *PySequence_InPlaceRepeat()* y tiene la misma firma. Debería modificar su primer operando y retornarlo. Este espacio puede dejarse en NULL, en este caso *PySequence_InPlaceRepeat()* volverá a *PySequence_Repeat()*. También es utilizado por la asignación aumentada *=, después de intentar la multiplicación numérica en el lugar a través de la ranura *nb_inplace_multiply*.

12.7 Estructuras de Objetos Búfer

PyBufferProcs

Esta estructura contiene punteros a las funciones requeridas por *Buffer protocol*. El protocolo define cómo un objeto exportador puede exponer sus datos internos a objetos de consumo.

getbufferproc **PyBufferProcs.bf_getbuffer**

La firma de esta función es:

```
int (PyObject *exporter, Py_buffer *view, int flags);
```

Maneja una solicitud a *exporter* para completar *view* según lo especificado por *flags*. Excepto por el punto (3), una implementación de esta función DEBE seguir estos pasos:

- (1) Comprueba si se puede cumplir con la solicitud. Si no, lanza `PyExc_BufferError`, establece `view->obj` en `NULL` y retorna `-1`.
- (2) Rellene los campos solicitados.
- (3) Incrementa un contador interno para el número de exportaciones (*exports*).
- (4) Establece `view->obj` en *exporter* e incremente `view->obj`.
- (5) Retorna `0`.

Si *exporter* es parte de una cadena o árbol de proveedores de búfer, se pueden usar dos esquemas principales:

- Re-exportación: cada miembro del árbol actúa como el objeto exportador y establece `view->obj` en una nueva referencia a sí mismo.
- Redirigir: la solicitud de búfer se redirige al objeto raíz del árbol. Aquí `view->obj` será una nueva referencia al objeto raíz.

Los campos individuales de *view* se describen en la sección *Estructura de búfer*, las reglas sobre cómo debe reaccionar un exportador a solicitudes específicas se encuentran en la sección *Tipos de solicitud de búfer*.

Toda la memoria señalada en la estructura `Py_buffer` pertenece al exportador y debe permanecer válida hasta que no queden consumidores. *format*, *shape*, *strides*, *suboffsets* y *internal* son de solo lectura para el consumidor.

`PyBuffer_FillInfo()` proporciona una manera fácil de exponer un búfer de bytes simple mientras se trata correctamente con todos los tipos de solicitud.

`PyObject_GetBuffer()` es la interfaz para el consumidor que envuelve esta función.

releasebufferproc **PyBufferProcs.bf_releasebuffer**

La firma de esta función es:

```
void (PyObject *exporter, Py_buffer *view);
```

Maneja una solicitud para liberar los recursos del búfer. Si no es necesario liberar recursos, `PyBufferProcs.bf_releasebuffer` puede ser `NULL`. De lo contrario, una implementación estándar de esta función tomará estos pasos opcionales:

- (1) Disminuir un contador interno para el número de exportaciones.
- (2) Si el contador es `0`, libera toda la memoria asociada con *view*.

El exportador DEBE utilizar el campo *internal* para realizar un seguimiento de los recursos específicos del búfer. Se garantiza que este campo permanecerá constante, mientras que un consumidor PUEDE pasar una copia del búfer original como argumento *view*.

Esta función NO DEBE disminuir `view->obj`, ya que esto se hace automáticamente en `PyBuffer_Release()` (este esquema es útil para romper los ciclos de referencia).

`PyBuffer_Release()` es la interfaz para el consumidor que envuelve esta función.

12.8 Estructuras de objetos asíncronos

Nuevo en la versión 3.5.

PyAsyncMethods

Esta estructura contiene punteros a las funciones requeridas para implementar objetos «esperable» (*awaitable*) y «iterador asíncrono» (*asynchronous iterator*).

Aquí está la definición de la estructura:

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
} PyAsyncMethods;
```

unaryfunc **PyAsyncMethods.am_await**

La firma de esta función es:

```
PyObject *am_await(PyObject *self);
```

El objeto retornado debe ser un iterador, es decir `PyIter_Check()` debe retornar 1 para ello.

Este espacio puede establecerse en NULL si un objeto no es *awaitable*.

unaryfunc **PyAsyncMethods.am_aiter**

La firma de esta función es:

```
PyObject *am_aiter(PyObject *self);
```

Must return an *asynchronous iterator* object. See `__anext__()` for details.

Este espacio puede establecerse en NULL si un objeto no implementa el protocolo de iteración asíncrona.

unaryfunc **PyAsyncMethods.am_anext**

La firma de esta función es:

```
PyObject *am_anext(PyObject *self);
```

Debe retornar un objeto «esperable» (*awaitable*). Ver `__anext__()` para más detalles. Esta ranura puede establecerse en NULL.

12.9 Tipo Ranura *typedefs*

PyObject * (***allocfunc**) (*PyTypeObject* *cls, *Py_ssize_t* nitems)

El propósito de esta función es separar la asignación de memoria de la inicialización de memoria. Debería retornar un puntero a un bloque de memoria de longitud adecuada para la instancia, adecuadamente alineado e inicializado a ceros, pero con `ob_refcnt` establecido en 1 y `ob_type` establecido en argumento de tipo. Si el tipo `tp_itemsize` no es cero, el campo del objeto `ob_size` debe inicializarse en `nitems` y la longitud del bloque de memoria asignado debe ser `tp_basicsize + nitems*tp_itemsize`, redondeado a un múltiplo de `sizeof(void*)`; de lo contrario, `nitems` no se usa y la longitud del bloque debe ser `tp_basicsize`.

Esta función no debe hacer ninguna otra instancia de inicialización, ni siquiera para asignar memoria adicional; eso debe ser realizado por `tp_new`.

void (***destructor**) (*PyObject* *)

void (***freefunc**) (void *)

Consulte `tp_free`.

PyObject * (***newfunc**) (*PyObject* *, *PyObject* *, *PyObject* *)

Consulte `tp_new`.

int (***initproc**) (*PyObject* *, *PyObject* *, *PyObject* *)

Consulte `tp_init`.

PyObject * (***reprfunc**) (*PyObject* *)

Consulte `tp_repr`.

PyObject * (***getattrfunc**) (*PyObject* *self, char *attr)

Retorna el valor del atributo nombrado para el objeto.

int (***setattrfunc**) (*PyObject* *self, char *attr, *PyObject* *value)

Establece el valor del atributo nombrado para el objeto. El argumento del valor se establece en `NULL` para eliminar el atributo.

PyObject * (***getattrofunc**) (*PyObject* *self, *PyObject* *attr)

Retorna el valor del atributo nombrado para el objeto.

Consulte `tp_getattro`.

int (***setattrofunc**) (*PyObject* *self, *PyObject* *attr, *PyObject* *value)

Establece el valor del atributo nombrado para el objeto. El argumento del valor se establece en `NULL` para eliminar el atributo.

Consulte `tp_setattro`.

PyObject * (***descrgetfunc**) (*PyObject* *, *PyObject* *, *PyObject* *)

Consulte `tp_descrget`.

int (***descrsetfunc**) (*PyObject* *, *PyObject* *, *PyObject* *)

Consulte `tp_descrset`.

Py_hash_t (***hashfunc**) (*PyObject* *)

Consulte `tp_hash`.

PyObject * (***richcmpfunc**) (*PyObject* *, *PyObject* *, int)

Consulte `tp_richcompare`.

PyObject * (***getiterfunc**) (*PyObject* *)

Consulte `tp_iter`.

PyObject * (***iternextfunc**) (*PyObject* *)

Consulte `tp_iternext`.

```
Py_ssize_t (*lenfunc) (PyObject *)
int (*getbufferproc) (PyObject *, Py_buffer *, int)
void (*releasebufferproc) (PyObject *, Py_buffer *)
PyObject* (*unaryfunc) (PyObject *)
PyObject* (*binaryfunc) (PyObject *, PyObject *)
PyObject* (*ternaryfunc) (PyObject *, PyObject *, PyObject *)
PyObject* (*ssizeargfunc) (PyObject *, Py_ssize_t)
int (*ssizeobjargproc) (PyObject *, Py_ssize_t)
int (*objobjproc) (PyObject *, PyObject *)
int (*objobjargproc) (PyObject *, PyObject *, PyObject *)
```

12.10 Ejemplos

Los siguientes son ejemplos simples de definiciones de tipo Python. Incluyen el uso común que puede encontrar. Algunos demuestran casos difíciles de esquina (*corner cases*). Para obtener más ejemplos, información práctica y un tutorial, consulte «definiendo nuevos tipos» ([defining-new-types](#)) y «tópicos de nuevos tipos ([new-types-topics](#))».

Un tipo estático básico:

```
typedef struct {
    PyObject_HEAD
    const char *data;
} PyObject;

static PyTypeObject PyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_new = myobj_new,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
};
```

También puede encontrar código más antiguo (especialmente en la base de código CPython) con un inicializador más detallado:

```
static PyTypeObject PyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "mymod.MyObject",           /* tp_name */
    sizeof(MyObject),           /* tp_basicsize */
    0,                           /* tp_itemsize */
    (destructor)myobj_dealloc,   /* tp_dealloc */
    0,                           /* tp_vectorcall_offset */
    0,                           /* tp_getattr */
    0,                           /* tp_setattr */
    0,                           /* tp_as_async */
    (reprfunc)myobj_repr,        /* tp_repr */
    0,                           /* tp_as_number */
    0,                           /* tp_as_sequence */
```

(continué en la próxima página)

(proviene de la página anterior)

```

0,          /* tp_as_mapping */
0,          /* tp_hash */
0,          /* tp_call */
0,          /* tp_str */
0,          /* tp_getattro */
0,          /* tp_setattro */
0,          /* tp_as_buffer */
0,          /* tp_flags */
PyDoc_STR("My objects"), /* tp_doc */
0,          /* tp_traverse */
0,          /* tp_clear */
0,          /* tp_richcompare */
0,          /* tp_weaklistoffset */
0,          /* tp_iter */
0,          /* tp_iternext */
0,          /* tp_methods */
0,          /* tp_members */
0,          /* tp_getset */
0,          /* tp_base */
0,          /* tp_dict */
0,          /* tp_descr_get */
0,          /* tp_descr_set */
0,          /* tp_dictoffset */
0,          /* tp_init */
0,          /* tp_alloc */
myobj_new, /* tp_new */
};

```

Un tipo que admite referencias débiles, instancias de diccionarios (*dicts*) y *hashing*:

```

typedef struct {
    PyObject_HEAD
    const char *data;
    PyObject *inst_dict;
    PyObject *weakreflist;
} PyObject;

static PyTypeObject PyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(PyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_weaklistoffset = offsetof(PyObject, weakreflist),
    .tp_dictoffset = offsetof(PyObject, inst_dict),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
    .tp_new = myobj_new,
    .tp_traverse = (traverseproc)myobj_traverse,
    .tp_clear = (inquiry)myobj_clear,
    .tp_alloc = PyType_GenericNew,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
    .tp_hash = (hashfunc)myobj_hash,
    .tp_richcompare = PyBaseObject_Type.tp_richcompare,
};

```

Una subclase de *str* que no se puede subclassificar (*subclassed*) y no se puede llamar para crear instancias (por ejemplo, utiliza una función de fábrica separada):


```
typedef struct {
    PyUnicodeObject raw;
    char *extra;
} MyStr;

static PyObject MyStr_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyStr",
    .tp_basicsize = sizeof(MyStr),
    .tp_base = NULL, // set to &PyUnicode_Type in module init
    .tp_doc = PyDoc_STR("my custom str"),
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = NULL,
    .tp_repr = (reprfunc)myobj_repr,
};
```

El tipo estático más simple (con instancias de longitud fija):

```
typedef struct {
    PyObject_HEAD
} MyObject;

static PyObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
};
```

El tipo estático más simple (con instancias de longitud variable):

```
typedef struct {
    PyObject_VAR_HEAD
    const char *data[1];
} MyObject;

static PyObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject) - sizeof(char *),
    .tp_itemsize = sizeof(char *),
};
```

12.11 Apoyo a la recolección de basura cíclica

El soporte de Python para detectar y recolectar basura que involucra referencias circulares requiere el soporte de tipos de objetos que son «contenedores» para otros objetos que también pueden ser contenedores. Los tipos que no almacenan referencias a otros objetos, o que solo almacenan referencias a tipos atómicos (como números o cadenas), no necesitan proporcionar ningún soporte explícito para la recolección de basura.

Para crear un tipo de contenedor, el campo `tp_flags` del objeto tipo debe incluir `Py_TPFLAGS_HAVE_GC` y proporcionar una implementación del manejador `tp_traverse`. Si las instancias del tipo son mutables, también se debe proporcionar una implementación a `tp_clear`.

Py_TPFLAGS_HAVE_GC

Los objetos con un tipo con este indicador establecido deben cumplir con las reglas documentadas aquí. Por conveniencia, estos objetos se denominarán objetos contenedor.

Los constructores para tipos de contenedores deben cumplir con dos reglas:

1. La memoria para el objeto debe asignarse usando `PyObject_GC_New()` o `PyObject_GC_NewVar()`.
2. Una vez que se inicializan todos los campos que pueden contener referencias a otros contenedores, debe llamar a `PyObject_GC_Track()`.

Del mismo modo, el desasignador (*dealloc*) para el objeto debe cumplir con un par similar de reglas:

1. Antes de invalidar los campos que se refieren a otros contenedores, debe llamarse `PyObject_GC_UnTrack()`.
2. La memoria del objeto debe ser desasignada (*deallocated*) usando `PyObject_GC_Del()`.

Advertencia: If a type adds the `Py_TPFLAGS_HAVE_GC`, then it *must* implement at least a `tp_traverse` handler or explicitly use one from its subclass or subclasses.

When calling `PyType_Ready()` or some of the APIs that indirectly call it like `PyType_FromSpecWithBases()` or `PyType_FromSpec()` the interpreter will automatically populate the `tp_flags`, `tp_traverse` and `tp_clear` fields if the type inherits from a class that implements the garbage collector protocol and the child class does *not* include the `Py_TPFLAGS_HAVE_GC` flag.

TYPE* **PyObject_GC_New**(TYPE, *PyTypeObject* *type)

Análogo a `PyObject_New()` pero para objetos de contenedor con el flag `Py_TPFLAGS_HAVE_GC` establecido.

TYPE* **PyObject_GC_NewVar**(TYPE, *PyTypeObject* *type, *Py_ssize_t* size)

Análogo a `PyObject_NewVar()` pero para objetos de contenedor con el flag `Py_TPFLAGS_HAVE_GC` establecido.

TYPE* **PyObject_GC_Resize**(TYPE, *PyVarObject* *op, *Py_ssize_t* newsize)

Cambia el tamaño de un objeto asignado por `PyObject_NewVar()`. Retorna el objeto redimensionado o NULL en caso de falla. *op* aún no debe ser rastreado por el recolector de basura.

void **PyObject_GC_Track**(*PyObject* *op)

Agrega el objeto *op* al conjunto de objetos contenedor seguidos por el recolector de basura. El recolector puede ejecutarse en momentos inesperados, por lo que los objetos deben ser válidos durante el seguimiento. Esto debería llamarse una vez que todos los campos seguidos por `tp_traverse` se vuelven válidos, generalmente cerca del final del constructor.

int **PyObject_IS_GC**(*PyObject* *obj)

Retorna un valor distinto de cero si el objeto implementa el protocolo del recolector de basura; de lo contrario, retorna 0.

El recolector de basura no puede rastrear el objeto si esta función retorna 0.

int **PyObject_GC_IsTracked**(*PyObject* *op)

Retorna 1 si el tipo de objeto de *op* implementa el protocolo GC y el recolector de basura está rastreando *op* y 0 en caso contrario.

Esto es análogo a la función de Python `gc.is_tracked()`.

Nuevo en la versión 3.9.

int **PyObject_GC_IsFinalized**(*PyObject* *op)

Retorna 1 si el tipo de objeto de *op* implementa el protocolo GC y *op* ya ha sido finalizado por el recolector de basura y 0 en caso contrario.

Esto es análogo a la función de Python `gc.is_finalized()`.

Nuevo en la versión 3.9.

void **PyObject_GC_Del** (void **op*)

Libera memoria asignada a un objeto usando `PyObject_GC_New()` o `PyObject_GC_NewVar()`.

void **PyObject_GC_UnTrack** (void **op*)

Elimina el objeto *op* del conjunto de objetos contenedor rastreados por el recolector de basura. Tenga en cuenta que `PyObject_GC_Track()` puede ser llamado nuevamente en este objeto para agregarlo nuevamente al conjunto de objetos rastreados. El desasignador (el manejador `tp_dealloc`) debería llamarlo para el objeto antes de que cualquiera de los campos utilizados por el manejador `tp_traverse` no sea válido.

Distinto en la versión 3.8: Los macros `_PyObject_GC_TRACK()` y `_PyObject_GC_UNTRACK()` se han eliminado de la API pública de C.

El manejador `tp_traverse` acepta un parámetro de función de este tipo:

int (***visitproc**) (*PyObject* **object*, void **arg*)

Tipo de la función visitante que se pasa al manejador `tp_traverse`. La función debe llamarse con un objeto para atravesar como *object* y el tercer parámetro para el manejador `tp_traverse` como *arg*. El núcleo de Python utiliza varias funciones visitantes para implementar la detección de basura cíclica; No se espera que los usuarios necesiten escribir sus propias funciones visitante.

El manejador `tp_traverse` debe tener el siguiente tipo:

int (***traverseproc**) (*PyObject* **self*, *visitproc* *visit*, void **arg*)

Función transversal para un objeto contenedor. Las implementaciones deben llamar a la función *visit* para cada objeto directamente contenido por *self*, siendo los parámetros a *visit* el objeto contenido y el valor *arg* pasado al controlador. La función *visit* no debe llamarse con un argumento de objeto NULL. Si *visit* retorna un valor distinto de cero, ese valor debe retornarse inmediatamente.

Para simplificar la escritura de los manejadores `tp_traverse`, se proporciona un macro a `Py_VISIT()`. Para usar este macro, la implementación `tp_traverse` debe nombrar sus argumentos exactamente *visit* y *arg*:

void **Py_VISIT** (*PyObject* **o*)

Si *o* no es NULL, llama a la devolución de llamada (*callback*) *visit*, con argumentos *o* y *arg*. Si *visit* retorna un valor distinto de cero, lo retorna. Usando este macro, los manejadores `tp_traverse` tienen el siguiente aspecto:

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

El manejador `tp_clear` debe ser del tipo `query`, o NULL si el objeto es inmutable.

int (***inquiry**) (*PyObject* **self*)

Descarta referencias que pueden haber creado ciclos de referencia. Los objetos inmutables no tienen que definir este método ya que nunca pueden crear directamente ciclos de referencia. Tenga en cuenta que el objeto aún debe ser válido después de llamar a este método (no solo llame a `Py_DECREF()` en una referencia). El recolector de basura llamará a este método si detecta que este objeto está involucrado en un ciclo de referencia.

CAPÍTULO 13

Versiones de API y ABI

PY_VERSION_HEX es el número de versión de Python codificado en un solo entero.

Por ejemplo, si PY_VERSION_HEX se establece en 0x030401a2, la información de la versión subyacente se puede encontrar tratándola como un número de 32 bits de la siguiente manera:

By-tes	Bits (orden <i>big-endian</i>)	Significado
1	1-8	PY_MAJOR_VERSION (el 3 en 3.4.1a2)
2	9-16	PY_MINOR_VERSION (el 4 en 3.4.1a2)
3	17-24	PY_MICRO_VERSION (el 1 en 3.4.1a2)
4	25-28	PY_RELEASE_LEVEL (0xA para alfa, 0xB para beta, 0xC para el candidato de lanzamiento y 0xF para final), en este caso es alfa.
	29-32	PY_RELEASE_SERIAL (el 2 en 3.4.1a2, cero para lanzamientos finales)

Así 3.4.1a2 es la hexadecimal 0x030401a2.

Todas las macros dadas se definen en [Include/patchlevel.h](#).

>>> 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 una 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 una sentencia `async with` al definir los métodos `__aenter__()` y `__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 referenciado por el nombre usado en 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 *Protocolo Búfer* 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 correr 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`.

retrollamada Una función de subrutina que se pasa como un argumento para ejecutarse en algún momento en el futuro.

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

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

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 descriptors 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 obtener más información sobre los métodos de los descriptors, consulte `descriptors` o Guía práctica de uso de los 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.

comprensión de diccionarios Una forma compacta de procesar todos o parte de los elementos en un iterable y retornar un diccionario con los resultados. `results = {n: n ** 2 for n in range(10)}` genera un diccionario que contiene la clave `n` asignada al valor `n ** 2`. Ver `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 *LYL* 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 *statements* 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__ A future statement, from `__future__ import <feature>`, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The `__future__` module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:

```
>>> 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**.

tipos genéricos A *type* that can be parameterized; typically a container class such as `list` or `dict`. Used for *type hints* and *annotations*.

For more details, see generic alias types, **PEP 483**, **PEP 484**, **PEP 585**, and the `typing` module.

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.

immutable 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 busca 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 intérprete es que el módulo `__main__` o el script que estaba corriendo termine su ejecución.

iterable Un objeto capaz de retornar sus miembros uno por vez. Ejemplos de iterables son todos los tipos de secuencias (como `list`, `str`, y `tuple`) y algunos de tipos no secuenciales, como `dict`, *objeto archivo*, y objetos de cualquier clase que defina con los métodos `__iter__()` o con un método `__getitem__()` que implementen la semántica de *Sequence*.

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 *tyeiter*.

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.

metaclass La clase de una clase. Las definiciones de clases crean nombres de clase, un diccionario de clase, y una lista de clases base. Las metaclasses 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 metaclasses a medida. La mayoría de los usuario nunca necesitarán esta herramienta, pero cuando la necesidad surge, las metaclasses 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 hecha personalmente 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, `__getattr__()`, 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 provisional 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 provisionales, 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.

La clase abstracta base `collections.abc.Sequence` define una interfaz mucho más rica que va más allá de sólo `__getitem__()` y `__len__()`, agregando `count()`, `index()`, `__contains__()`, y `__reversed__()`. Los tipos que implementan esta interfaz expandida pueden ser registrados explícitamente usando `register()`.

comprensión de conjuntos Una forma compacta de procesar todos o parte de los elementos en un iterable y retornar un conjunto con los resultados. `results = {c for c in 'abracadabra' if c not in 'abc'}` genera el conjunto de cadenas `{'r', 'd'}`. Ver *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 suscriptor, `[]` con dos puntos entre los números cuando se ponen varios, como en `nombre_variable[1:3:5]`. La notación con corchete (suscriptor) 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 A string in Python is a sequence of Unicode code points (in range U+0000–U+10FFFF). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as «encoding», and recreating the string from the sequence of bytes is known as «decoding».

There are a variety of different text serialization codecs, which are collectively referred to as «text encodings».

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:

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

podría ser más legible así:

```
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.

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 for his Alternative Python Reference project from which Sphinx got many good 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!

Historia y Licencia

C.1 Historia del software

Python fue creado a principios de la década de 1990 por Guido van Rossum en Stichting Mathematisch Centrum (CWI, ver <https://www.cwi.nl/>) en los Países Bajos como sucesor de un idioma llamado ABC. Guido sigue siendo el autor principal de Python, aunque incluye muchas contribuciones de otros.

En 1995, Guido continuó su trabajo en Python en la Corporation for National Research Initiatives (CNRI, consulte <https://www.cnri.reston.va.us/>) en Reston, Virginia, donde lanzó varias versiones del software.

En mayo de 2000, Guido y el equipo de desarrollo central de Python se trasladaron a BeOpen.com para formar el equipo de BeOpen PythonLabs. En octubre del mismo año, el equipo de PythonLabs se trasladó a Digital Creations (ahora Zope Corporation; consulte <https://www.zope.org/>). En 2001, se formó la Python Software Foundation (PSF, consulte <https://www.python.org/psf/>), una organización sin fines de lucro creada específicamente para poseer la propiedad intelectual relacionada con Python. Zope Corporation es miembro patrocinador del PSF.

Todas las versiones de Python son de código abierto (consulte <https://opensource.org/> para conocer la definición de código abierto). Históricamente, la mayoría de las versiones de Python, pero no todas, también han sido compatibles con GPL; la siguiente tabla resume las distintas versiones.

Lanzamiento	Derivado de	Año	Dueño/a	¿compatible con GPL?
0.9.0 hasta 1.2	n/a	1991-1995	CWI	sí
1.3 hasta 1.5.2	1.2	1995-1999	CNRI	sí
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	sí
2.1.1	2.1+2.0.1	2001	PSF	sí
2.1.2	2.1.1	2002	PSF	sí
2.1.3	2.1.2	2002	PSF	sí
2.2 y superior	2.1.1	2001-ahora	PSF	sí

Nota: Compatible con GPL no significa que estemos distribuyendo Python bajo la GPL. Todas las licencias de Python, a diferencia de la GPL, le permiten distribuir una versión modificada sin que los cambios sean de código abierto. Las licencias compatibles con GPL permiten combinar Python con otro software que se publica bajo la GPL; los otros no lo hacen.

Gracias a los muchos voluntarios externos que han trabajado bajo la dirección de Guido para hacer posibles estos lanzamientos.

C.2 Términos y condiciones para acceder o usar Python

El software y la documentación de Python están sujetos a *Acuerdo de licencia de PSF*.

A partir de Python 3.8.6, los ejemplos, recetas y otros códigos de la documentación tienen licencia doble según el Acuerdo de licencia de PSF y la *Licencia BSD de cláusula cero*.

Parte del software incorporado en Python está bajo diferentes licencias. Las licencias se enumeran con el código correspondiente a esa licencia. Consulte *Licencias y reconocimientos para software incorporado* para obtener una lista incompleta de estas licencias.

C.2.1 ACUERDO DE LICENCIA DE PSF PARA PYTHON | lanzamiento |

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),
→and
the Individual or Organization ("Licensee") accessing and otherwise using.
→Python
3.9.18 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 3.9.18 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 3.9.18 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 3.9.18 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
3.9.18.
4. PSF is making Python 3.9.18 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 3.9.18 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.9.18
 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT
 →OF
 MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.9.18, 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 3.9.18, Licensee agrees
 to be bound by the terms and conditions of this License Agreement.

C.2.2 ACUERDO DE LICENCIA DE BEOPEN.COM PARA PYTHON 2.0

ACUERDO DE LICENCIA DE CÓDIGO ABIERTO DE BEOPEN PYTHON VERSIÓN 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.

(continué en la próxima página)

(proviene de la página anterior)

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.3 ACUERDO DE LICENCIA CNRI PARA 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

(continué en la próxima página)

(proviene de la página anterior)

THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

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

C.2.4 ACUERDO DE LICENCIA CWI PARA PYTHON 0.9.0 HASTA 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 LICENCIA BSD DE CLÁUSULA CERO PARA CÓDIGO EN EL PYTHON | lanzamiento | DOCUMENTACIÓN

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

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3 Licencias y reconocimientos para software incorporado

Esta sección es una lista incompleta, pero creciente, de licencias y reconocimientos para software de terceros incorporado en la distribución de Python.

C.3.1 Mersenne Twister

El módulo `_random` incluye código basado en una descarga de <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html>. Los siguientes son los comentarios textuales del código original:

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,

(continué en la próxima página)

(proviene de la página anterior)

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

El módulo `socket` usa las funciones, `getaddrinfo()`, y `getnameinfo()`, que están codificadas en archivos fuente separados del Proyecto WIDE, <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 Servicios de socket asincrónicos

Los módulos `asyncchat` y `asyncore` contienen el siguiente aviso:

```
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 Gestión de cookies

El módulo `http.cookies` contiene el siguiente aviso:

```
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 Seguimiento de ejecución

El módulo `trace` contiene el siguiente aviso:

```
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 funciones UUencode y UUdecode

El módulo `uu` contiene el siguiente aviso:

```
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
```

(continué en la próxima página)

(proviene de la página anterior)

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

C.3.7 Llamadas a procedimientos remotos XML

El módulo `xmlrpc.client` contiene el siguiente aviso:

```
The XML-RPC client interface is  
  
Copyright (c) 1999-2002 by Secret Labs AB  
Copyright (c) 1999-2002 by Fredrik Lundh  
  
By obtaining, using, and/or copying this software and/or its  
associated documentation, you agree that you have read, understood,  
and will comply with the following terms and conditions:  
  
Permission to use, copy, modify, and distribute this software and  
its associated documentation for any purpose and without fee is  
hereby granted, provided that the above copyright notice appears in  
all copies, and that both that copyright notice and this permission  
notice appear in supporting documentation, and that the name of  
Secret Labs AB or the author not be used in advertising or publicity  
pertaining to distribution of the software without specific, written  
prior permission.  
  
SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD  
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-  
ABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR  
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY  
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,  
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS  
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE  
OF THIS SOFTWARE.
```

C.3.8 test_epoll

El módulo `test_epoll` contiene el siguiente aviso:

```
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
```

(continué en la próxima página)

(proviene de la página anterior)

```
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 Seleccionar kqueue

El módulo `select` contiene el siguiente aviso para la interfaz `kqueue`:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.
```

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

```
THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

El archivo `Python/pyhash.c` contiene la implementación de Marek Majkowski del algoritmo SipHash24 de Dan Bernstein. Contiene la siguiente nota:

```
<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>
```

(continué en la próxima página)

(proviene de la página anterior)

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

Solution inspired by code from:
    Samuel Neves (supercop/crypto_auth/siphhash24/little)
    djb (supercop/crypto_auth/siphhash24/little2)
    Jean-Philippe Aumasson (https://131002.net/siphhash/siphhash24.c)
```

C.3.11 strtod y dtoa

El archivo `Python/dtoa.c`, que proporciona las funciones de C `dtoa` y `strtod` para la conversión de C dobles hacia y desde cadenas, se deriva del archivo del mismo nombre de David M. Gay, actualmente disponible en <http://www.netlib.org/fp/>. El archivo original, recuperado el 16 de marzo de 2009, contiene el siguiente aviso de licencia y derechos de autor:

```
/* *****
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 * *****/
```

C.3.12 OpenSSL

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS 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
-----

/* =====
```

(continué en la próxima página)

(proviene de la página anterior)

```

* 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/)"
*
* 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)

(continué en la próxima página)

(proviene de la página anterior)

```

* 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
*    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

La extensión `pyexpat` se construye usando una copia incluida de las fuentes de `expat` a menos que la construcción esté configurada `--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

La extensión `_ctypes` se construye usando una copia incluida de las fuentes de `libffi` a menos que la construcción esté configurada `--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

La extensión `zlib` se crea utilizando una copia incluida de las fuentes de `zlib` si la versión de `zlib` encontrada en el sistema es demasiado antigua para ser utilizada para la compilación:

```
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

La implementación de la tabla hash utilizada por `tracemalloc` se basa en el proyecto `cfuhash`:

```
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
```

(continué en la próxima página)

(proviene de la página anterior)

```
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

El módulo `_decimal` se construye usando una copia incluida de la biblioteca `libmpdec` a menos que la construcción esté configurada `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

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

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

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 Conjunto de pruebas W3C C14N

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,
```

(continué en la próxima página)

(proviene de la página anterior)

```
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

Derechos de autor

Python y esta documentación es:

Copyright © 2001-2023 Python Software Foundation. All rights reserved.

Derechos de autor © 2000 BeOpen.com. Todos los derechos reservados.

Derechos de autor © 1995-2000 Corporation for National Research Initiatives. Todos los derechos reservados.

Derechos de autor © 1991-1995 Stichting Mathematisch Centrum. Todos los derechos reservados.

Consulte [Historia y Licencia](#) para obtener información completa sobre licencias y permisos.

No alfabético

..., [247](#)
 2to3, [247](#)
 >>>, [247](#)
 __all__ (*package variable*), [44](#)
 __dict__ (*module attribute*), [134](#)
 __doc__ (*module attribute*), [134](#)
 __file__ (*module attribute*), [134](#), [135](#)
 __future__, [252](#)
 __import__
 función incorporada, [44](#)
 __loader__ (*module attribute*), [134](#)
 __main__
 módulo, [12](#), [156](#), [168](#)
 __name__ (*module attribute*), [134](#)
 __package__ (*module attribute*), [134](#)
 __slots__, [258](#)
 _frozen (*tipo C*), [47](#)
 _inittab (*tipo C*), [47](#)
 _Py_c_diff (*función C*), [96](#)
 _Py_c_neg (*función C*), [96](#)
 _Py_c_pow (*función C*), [96](#)
 _Py_c_prod (*función C*), [96](#)
 _Py_c_quot (*función C*), [96](#)
 _Py_c_sum (*función C*), [96](#)
 _Py_InitializeMain (*función C*), [190](#)
 _Py_NoneStruct (*variable C*), [202](#)
 _PyBytes_Resize (*función C*), [99](#)
 _PyCFunctionFast (*tipo C*), [204](#)
 _PyCFunctionFastWithKeywords (*tipo C*), [204](#)
 _PyFrameEvalFunction (*tipo C*), [166](#)
 _PyInterpreterState_GetEvalFrameFunc (*función C*), [166](#)
 _PyInterpreterState_SetEvalFrameFunc (*función C*), [166](#)
 _PyObject_New (*función C*), [201](#)
 _PyObject_NewVar (*función C*), [201](#)
 _PyTuple_Resize (*función C*), [121](#)
 _thread

módulo, [163](#)

A

a la espera, [248](#)
 abort(), [44](#)
 abs
 función incorporada, [73](#)
 administrador asincrónico de contexto,
 [248](#)
 administrador de contextos, [249](#)
 alcances anidados, [256](#)
 alias de tipos, [259](#)
 allocfunc (*tipo C*), [239](#)
 anotación, [247](#)
 anotación de función, [251](#)
 anotación de variable, [260](#)
 apagado del intérprete, [253](#)
 API provisional, [257](#)
 archivo binario, [249](#)
 archivo de texto, [259](#)
 argumento, [247](#)
 argumento nombrado, [254](#)
 argumento posicional, [257](#)
 argv (*in module sys*), [159](#)
 ascii
 función incorporada, [65](#)
 atributo, [248](#)

B

BDFL, [248](#)
 binaryfunc (*tipo C*), [240](#)
 bloqueo global del intérprete, [252](#)
 buffer interface
 (see buffer protocol), [78](#)
 buffer object
 (see buffer protocol), [78](#)
 buffer protocol, [78](#)
 builtins
 módulo, [12](#), [156](#), [168](#)
 buscador, [251](#)

buscador basado en ruta, [257](#)
 buscador de entradas de ruta, [257](#)
 bytearray
 objeto, [99](#)
 bytecode, [249](#)
 bytes
 función incorporada, [65](#)
 objeto, [97](#)

C

cadena con triple comilla, [259](#)
 calloc(), [191](#)
 Capsule
 objeto, [145](#)
 cargador, [254](#)
 C-contiguous, [82](#), [249](#)
 clase, [249](#)
 clase base abstracta, [247](#)
 clase de nuevo estilo, [256](#)
 classmethod
 función incorporada, [205](#)
 cleanup functions, [44](#)
 close() (in module os), [168](#)
 CO_FUTURE_DIVISION (variable C), [22](#)
 code object, [132](#)
 codificación de texto, [259](#)
 coerción, [249](#)
 compile
 función incorporada, [45](#)
 complex number
 objeto, [95](#)
 comprensión de conjuntos, [259](#)
 comprensión de diccionarios, [250](#)
 comprensión de listas, [254](#)
 contador de referencias, [258](#)
 contiguo, [249](#)
 contiguous, [82](#)
 copyright (in module sys), [159](#)
 corrutina, [250](#)
 CPython, [250](#)
 create_module (función C), [137](#)

D

decorador, [250](#)
 descrgetfunc (tipo C), [239](#)
 descriptor, [250](#)
 descrsetfunc (tipo C), [239](#)
 despacho único, [259](#)
 destructor (tipo C), [239](#)
 diccionario, [250](#)
 dictionary
 objeto, [125](#)
 división entera, [251](#)
 divmod

 función incorporada, [73](#)
 docstring, [250](#)

E

EAFP, [251](#)
 entorno virtual, [260](#)
 entrada de ruta, [257](#)
 EOFError (built-in exception), [133](#)
 espacio de nombres, [255](#)
 especificador de módulo, [255](#)
 exc_info() (in module sys), [11](#)
 exec_module (función C), [138](#)
 exec_prefix, [4](#)
 executable (in module sys), [158](#)
 exit(), [44](#)
 expresión, [251](#)
 expresión generadora, [252](#)

F

f-string, [251](#)
 file
 objeto, [133](#)
 float
 función incorporada, [74](#)
 floating point
 objeto, [95](#)
 Fortran contiguous, [82](#), [249](#)
 free(), [191](#)
 freefunc (tipo C), [239](#)
 freeze utility, [47](#)
 frozenset
 objeto, [127](#)
 función, [251](#)
 función clave, [254](#)
 función corrutina, [250](#)
 función genérica, [252](#)
 función incorporada
 __import__, [44](#)
 abs, [73](#)
 ascii, [65](#)
 bytes, [65](#)
 classmethod, [205](#)
 compile, [45](#)
 divmod, [73](#)
 float, [74](#)
 hash, [66](#), [219](#)
 int, [74](#)
 len, [66](#), [75](#), [77](#), [123](#), [126](#), [128](#)
 pow, [73](#), [74](#)
 repr, [65](#), [218](#)
 staticmethod, [206](#)
 tuple, [76](#), [124](#)
 type, [66](#)
 function

objeto, 129

G

gancho a entrada de ruta, 257
 generador, 252
 generador asincrónico, 248
 generator, 252
 generator expression, 252
 getattrfunc (tipo C), 239
 getattrofunc (tipo C), 239
 getbufferproc (tipo C), 240
 getiterfunc (tipo C), 239
 GIL, 252
 global interpreter lock, 160

H

hash
 función incorporada, 66, 219
 hash-based pyc, 253
 hashable, 253
 hashfunc (tipo C), 239

I

IDLE, 253
 importador, 253
 importar, 253
 incr_item(), 11, 12
 indicador de tipo, 259
 initproc (tipo C), 239
 immutable, 253
 inquiry (tipo C), 244
 instancemethod
 objeto, 130
 int
 función incorporada, 74
 integer
 objeto, 91
 interactivo, 253
 interpretado, 253
 interpreter lock, 160
 iterable, 253
 iterable asincrónico, 248
 iterador, 254
 iterador asincrónico, 248
 iterador generador, 252
 iterador generador asincrónico, 248
 iternextfunc (tipo C), 239

K

KeyboardInterrupt (*built-in exception*), 31

L

lambda, 254

LBYL, 254

len

 función incorporada, 66, 75, 77, 123, 126, 128

lenfunc (tipo C), 239

list

 objeto, 123

lista, 254

lock, interpreter, 160

long integer

 objeto, 91

LONG_MAX, 92

M

magic

 method, 254

main(), 157, 159

malloc(), 191

mapeado, 255

mapping

 objeto, 125

máquina virtual, 260

memoryview

 objeto, 143

meta buscadores de ruta, 255

metaclass, 255

METH_CLASS (*variable incorporada*), 205

METH_COEXIST (*variable incorporada*), 206

METH_FASTCALL (*variable incorporada*), 205

METH_NOARGS (*variable incorporada*), 205

METH_O (*variable incorporada*), 205

METH_STATIC (*variable incorporada*), 206

METH_VARARGS (*variable incorporada*), 205

method

 magic, 254

 objeto, 131

 special, 259

MethodType (*in module types*), 129, 131

método, 255

método especial, 259

método mágico, 254

module

 objeto, 134

 search path, 12, 156, 158

modules (*in module sys*), 44, 156

ModuleType (*in module types*), 134

módulo, 255

 __main__, 12, 156, 168

 _thread, 163

 builtins, 12, 156, 168

 signal, 31

 sys, 12, 156, 168

módulo de extensión, 251

MRO, 255

mutable, [255](#)

N

newfunc (*tipo C*), [239](#)

nombre calificado, [258](#)

None

objeto, [91](#)

numeric

objeto, [91](#)

número complejo, [249](#)

O

object

code, [132](#)

objeto, [256](#)

bytearray, [99](#)

bytes, [97](#)

Capsule, [145](#)

complex number, [95](#)

dictionary, [125](#)

file, [133](#)

floating point, [95](#)

frozenset, [127](#)

function, [129](#)

instancemethod, [130](#)

integer, [91](#)

list, [123](#)

long integer, [91](#)

mapping, [125](#)

memoryview, [143](#)

method, [131](#)

module, [134](#)

None, [91](#)

numeric, [91](#)

sequence, [97](#)

set, [127](#)

tuple, [121](#)

type, [6](#), [87](#)

objeto archivo, [251](#)

objeto tipo ruta, [257](#)

objetos tipo archivo, [251](#)

objetos tipo binarios, [249](#)

objobjargproc (*tipo C*), [240](#)

objobjproc (*tipo C*), [240](#)

orden de resolución de métodos, [255](#)

OverflowError (*built-in exception*), [92](#), [93](#)

P

package variable

__all__, [44](#)

paquete, [256](#)

paquete de espacios de nombres, [256](#)

paquete provisorio, [257](#)

paquete regular, [258](#)

parámetro, [256](#)

path

module search, [12](#), [156](#), [158](#)

PATH, [12](#)

path (*in module sys*), [12](#), [156](#), [158](#)

PEP, [257](#)

platform (*in module sys*), [159](#)

porción, [257](#)

pow

función incorporada, [73](#), [74](#)

prefix, [4](#)

Py_ABS (*macro C*), [5](#)

Py_AddPendingCall (*función C*), [169](#)

Py_AddPendingCall (), [169](#)

Py_AtExit (*función C*), [44](#)

Py_BEGIN_ALLOW_THREADS, [161](#)

Py_BEGIN_ALLOW_THREADS (*macro C*), [164](#)

Py_BLOCK_THREADS (*macro C*), [164](#)

Py_buffer (*tipo C*), [79](#)

Py_buffer.buf (*miembro C*), [79](#)

Py_buffer.format (*miembro C*), [80](#)

Py_buffer.internal (*miembro C*), [81](#)

Py_buffer.itemsize (*miembro C*), [80](#)

Py_buffer.len (*miembro C*), [80](#)

Py_buffer.ndim (*miembro C*), [80](#)

Py_buffer.obj (*miembro C*), [80](#)

Py_buffer.readonly (*miembro C*), [80](#)

Py_buffer.shape (*miembro C*), [80](#)

Py_buffer.strides (*miembro C*), [81](#)

Py_buffer.suboffsets (*miembro C*), [81](#)

Py_BuildValue (*función C*), [55](#)

Py_BytesMain (*función C*), [17](#)

Py_BytesWarningFlag (*variable C*), [154](#)

Py_CHARMASK (*macro C*), [5](#)

Py_CLEAR (*función C*), [23](#)

Py_CompileString (*función C*), [20](#)

Py_CompileString (), [21](#)

Py_CompileStringExFlags (*función C*), [20](#)

Py_CompileStringFlags (*función C*), [20](#)

Py_CompileStringObject (*función C*), [20](#)

Py_complex (*tipo C*), [96](#)

Py_DebugFlag (*variable C*), [154](#)

Py_DecodeLocale (*función C*), [40](#)

Py_DECREF (*función C*), [23](#)

Py_DECREF (), [7](#)

Py_DEPRECATED (*macro C*), [5](#)

Py_DontWriteBytecodeFlag (*variable C*), [154](#)

Py_Ellipsis (*variable C*), [143](#)

Py_EncodeLocale (*función C*), [41](#)

Py_END_ALLOW_THREADS, [161](#)

Py_END_ALLOW_THREADS (*macro C*), [164](#)

Py_EndInterpreter (*función C*), [168](#)

Py_EnterRecursiveCall (*función C*), [34](#)

Py_eval_input (*variable C*), [21](#)

- Py_Exit (función C), 44
- Py_ExitStatusException (función C), 177
- Py_False (variable C), 94
- Py_FatalError (función C), 44
- Py_FatalError(), 159
- Py_FdIsInteractive (función C), 39
- Py_file_input (variable C), 21
- Py_Finalize (función C), 157
- Py_FinalizeEx (función C), 156
- Py_FinalizeEx(), 44, 156, 168
- Py_FrozenFlag (variable C), 154
- Py_GenericAlias (función C), 152
- Py_GenericAliasType (variable C), 152
- Py_GetArgcArgv (función C), 189
- Py_GetBuildInfo (función C), 159
- Py_GetCompiler (función C), 159
- Py_GetCopyright (función C), 159
- Py_GETENV (macro C), 5
- Py_GetExecPrefix (función C), 158
- Py_GetExecPrefix(), 12
- Py_GetPath (función C), 158
- Py_GetPath(), 12, 157, 158
- Py_GetPlatform (función C), 159
- Py_GetPrefix (función C), 157
- Py_GetPrefix(), 12
- Py_GetProgramFullPath (función C), 158
- Py_GetProgramFullPath(), 12
- Py_GetProgramName (función C), 157
- Py_GetPythonHome (función C), 160
- Py_GetVersion (función C), 159
- Py_HashRandomizationFlag (variable C), 155
- Py_IgnoreEnvironmentFlag (variable C), 155
- Py_INCREF (función C), 23
- Py_INCREF(), 7
- Py_Initialize (función C), 156
- Py_Initialize(), 12, 157, 168
- Py_InitializeEx (función C), 156
- Py_InitializeFromConfig (función C), 185
- Py_InspectFlag (variable C), 155
- Py_InteractiveFlag (variable C), 155
- Py_IS_TYPE (función C), 203
- Py_IsInitialized (función C), 156
- Py_IsInitialized(), 12
- Py_IsolatedFlag (variable C), 155
- Py_LeaveRecursiveCall (función C), 34
- Py_LegacyWindowsFSEncodingFlag (variable C), 155
- Py_LegacyWindowsStdioFlag (variable C), 155
- Py_Main (función C), 17
- Py_MAX (macro C), 5
- Py_MEMBER_SIZE (macro C), 5
- Py_MIN (macro C), 5
- Py_mod_create (macro C), 137
- Py_mod_exec (macro C), 138
- Py_NewInterpreter (función C), 168
- Py_None (variable C), 91
- Py_NoSiteFlag (variable C), 155
- Py_NotImplemented (variable C), 63
- Py_NoUserSiteDirectory (variable C), 155
- Py_OptimizeFlag (variable C), 156
- Py_PreInitialize (función C), 179
- Py_PreInitializeFromArgs (función C), 179
- Py_PreInitializeFromBytesArgs (función C), 179
- Py_PRINT_RAW, 134
- Py_QuietFlag (variable C), 156
- Py_REFCNT (macro C), 203
- Py_ReprEnter (función C), 34
- Py_ReprLeave (función C), 35
- Py_RETURN_FALSE (macro C), 94
- Py_RETURN_NONE (macro C), 91
- Py_RETURN_NOTIMPLEMENTED (macro C), 63
- Py_RETURN_RICHCOMPARE (macro C), 225
- Py_RETURN_TRUE (macro C), 94
- Py_RunMain (función C), 189
- Py_SET_REFCNT (función C), 203
- Py_SET_SIZE (función C), 203
- Py_SET_TYPE (función C), 203
- Py_SetPath (función C), 158
- Py_SetPath(), 158
- Py_SetProgramName (función C), 157
- Py_SetProgramName(), 12, 156, 158
- Py_SetPythonHome (función C), 160
- Py_SetStandardStreamEncoding (función C), 157
- Py_single_input (variable C), 21
- Py_SIZE (macro C), 203
- Py_ssize_t (tipo C), 10
- PY_SSIZE_T_MAX, 93
- Py_STRINGIFY (macro C), 5
- Py_TPFLAGS_BASE_EXC_SUBCLASS (variable incorporada), 222
- Py_TPFLAGS_BASETYPE (variable incorporada), 221
- Py_TPFLAGS_BYTES_SUBCLASS (variable incorporada), 222
- Py_TPFLAGS_DEFAULT (variable incorporada), 222
- Py_TPFLAGS_DICT_SUBCLASS (variable incorporada), 222
- Py_TPFLAGS_HAVE_FINALIZE (variable incorporada), 222
- Py_TPFLAGS_HAVE_GC (variable incorporada), 221
- Py_TPFLAGS_HAVE_VECTORCALL (variable incorporada), 223
- Py_TPFLAGS_HEAPTYPE (variable incorporada), 221
- Py_TPFLAGS_LIST_SUBCLASS (variable incorporada), 222
- Py_TPFLAGS_LONG_SUBCLASS (variable incorporada), 222

- `Py_TPFLAGS_METHOD_DESCRIPTOR` (variable incorporada), 222
- `Py_TPFLAGS_READY` (variable incorporada), 221
- `Py_TPFLAGS_READYING` (variable incorporada), 221
- `Py_TPFLAGS_TUPLE_SUBCLASS` (variable incorporada), 222
- `Py_TPFLAGS_TYPE_SUBCLASS` (variable incorporada), 222
- `Py_TPFLAGS_UNICODE_SUBCLASS` (variable incorporada), 222
- `Py_tracefunc` (tipo C), 170
- `Py_True` (variable C), 94
- `Py_tss_NEEDS_INIT` (macro C), 172
- `Py_tss_t` (tipo C), 172
- `Py_TYPE` (macro C), 202
- `Py_UCS1` (tipo C), 101
- `Py_UCS2` (tipo C), 101
- `Py_UCS4` (tipo C), 101
- `Py_UNBLOCK_THREADS` (macro C), 164
- `Py_UnbufferedStdioFlag` (variable C), 156
- `Py_UNICODE` (tipo C), 101
- `Py_UNICODE_IS_HIGH_SURROGATE` (macro C), 104
- `Py_UNICODE_IS_LOW_SURROGATE` (macro C), 104
- `Py_UNICODE_IS_SURROGATE` (macro C), 104
- `Py_UNICODE_ISALNUM` (función C), 103
- `Py_UNICODE_ISALPHA` (función C), 103
- `Py_UNICODE_ISDECIMAL` (función C), 103
- `Py_UNICODE_ISDIGIT` (función C), 103
- `Py_UNICODE_ISLINEBREAK` (función C), 103
- `Py_UNICODE_ISLOWER` (función C), 103
- `Py_UNICODE_ISNUMERIC` (función C), 103
- `Py_UNICODE_ISPRINTABLE` (función C), 104
- `Py_UNICODE_ISSPACE` (función C), 103
- `Py_UNICODE_ISTITLE` (función C), 103
- `Py_UNICODE_ISUPPER` (función C), 103
- `Py_UNICODE_JOIN_SURROGATES` (macro C), 104
- `Py_UNICODE_TODECIMAL` (función C), 104
- `Py_UNICODE_TODIGIT` (función C), 104
- `Py_UNICODE_TOLOWER` (función C), 104
- `Py_UNICODE_TONUMERIC` (función C), 104
- `Py_UNICODE_TOTITLE` (función C), 104
- `Py_UNICODE_TOUPPER` (función C), 104
- `Py_UNREACHABLE` (macro C), 5
- `Py_UNUSED` (macro C), 5
- `Py_VaBuildValue` (función C), 57
- `Py_VECTORCALL_ARGUMENTS_OFFSET` (macro C), 68
- `Py_VerboseFlag` (variable C), 156
- `Py_VISIT` (función C), 244
- `Py_XDECREF` (función C), 23
- `Py_XDECREF()`, 12
- `Py_XINCREf` (función C), 23
- `PyAnySet_Check` (función C), 128
- `PyAnySet_CheckExact` (función C), 128
- `PyArg_Parse` (función C), 54
- `PyArg_ParseTuple` (función C), 54
- `PyArg_ParseTupleAndKeywords` (función C), 54
- `PyArg_UnpackTuple` (función C), 54
- `PyArg_ValidateKeywordArguments` (función C), 54
- `PyArg_VaParse` (función C), 54
- `PyArg_VaParseTupleAndKeywords` (función C), 54
- `PyASCIIObject` (tipo C), 101
- `PyAsyncMethods` (tipo C), 238
- `PyAsyncMethods.am_aiter` (miembro C), 238
- `PyAsyncMethods.am_anext` (miembro C), 238
- `PyAsyncMethods.am_await` (miembro C), 238
- `PyBool_Check` (función C), 94
- `PyBool_FromLong` (función C), 94
- `PyBUF_ANY_CONTIGUOUS` (macro C), 82
- `PyBUF_C_CONTIGUOUS` (macro C), 82
- `PyBUF_CONTIG` (macro C), 83
- `PyBUF_CONTIG_RO` (macro C), 83
- `PyBUF_F_CONTIGUOUS` (macro C), 82
- `PyBUF_FORMAT` (macro C), 81
- `PyBUF_FULL` (macro C), 83
- `PyBUF_FULL_RO` (macro C), 83
- `PyBUF_INDIRECT` (macro C), 82
- `PyBUF_ND` (macro C), 82
- `PyBUF_RECORDS` (macro C), 83
- `PyBUF_RECORDS_RO` (macro C), 83
- `PyBUF_SIMPLE` (macro C), 82
- `PyBUF_STRIDED` (macro C), 83
- `PyBUF_STRIDED_RO` (macro C), 83
- `PyBUF_STRIDES` (macro C), 82
- `PyBUF_WRITABLE` (macro C), 81
- `PyBuffer_FillContiguousStrides` (función C), 85
- `PyBuffer_FillInfo` (función C), 85
- `PyBuffer_FromContiguous` (función C), 85
- `PyBuffer_GetPointer` (función C), 85
- `PyBuffer_IsContiguous` (función C), 85
- `PyBuffer_Release` (función C), 85
- `PyBuffer_SizeFromFormat` (función C), 85
- `PyBuffer_ToContiguous` (función C), 85
- `PyBufferProcs`, 79
- `PyBufferProcs` (tipo C), 237
- `PyBufferProcs.bf_getbuffer` (miembro C), 237
- `PyBufferProcs.bf_releasebuffer` (miembro C), 237
- `PyByteArray_AS_STRING` (función C), 100
- `PyByteArray_AsString` (función C), 99
- `PyByteArray_Check` (función C), 99
- `PyByteArray_CheckExact` (función C), 99
- `PyByteArray_Concat` (función C), 99
- `PyByteArray_FromObject` (función C), 99

- PyByteArray_FromStringAndSize (función C), 99
- PyByteArray_GET_SIZE (función C), 100
- PyByteArray_Resize (función C), 100
- PyByteArray_Size (función C), 99
- PyByteArray_Type (variable C), 99
- PyByteArrayObject (tipo C), 99
- PyBytes_AS_STRING (función C), 98
- PyBytes_AsString (función C), 98
- PyBytes_AsStringAndSize (función C), 98
- PyBytes_Check (función C), 97
- PyBytes_CheckExact (función C), 97
- PyBytes_Concat (función C), 98
- PyBytes_ConcatAndDel (función C), 99
- PyBytes_FromFormat (función C), 97
- PyBytes_FromFormatV (función C), 98
- PyBytes_FromObject (función C), 98
- PyBytes_FromString (función C), 97
- PyBytes_FromStringAndSize (función C), 97
- PyBytes_GET_SIZE (función C), 98
- PyBytes_Size (función C), 98
- PyBytes_Type (variable C), 97
- PyBytesObject (tipo C), 97
- PyCallable_Check (función C), 72
- PyCallIter_Check (función C), 141
- PyCallIter_New (función C), 141
- PyCallIter_Type (variable C), 141
- PyCapsule (tipo C), 145
- PyCapsule_CheckExact (función C), 145
- PyCapsule_Destructor (tipo C), 145
- PyCapsule_GetContext (función C), 145
- PyCapsule_GetDestructor (función C), 145
- PyCapsule_GetName (función C), 145
- PyCapsule_GetPointer (función C), 145
- PyCapsule_Import (función C), 145
- PyCapsule_IsValid (función C), 146
- PyCapsule_New (función C), 145
- PyCapsule_SetContext (función C), 146
- PyCapsule_SetDestructor (función C), 146
- PyCapsule_SetName (función C), 146
- PyCapsule_SetPointer (función C), 146
- PyCell_Check (función C), 131
- PyCell_Get (función C), 131
- PyCell_GET (función C), 131
- PyCell_New (función C), 131
- PyCell_Set (función C), 131
- PyCell_SET (función C), 132
- PyCell_Type (variable C), 131
- PyCellObject (tipo C), 131
- PyCFunction (tipo C), 203
- PyCFunctionWithKeywords (tipo C), 204
- PyCMethod (tipo C), 204
- PyCode_Check (función C), 132
- PyCode_GetNumFree (función C), 132
- PyCode_New (función C), 132
- PyCode_NewEmpty (función C), 132
- PyCode_NewWithPosOnlyArgs (función C), 132
- PyCode_Type (variable C), 132
- PyCodec_BackslashReplaceErrors (función C), 61
- PyCodec_Decompile (función C), 60
- PyCodec_Decoder (función C), 60
- PyCodec_Encode (función C), 60
- PyCodec_Encoder (función C), 60
- PyCodec_IgnoreErrors (función C), 61
- PyCodec_IncrementalDecoder (función C), 60
- PyCodec_IncrementalEncoder (función C), 60
- PyCodec_KnownEncoding (función C), 60
- PyCodec_LookupError (función C), 61
- PyCodec_NameReplaceErrors (función C), 61
- PyCodec_Register (función C), 60
- PyCodec_RegisterError (función C), 61
- PyCodec_ReplaceErrors (función C), 61
- PyCodec_StreamReader (función C), 60
- PyCodec_StreamWriter (función C), 60
- PyCodec_StrictErrors (función C), 61
- PyCodec_XMLCharRefReplaceErrors (función C), 61
- PyCodeObject (tipo C), 132
- PyCompactUnicodeObject (tipo C), 101
- PyCompilerFlags (tipo C), 21
- PyCompilerFlags.cf_feature_version (miembro C), 21
- PyCompilerFlags.cf_flags (miembro C), 21
- PyComplex_AsCComplex (función C), 97
- PyComplex_Check (función C), 96
- PyComplex_CheckExact (función C), 96
- PyComplex_FromCComplex (función C), 96
- PyComplex_FromDoubles (función C), 96
- PyComplex_ImagAsDouble (función C), 97
- PyComplex_RealAsDouble (función C), 96
- PyComplex_Type (variable C), 96
- PyComplexObject (tipo C), 96
- PyConfig (tipo C), 180
- PyConfig_Clear (función C), 181
- PyConfig_InitIsolatedConfig (función C), 180
- PyConfig_InitPythonConfig (función C), 180
- PyConfig_Read (función C), 181
- PyConfig_SetArgv (función C), 180
- PyConfig_SetBytesArgv (función C), 180
- PyConfig_SetBytesString (función C), 180
- PyConfig_SetString (función C), 180
- PyConfig_SetWideStringList (función C), 180
- PyConfig._use_peg_parser (miembro C), 184
- PyConfig.argv (miembro C), 181
- PyConfig.base_exec_prefix (miembro C), 181
- PyConfig.base_executable (miembro C), 181
- PyConfig.base_prefix (miembro C), 181

- PyConfig.buffered_stdio (*miembro C*), 181
- PyConfig.bytes_warning (*miembro C*), 181
- PyConfig.check_hash_pycs_mode (*miembro C*), 182
- PyConfig.configure_c_stdio (*miembro C*), 182
- PyConfig.dev_mode (*miembro C*), 182
- PyConfig.dump_refs (*miembro C*), 182
- PyConfig.exec_prefix (*miembro C*), 182
- PyConfig.executable (*miembro C*), 182
- PyConfig.faulthandler (*miembro C*), 182
- PyConfig.filesystem_encoding (*miembro C*), 182
- PyConfig.filesystem_errors (*miembro C*), 182
- PyConfig.hash_seed (*miembro C*), 182
- PyConfig.home (*miembro C*), 182
- PyConfig.import_time (*miembro C*), 182
- PyConfig.inspect (*miembro C*), 182
- PyConfig.install_signal_handlers (*miembro C*), 182
- PyConfig.interactive (*miembro C*), 182
- PyConfig.isolated (*miembro C*), 182
- PyConfig.legacy_windows_stdio (*miembro C*), 183
- PyConfig.malloc_stats (*miembro C*), 183
- PyConfig.module_search_paths (*miembro C*), 183
- PyConfig.module_search_paths_set (*miembro C*), 183
- PyConfig.optimization_level (*miembro C*), 183
- PyConfig.parse_argv (*miembro C*), 183
- PyConfig.parser_debug (*miembro C*), 183
- PyConfig.pathconfig_warnings (*miembro C*), 183
- PyConfig.platlibdir (*miembro C*), 181
- PyConfig.prefix (*miembro C*), 183
- PyConfig.program_name (*miembro C*), 183
- PyConfig.pycache_prefix (*miembro C*), 183
- PyConfig.pythonpath_env (*miembro C*), 183
- PyConfig.quiet (*miembro C*), 183
- PyConfig.run_command (*miembro C*), 184
- PyConfig.run_filename (*miembro C*), 184
- PyConfig.run_module (*miembro C*), 184
- PyConfig.show_ref_count (*miembro C*), 184
- PyConfig.site_import (*miembro C*), 184
- PyConfig.skip_source_first_line (*miembro C*), 184
- PyConfig.stdio_encoding (*miembro C*), 184
- PyConfig.stdio_errors (*miembro C*), 184
- PyConfig.tracemalloc (*miembro C*), 184
- PyConfig.use_environment (*miembro C*), 184
- PyConfig.use_hash_seed (*miembro C*), 182
- PyConfig.user_site_directory (*miembro C*), 184
- PyConfig.verbose (*miembro C*), 184
- PyConfig.warnoptions (*miembro C*), 184
- PyConfig.write_bytecode (*miembro C*), 184
- PyConfig.xoptions (*miembro C*), 184
- PyContext (*tipo C*), 147
- PyContext_CheckExact (*función C*), 148
- PyContext_Copy (*función C*), 148
- PyContext_CopyCurrent (*función C*), 148
- PyContext_Enter (*función C*), 148
- PyContext_Exit (*función C*), 148
- PyContext_New (*función C*), 148
- PyContext_Type (*variable C*), 147
- PyContextToken (*tipo C*), 147
- PyContextToken_CheckExact (*función C*), 148
- PyContextToken_Type (*variable C*), 148
- PyContextVar (*tipo C*), 147
- PyContextVar_CheckExact (*función C*), 148
- PyContextVar_Get (*función C*), 148
- PyContextVar_New (*función C*), 148
- PyContextVar_Reset (*función C*), 149
- PyContextVar_Set (*función C*), 148
- PyContextVar_Type (*variable C*), 148
- PyCoro_CheckExact (*función C*), 147
- PyCoro_New (*función C*), 147
- PyCoro_Type (*variable C*), 147
- PyCoroObject (*tipo C*), 147
- PyDate_Check (*función C*), 149
- PyDate_CheckExact (*función C*), 149
- PyDate_FromDate (*función C*), 150
- PyDate_FromTimestamp (*función C*), 151
- PyDateTime_Check (*función C*), 149
- PyDateTime_CheckExact (*función C*), 149
- PyDateTime_DATE_GET_FOLD (*función C*), 151
- PyDateTime_DATE_GET_HOUR (*función C*), 150
- PyDateTime_DATE_GET_MICROSECOND (*función C*), 151
- PyDateTime_DATE_GET_MINUTE (*función C*), 151
- PyDateTime_DATE_GET_SECOND (*función C*), 151
- PyDateTime_DELTA_GET_DAYS (*función C*), 151
- PyDateTime_DELTA_GET_MICROSECONDS (*función C*), 151
- PyDateTime_DELTA_GET_SECONDS (*función C*), 151
- PyDateTime_FromDateAndTime (*función C*), 150
- PyDateTime_FromDateAndTimeAndFold (*función C*), 150
- PyDateTime_FromTimestamp (*función C*), 151
- PyDateTime_GET_DAY (*función C*), 150
- PyDateTime_GET_MONTH (*función C*), 150
- PyDateTime_GET_YEAR (*función C*), 150
- PyDateTime_TIME_GET_FOLD (*función C*), 151
- PyDateTime_TIME_GET_HOUR (*función C*), 151
- PyDateTime_TIME_GET_MICROSECOND (*función C*), 151

- PyDateTime_TIME_GET_MINUTE (función C), 151
 PyDateTime_TIME_GET_SECOND (función C), 151
 PyDateTime_TimeZone_UTC (variable C), 149
 PyDelta_Check (función C), 149
 PyDelta_CheckExact (función C), 149
 PyDelta_FromDSU (función C), 150
 PyDescr_IsData (función C), 141
 PyDescr_NewClassMethod (función C), 141
 PyDescr_NewGetSet (función C), 141
 PyDescr_NewMember (función C), 141
 PyDescr_NewMethod (función C), 141
 PyDescr_NewWrapper (función C), 141
 PyDict_Check (función C), 125
 PyDict_CheckExact (función C), 125
 PyDict_Clear (función C), 125
 PyDict_Contains (función C), 125
 PyDict_Copy (función C), 125
 PyDict_DelItem (función C), 125
 PyDict_DelItemString (función C), 125
 PyDict_GetItem (función C), 125
 PyDict_GetItemString (función C), 126
 PyDict_GetItemWithError (función C), 126
 PyDict_Items (función C), 126
 PyDict_Keys (función C), 126
 PyDict_Merge (función C), 127
 PyDict_MergeFromSeq2 (función C), 127
 PyDict_New (función C), 125
 PyDict_Next (función C), 126
 PyDict_SetDefault (función C), 126
 PyDict_SetItem (función C), 125
 PyDict_SetItemString (función C), 125
 PyDict_Size (función C), 126
 PyDict_Type (variable C), 125
 PyDict_Update (función C), 127
 PyDict_Values (función C), 126
 PyDictObject (tipo C), 125
 PyDictProxy_New (función C), 125
 PyDoc_STR (macro C), 6
 PyDoc_STRVAR (macro C), 6
 PyErr_BadArgument (función C), 26
 PyErr_BadInternalCall (función C), 28
 PyErr_CheckSignals (función C), 31
 PyErr_Clear (función C), 26
 PyErr_Clear(), 10, 12
 PyErr_ExceptionMatches (función C), 30
 PyErr_ExceptionMatches(), 12
 PyErr_Fetch (función C), 30
 PyErr_Format (función C), 26
 PyErr_FormatV (función C), 26
 PyErr_GetExcInfo (función C), 31
 PyErr_GivenExceptionMatches (función C), 30
 PyErr_NewException (función C), 32
 PyErr_NewExceptionWithDoc (función C), 32
 PyErr_NoMemory (función C), 27
 PyErr_NormalizeException (función C), 30
 PyErr_Occurred (función C), 30
 PyErr_Occurred(), 10
 PyErr_Print (función C), 26
 PyErr_PrintEx (función C), 26
 PyErr_ResourceWarning (función C), 29
 PyErr_Restore (función C), 30
 PyErr_SetExcFromWindowsErr (función C), 27
 PyErr_SetExcFromWindowsErrWithFilename (función C), 28
 PyErr_SetExcFromWindowsErrWithFilenameObject (función C), 27
 PyErr_SetExcFromWindowsErrWithFilenameObjects (función C), 28
 PyErr_SetExcInfo (función C), 31
 PyErr_SetFromErrno (función C), 27
 PyErr_SetFromErrnoWithFilename (función C), 27
 PyErr_SetFromErrnoWithFilenameObject (función C), 27
 PyErr_SetFromErrnoWithFilenameObjects (función C), 27
 PyErr_SetFromWindowsErr (función C), 27
 PyErr_SetFromWindowsErrWithFilename (función C), 27
 PyErr_SetImportError (función C), 28
 PyErr_SetImportErrorSubclass (función C), 28
 PyErr_SetInterrupt (función C), 31
 PyErr_SetNone (función C), 26
 PyErr_SetObject (función C), 26
 PyErr_SetString (función C), 26
 PyErr_SetString(), 10
 PyErr_SyntaxLocation (función C), 28
 PyErr_SyntaxLocationEx (función C), 28
 PyErr_SyntaxLocationObject (función C), 28
 PyErr_WarnEx (función C), 29
 PyErr_WarnExplicit (función C), 29
 PyErr_WarnExplicitObject (función C), 29
 PyErr_WarnFormat (función C), 29
 PyErr_WriteUnraisable (función C), 26
 PyEval_AcquireLock (función C), 167
 PyEval_AcquireThread (función C), 167
 PyEval_AcquireThread(), 163
 PyEval_EvalCode (función C), 20
 PyEval_EvalCodeEx (función C), 21
 PyEval_EvalFrame (función C), 21
 PyEval_EvalFrameEx (función C), 21
 PyEval_GetBuiltins (función C), 59
 PyEval_GetFrame (función C), 59
 PyEval_GetFuncDesc (función C), 59
 PyEval_GetFuncName (función C), 59
 PyEval_GetGlobals (función C), 59
 PyEval_GetLocals (función C), 59
 PyEval_InitThreads (función C), 163

PyEval_InitThreads(), 156
PyEval_MergeCompilerFlags (función C), 21
PyEval_ReleaseLock (función C), 167
PyEval_ReleaseThread (función C), 167
PyEval_ReleaseThread(), 163
PyEval_RestoreThread (función C), 163
PyEval_RestoreThread(), 161, 163
PyEval_SaveThread (función C), 163
PyEval_SaveThread(), 161, 163
PyEval_SetProfile (función C), 171
PyEval_SetTrace (función C), 171
PyEval_ThreadsInitialized (función C), 163
PyExc_ArithmeticError, 35
PyExc_AssertionError, 35
PyExc_AttributeError, 35
PyExc_BaseException, 35
PyExc_BlockingIOError, 35
PyExc_BrokenPipeError, 35
PyExc_BufferError, 35
PyExc_BytesWarning, 37
PyExc_ChildProcessError, 35
PyExc_ConnectionAbortedError, 35
PyExc_ConnectionError, 35
PyExc_ConnectionRefusedError, 35
PyExc_ConnectionResetError, 35
PyExc_DeprecationWarning, 37
PyExc_EnvironmentError, 36
PyExc_EOFError, 35
PyExc_Exception, 35
PyExc_FileExistsError, 35
PyExc_FileNotFoundError, 35
PyExc_FloatingPointError, 35
PyExc_FutureWarning, 37
PyExc_GeneratorExit, 35
PyExc_ImportError, 35
PyExc_ImportWarning, 37
PyExc_IndentationError, 35
PyExc_IndexError, 35
PyExc_InterruptedError, 35
PyExc_IOError, 36
PyExc_IsADirectoryError, 35
PyExc_KeyboardInterrupt, 35
PyExc_KeyError, 35
PyExc_LookupError, 35
PyExc_MemoryError, 35
PyExc_ModuleNotFoundError, 35
PyExc_NameError, 35
PyExc_NotADirectoryError, 35
PyExc_NotImplementedError, 35
PyExc_OSError, 35
PyExc_OverflowError, 35
PyExc_PendingDeprecationWarning, 37
PyExc_PermissionError, 35
PyExc_ProcessLookupError, 35
PyExc_RecursionError, 35
PyExc_ReferenceError, 35
PyExc_ResourceWarning, 37
PyExc_RuntimeError, 35
PyExc_RuntimeWarning, 37
PyExc_StopAsyncIteration, 35
PyExc_StopIteration, 35
PyExc_SyntaxError, 35
PyExc_SyntaxWarning, 37
PyExc_SystemError, 35
PyExc_SystemExit, 35
PyExc_TabError, 35
PyExc_TimeoutError, 35
PyExc_TypeError, 35
PyExc_UnboundLocalError, 35
PyExc_UnicodeDecodeError, 35
PyExc_UnicodeEncodeError, 35
PyExc_UnicodeError, 35
PyExc_UnicodeTranslateError, 35
PyExc_UnicodeWarning, 37
PyExc_UserWarning, 37
PyExc_ValueError, 35
PyExc_Warning, 37
PyExc_WindowsError, 36
PyExc_ZeroDivisionError, 35
PyException_GetCause (función C), 32
PyException_GetContext (función C), 32
PyException_GetTraceback (función C), 32
PyException_SetCause (función C), 33
PyException_SetContext (función C), 32
PyException_SetTraceback (función C), 32
PyFile_FromFd (función C), 133
PyFile_GetLine (función C), 133
PyFile_SetOpenCodeHook (función C), 133
PyFile_WriteObject (función C), 134
PyFile_WriteString (función C), 134
PyFloat_AS_DOUBLE (función C), 95
PyFloat_AsDouble (función C), 95
PyFloat_Check (función C), 95
PyFloat_CheckExact (función C), 95
PyFloat_FromDouble (función C), 95
PyFloat_FromString (función C), 95
PyFloat_GetInfo (función C), 95
PyFloat_GetMax (función C), 95
PyFloat_GetMin (función C), 95
PyFloat_Type (variable C), 95
PyFloatObject (tipo C), 95
PyFrame_GetBack (función C), 59
PyFrame_GetCode (función C), 59
PyFrame_GetLineNumber (función C), 59
PyFrameObject (tipo C), 21
PyFrozenSet_Check (función C), 128
PyFrozenSet_CheckExact (función C), 128
PyFrozenSet_New (función C), 128

- PyFrozenSet_Type (variable C), 128
- PyFunction_Check (función C), 129
- PyFunction_GetAnnotations (función C), 130
- PyFunction_GetClosure (función C), 130
- PyFunction_GetCode (función C), 129
- PyFunction_GetDefaults (función C), 130
- PyFunction_GetGlobals (función C), 129
- PyFunction_GetModule (función C), 129
- PyFunction_New (función C), 129
- PyFunction_NewWithQualName (función C), 129
- PyFunction_SetAnnotations (función C), 130
- PyFunction_SetClosure (función C), 130
- PyFunction_SetDefaults (función C), 130
- PyFunction_Type (variable C), 129
- PyFunctionObject (tipo C), 129
- PyGen_Check (función C), 146
- PyGen_CheckExact (función C), 146
- PyGen_New (función C), 146
- PyGen_NewWithQualName (función C), 147
- PyGen_Type (variable C), 146
- PyGenObject (tipo C), 146
- PyGetSetDef (tipo C), 207
- PyGILState_Check (función C), 164
- PyGILState_Ensure (función C), 163
- PyGILState_GetThisThreadState (función C), 164
- PyGILState_Release (función C), 164
- PyImport_AddModule (función C), 45
- PyImport_AddModuleObject (función C), 45
- PyImport_AppendInittab (función C), 47
- PyImport_ExecCodeModule (función C), 45
- PyImport_ExecCodeModuleEx (función C), 46
- PyImport_ExecCodeModuleObject (función C), 46
- PyImport_ExecCodeModuleWithPathnames (función C), 46
- PyImport_ExtendInittab (función C), 48
- PyImport_FrozenModules (variable C), 47
- PyImport_GetImporter (función C), 47
- PyImport_GetMagicNumber (función C), 46
- PyImport_GetMagicTag (función C), 46
- PyImport_GetModule (función C), 46
- PyImport_GetModuleDict (función C), 46
- PyImport_Import (función C), 45
- PyImport_ImportFrozenModule (función C), 47
- PyImport_ImportFrozenModuleObject (función C), 47
- PyImport_ImportModule (función C), 44
- PyImport_ImportModuleEx (función C), 44
- PyImport_ImportModuleLevel (función C), 45
- PyImport_ImportModuleLevelObject (función C), 45
- PyImport_ImportModuleNoBlock (función C), 44
- PyImport_ReloadModule (función C), 45
- PyIndex_Check (función C), 75
- PyInstanceMethod_Check (función C), 130
- PyInstanceMethod_Function (función C), 130
- PyInstanceMethod_GET_FUNCTION (función C), 130
- PyInstanceMethod_New (función C), 130
- PyInstanceMethod_Type (variable C), 130
- PyInterpreterState (tipo C), 162
- PyInterpreterState_Clear (función C), 165
- PyInterpreterState_Delete (función C), 165
- PyInterpreterState_Get (función C), 165
- PyInterpreterState_GetDict (función C), 166
- PyInterpreterState_GetID (función C), 166
- PyInterpreterState_Head (función C), 171
- PyInterpreterState_Main (función C), 171
- PyInterpreterState_New (función C), 165
- PyInterpreterState_Next (función C), 171
- PyInterpreterState_ThreadHead (función C), 171
- PyIter_Check (función C), 78
- PyIter_Next (función C), 78
- PyList_Append (función C), 124
- PyList_AsTuple (función C), 124
- PyList_Check (función C), 123
- PyList_CheckExact (función C), 123
- PyList_GET_ITEM (función C), 124
- PyList_GET_SIZE (función C), 123
- PyList_GetItem (función C), 123
- PyList_GetItem(), 9
- PyList_GetSlice (función C), 124
- PyList_Insert (función C), 124
- PyList_New (función C), 123
- PyList_Reverse (función C), 124
- PyList_SET_ITEM (función C), 124
- PyList_SetItem (función C), 124
- PyList_SetItem(), 8
- PyList_SetSlice (función C), 124
- PyList_Size (función C), 123
- PyList_Sort (función C), 124
- PyList_Type (variable C), 123
- PyListObject (tipo C), 123
- PyLong_AsDouble (función C), 94
- PyLong_AsLong (función C), 92
- PyLong_AsLongAndOverflow (función C), 92
- PyLong_AsLongLong (función C), 92
- PyLong_AsLongLongAndOverflow (función C), 93
- PyLong_AsSize_t (función C), 93
- PyLong_AsSsize_t (función C), 93
- PyLong_AsUnsignedLong (función C), 93
- PyLong_AsUnsignedLongLong (función C), 93
- PyLong_AsUnsignedLongLongMask (función C), 94
- PyLong_AsUnsignedLongMask (función C), 93
- PyLong_AsVoidPtr (función C), 94

- PyLong_Check (*función C*), 91
- PyLong_CheckExact (*función C*), 91
- PyLong_FromDouble (*función C*), 92
- PyLong_FromLong (*función C*), 91
- PyLong_FromLongLong (*función C*), 91
- PyLong_FromSize_t (*función C*), 91
- PyLong_FromSsize_t (*función C*), 91
- PyLong_FromString (*función C*), 92
- PyLong_FromUnicode (*función C*), 92
- PyLong_FromUnicodeObject (*función C*), 92
- PyLong_FromUnsignedLong (*función C*), 91
- PyLong_FromUnsignedLongLong (*función C*), 92
- PyLong_FromVoidPtr (*función C*), 92
- PyLong_Type (*variable C*), 91
- PyLongObject (*tipo C*), 91
- PyMapping_Check (*función C*), 77
- PyMapping_DelItem (*función C*), 77
- PyMapping_DelItemString (*función C*), 77
- PyMapping_GetItemString (*función C*), 77
- PyMapping_HasKey (*función C*), 77
- PyMapping_HasKeyString (*función C*), 77
- PyMapping_Items (*función C*), 78
- PyMapping_Keys (*función C*), 78
- PyMapping_Length (*función C*), 77
- PyMapping_SetItemString (*función C*), 77
- PyMapping_Size (*función C*), 77
- PyMapping_Values (*función C*), 78
- PyMappingMethods (*tipo C*), 235
- PyMappingMethods.mp_ass_subscript (*miembro C*), 235
- PyMappingMethods.mp_length (*miembro C*), 235
- PyMappingMethods.mp_subscript (*miembro C*), 235
- PyMarshal_ReadLastObjectFromFile (*función C*), 48
- PyMarshal_ReadLongFromFile (*función C*), 48
- PyMarshal_ReadObjectFromFile (*función C*), 48
- PyMarshal_ReadObjectFromString (*función C*), 49
- PyMarshal_ReadShortFromFile (*función C*), 48
- PyMarshal_WriteLongToFile (*función C*), 48
- PyMarshal_WriteObjectToFile (*función C*), 48
- PyMarshal_WriteObjectToString (*función C*), 48
- PyMem_Calloc (*función C*), 193
- PyMem_Del (*función C*), 194
- PYMEM_DOMAIN_MEM (*macro C*), 196
- PYMEM_DOMAIN_OBJ (*macro C*), 196
- PYMEM_DOMAIN_RAW (*macro C*), 196
- PyMem_Free (*función C*), 193
- PyMem_GetAllocator (*función C*), 196
- PyMem_Malloc (*función C*), 193
- PyMem_New (*función C*), 194
- PyMem_RawCalloc (*función C*), 192
- PyMem_RawFree (*función C*), 193
- PyMem_RawMalloc (*función C*), 192
- PyMem_RawRealloc (*función C*), 192
- PyMem_Realloc (*función C*), 193
- PyMem_Resize (*función C*), 194
- PyMem_SetAllocator (*función C*), 196
- PyMem_SetupDebugHooks (*función C*), 197
- PyMemAllocatorDomain (*tipo C*), 196
- PyMemAllocatorEx (*tipo C*), 196
- PyMember_GetOne (*función C*), 207
- PyMember_SetOne (*función C*), 207
- PyMemberDef (*tipo C*), 206
- PyMemoryView_Check (*función C*), 143
- PyMemoryView_FromBuffer (*función C*), 143
- PyMemoryView_FromMemory (*función C*), 143
- PyMemoryView_FromObject (*función C*), 143
- PyMemoryView_GET_BASE (*función C*), 143
- PyMemoryView_GET_BUFFER (*función C*), 143
- PyMemoryView_GetContiguous (*función C*), 143
- PyMethod_Check (*función C*), 131
- PyMethod_Function (*función C*), 131
- PyMethod_GET_FUNCTION (*función C*), 131
- PyMethod_GET_SELF (*función C*), 131
- PyMethod_New (*función C*), 131
- PyMethod_Self (*función C*), 131
- PyMethod_Type (*variable C*), 131
- PyMethodDef (*tipo C*), 204
- PyModule_AddFunctions (*función C*), 139
- PyModule_AddIntConstant (*función C*), 139
- PyModule_AddIntMacro (*función C*), 139
- PyModule_AddObject (*función C*), 139
- PyModule_AddStringConstant (*función C*), 139
- PyModule_AddStringMacro (*función C*), 139
- PyModule_AddType (*función C*), 140
- PyModule_Check (*función C*), 134
- PyModule_CheckExact (*función C*), 134
- PyModule_Create (*función C*), 136
- PyModule_Create2 (*función C*), 136
- PyModule_ExecDef (*función C*), 138
- PyModule_FromDefAndSpec (*función C*), 138
- PyModule_FromDefAndSpec2 (*función C*), 138
- PyModule_GetDef (*función C*), 134
- PyModule_GetDict (*función C*), 134
- PyModule_GetFilename (*función C*), 135
- PyModule_GetFilenameObject (*función C*), 135
- PyModule_GetName (*función C*), 134
- PyModule_GetNameObject (*función C*), 134
- PyModule_GetState (*función C*), 134
- PyModule_New (*función C*), 134
- PyModule_NewObject (*función C*), 134
- PyModule_SetDocString (*función C*), 138
- PyModule_Type (*variable C*), 134
- PyModuleDef (*tipo C*), 135
- PyModuleDef_Init (*función C*), 137

- PyModuleDef_Slot (tipo C), 137
- PyModuleDef_Slot.slot (miembro C), 137
- PyModuleDef_Slot.value (miembro C), 137
- PyModuleDef.m_base (miembro C), 135
- PyModuleDef.m_clear (miembro C), 136
- PyModuleDef.m_doc (miembro C), 135
- PyModuleDef.m_free (miembro C), 136
- PyModuleDef.m_methods (miembro C), 135
- PyModuleDef.m_name (miembro C), 135
- PyModuleDef.m_reload (miembro C), 136
- PyModuleDef.m_size (miembro C), 135
- PyModuleDef.m_slots (miembro C), 135
- PyModuleDef.m_traverse (miembro C), 136
- PyNumber_Absolute (función C), 73
- PyNumber_Add (función C), 72
- PyNumber_And (función C), 73
- PyNumber_AsSsize_t (función C), 75
- PyNumber_Check (función C), 72
- PyNumber_Divmod (función C), 73
- PyNumber_Float (función C), 74
- PyNumber_FloorDivide (función C), 72
- PyNumber_Index (función C), 75
- PyNumber_InPlaceAdd (función C), 73
- PyNumber_InPlaceAnd (función C), 74
- PyNumber_InPlaceFloorDivide (función C), 74
- PyNumber_InPlaceLshift (función C), 74
- PyNumber_InPlaceMatrixMultiply (función C), 73
- PyNumber_InPlaceMultiply (función C), 73
- PyNumber_InPlaceOr (función C), 74
- PyNumber_InPlacePower (función C), 74
- PyNumber_InPlaceRemainder (función C), 74
- PyNumber_InPlaceRshift (función C), 74
- PyNumber_InPlaceSubtract (función C), 73
- PyNumber_InPlaceTrueDivide (función C), 74
- PyNumber_InPlaceXor (función C), 74
- PyNumber_Invert (función C), 73
- PyNumber_Long (función C), 74
- PyNumber_Lshift (función C), 73
- PyNumber_MatrixMultiply (función C), 72
- PyNumber_Multiply (función C), 72
- PyNumber_Negative (función C), 73
- PyNumber_Or (función C), 73
- PyNumber_Positive (función C), 73
- PyNumber_Power (función C), 73
- PyNumber_Remainder (función C), 72
- PyNumber_Rshift (función C), 73
- PyNumber_Subtract (función C), 72
- PyNumber_ToBase (función C), 75
- PyNumber_TrueDivide (función C), 72
- PyNumber_Xor (función C), 73
- PyNumberMethods (tipo C), 233
- PyNumberMethods.nb_absolute (miembro C), 234
- PyNumberMethods.nb_add (miembro C), 234
- PyNumberMethods.nb_and (miembro C), 234
- PyNumberMethods.nb_bool (miembro C), 234
- PyNumberMethods.nb_divmod (miembro C), 234
- PyNumberMethods.nb_float (miembro C), 234
- PyNumberMethods.nb_floor_divide (miembro C), 235
- PyNumberMethods.nb_index (miembro C), 235
- PyNumberMethods.nb_inplace_add (miembro C), 235
- PyNumberMethods.nb_inplace_and (miembro C), 235
- PyNumberMethods.nb_inplace_floor_divide (miembro C), 235
- PyNumberMethods.nb_inplace_lshift (miembro C), 235
- PyNumberMethods.nb_inplace_matrix_multiply (miembro C), 235
- PyNumberMethods.nb_inplace_multiply (miembro C), 235
- PyNumberMethods.nb_inplace_or (miembro C), 235
- PyNumberMethods.nb_inplace_power (miembro C), 235
- PyNumberMethods.nb_inplace_remainder (miembro C), 235
- PyNumberMethods.nb_inplace_rshift (miembro C), 235
- PyNumberMethods.nb_inplace_subtract (miembro C), 235
- PyNumberMethods.nb_inplace_true_divide (miembro C), 235
- PyNumberMethods.nb_inplace_xor (miembro C), 235
- PyNumberMethods.nb_int (miembro C), 234
- PyNumberMethods.nb_invert (miembro C), 234
- PyNumberMethods.nb_lshift (miembro C), 234
- PyNumberMethods.nb_matrix_multiply (miembro C), 235
- PyNumberMethods.nb_multiply (miembro C), 234
- PyNumberMethods.nb_negative (miembro C), 234
- PyNumberMethods.nb_or (miembro C), 234
- PyNumberMethods.nb_positive (miembro C), 234
- PyNumberMethods.nb_power (miembro C), 234
- PyNumberMethods.nb_remainder (miembro C), 234
- PyNumberMethods.nb_reserved (miembro C), 234
- PyNumberMethods.nb_rshift (miembro C), 234
- PyNumberMethods.nb_subtract (miembro C), 234

- PyNumberMethods.nb_true_divide (miembro C), 235
- PyNumberMethods.nb_xor (miembro C), 234
- PyObject (tipo C), 202
- PyObject_AsCharBuffer (función C), 86
- PyObject_ASCII (función C), 65
- PyObject_AsFileDescriptor (función C), 133
- PyObject_AsReadBuffer (función C), 86
- PyObject_AsWriteBuffer (función C), 86
- PyObject_Bytes (función C), 65
- PyObject_Call (función C), 70
- PyObject_CallFunction (función C), 70
- PyObject_CallFunctionObjArgs (función C), 70
- PyObject_CallMethod (función C), 70
- PyObject_CallMethodNoArgs (función C), 71
- PyObject_CallMethodObjArgs (función C), 71
- PyObject_CallMethodOneArg (función C), 71
- PyObject_CallNoArgs (función C), 70
- PyObject_CallObject (función C), 70
- PyObject_Calloc (función C), 194
- PyObject_CallOneArg (función C), 70
- PyObject_CheckBuffer (función C), 84
- PyObject_CheckReadBuffer (función C), 86
- PyObject_Del (función C), 201
- PyObject_DelAttr (función C), 64
- PyObject_DelAttrString (función C), 64
- PyObject_DelItem (función C), 67
- PyObject_Dir (función C), 67
- PyObject_Free (función C), 195
- PyObject_GC_Del (función C), 243
- PyObject_GC_IsFinalized (función C), 243
- PyObject_GC_IsTracked (función C), 243
- PyObject_GC_New (función C), 243
- PyObject_GC_NewVar (función C), 243
- PyObject_GC_Resize (función C), 243
- PyObject_GC_Track (función C), 243
- PyObject_GC_UnTrack (función C), 244
- PyObject_GenericGetAttr (función C), 64
- PyObject_GenericGetDict (función C), 64
- PyObject_GenericSetAttr (función C), 64
- PyObject_GenericSetDict (función C), 64
- PyObject_GetArenaAllocator (función C), 198
- PyObject_GetAttr (función C), 64
- PyObject_GetAttrString (función C), 64
- PyObject_GetBuffer (función C), 84
- PyObject_GetItem (función C), 66
- PyObject_GetIter (función C), 67
- PyObject_HasAttr (función C), 63
- PyObject_HasAttrString (función C), 63
- PyObject_Hash (función C), 66
- PyObject_HashNotImplemented (función C), 66
- PyObject_HEAD (macro C), 202
- PyObject_HEAD_INIT (macro C), 203
- PyObject_Init (función C), 201
- PyObject_InitVar (función C), 201
- PyObject_IS_GC (función C), 243
- PyObject_IsInstance (función C), 65
- PyObject_IsSubclass (función C), 65
- PyObject_IsTrue (función C), 66
- PyObject_Length (función C), 66
- PyObject_LengthHint (función C), 66
- PyObject_Malloc (función C), 194
- PyObject_New (función C), 201
- PyObject_NewVar (función C), 201
- PyObject_Not (función C), 66
- PyObject._ob_next (miembro C), 214
- PyObject._ob_prev (miembro C), 214
- PyObject_Print (función C), 63
- PyObject_Realloc (función C), 194
- PyObject_Repr (función C), 65
- PyObject_RichCompare (función C), 64
- PyObject_RichCompareBool (función C), 65
- PyObject_SetArenaAllocator (función C), 198
- PyObject_SetAttr (función C), 64
- PyObject_SetAttrString (función C), 64
- PyObject_SetItem (función C), 66
- PyObject_Size (función C), 66
- PyObject_Str (función C), 65
- PyObject_Type (función C), 66
- PyObject_TypeCheck (función C), 66
- PyObject_VAR_HEAD (macro C), 202
- PyObject_Vectorcall (función C), 71
- PyObject_VectorcallDict (función C), 71
- PyObject_VectorcallMethod (función C), 71
- PyObjectArenaAllocator (tipo C), 198
- PyObject.ob_refcnt (miembro C), 214
- PyObject.ob_type (miembro C), 215
- PyOS_AfterFork (función C), 40
- PyOS_AfterFork_Child (función C), 40
- PyOS_AfterFork_Parent (función C), 39
- PyOS_BeforeFork (función C), 39
- PyOS_CheckStack (función C), 40
- PyOS_double_to_string (función C), 58
- PyOS_FSPath (función C), 39
- PyOS_getsig (función C), 40
- PyOS_InputHook (variable C), 19
- PyOS_ReadlineFunctionPointer (variable C), 19
- PyOS_setsig (función C), 40
- PyOS_snprintf (función C), 57
- PyOS_stricmp (función C), 58
- PyOS_string_to_double (función C), 57
- PyOS_strnicmp (función C), 58
- PyOS_vsnprintf (función C), 57
- PyParser_SimpleParseFile (función C), 19
- PyParser_SimpleParseFileFlags (función C), 19
- PyParser_SimpleParseString (función C), 19

- PyParser_SimpleParseStringFlags (*función C*), 19
- PyParser_SimpleParseStringFlagsFilename (*función C*), 19
- PyPreConfig (*tipo C*), 178
- PyPreConfig_InitIsolatedConfig (*función C*), 178
- PyPreConfig_InitPythonConfig (*función C*), 178
- PyPreConfig.allocator (*miembro C*), 178
- PyPreConfig.coerce_c_locale (*miembro C*), 179
- PyPreConfig.coerce_c_locale_warn (*miembro C*), 179
- PyPreConfig.configure_locale (*miembro C*), 178
- PyPreConfig.dev_mode (*miembro C*), 179
- PyPreConfig.isolated (*miembro C*), 179
- PyPreConfig.legacy_windows_fs_encoding (*miembro C*), 179
- PyPreConfig.parse_argv (*miembro C*), 179
- PyPreConfig.use_environment (*miembro C*), 179
- PyPreConfig.utf8_mode (*miembro C*), 179
- PyProperty_Type (*variable C*), 141
- PyRun_AnyFile (*función C*), 17
- PyRun_AnyFileEx (*función C*), 17
- PyRun_AnyFileExFlags (*función C*), 18
- PyRun_AnyFileFlags (*función C*), 17
- PyRun_File (*función C*), 20
- PyRun_FileEx (*función C*), 20
- PyRun_FileExFlags (*función C*), 20
- PyRun_FileFlags (*función C*), 20
- PyRun_InteractiveLoop (*función C*), 18
- PyRun_InteractiveLoopFlags (*función C*), 18
- PyRun_InteractiveOne (*función C*), 18
- PyRun_InteractiveOneFlags (*función C*), 18
- PyRun_SimpleFile (*función C*), 18
- PyRun_SimpleFileEx (*función C*), 18
- PyRun_SimpleFileExFlags (*función C*), 18
- PyRun_SimpleString (*función C*), 18
- PyRun_SimpleStringFlags (*función C*), 18
- PyRun_String (*función C*), 19
- PyRun_StringFlags (*función C*), 19
- PySeqIter_Check (*función C*), 141
- PySeqIter_New (*función C*), 141
- PySeqIter_Type (*variable C*), 141
- PySequence_Check (*función C*), 75
- PySequence_Concat (*función C*), 75
- PySequence_Contains (*función C*), 76
- PySequence_Count (*función C*), 76
- PySequence_DelItem (*función C*), 76
- PySequence_DelSlice (*función C*), 76
- PySequence_Fast (*función C*), 76
- PySequence_Fast_GET_ITEM (*función C*), 76
- PySequence_Fast_GET_SIZE (*función C*), 76
- PySequence_Fast_ITEMS (*función C*), 77
- PySequence_GetItem (*función C*), 75
- PySequence_GetItem(), 9
- PySequence_GetSlice (*función C*), 76
- PySequence_Index (*función C*), 76
- PySequence_InPlaceConcat (*función C*), 75
- PySequence_InPlaceRepeat (*función C*), 75
- PySequence_ITEM (*función C*), 77
- PySequence_Length (*función C*), 75
- PySequence_List (*función C*), 76
- PySequence_Repeat (*función C*), 75
- PySequence_SetItem (*función C*), 76
- PySequence_SetSlice (*función C*), 76
- PySequence_Size (*función C*), 75
- PySequence_Tuple (*función C*), 76
- PySequenceMethods (*tipo C*), 236
- PySequenceMethods.sq_ass_item (*miembro C*), 236
- PySequenceMethods.sq_concat (*miembro C*), 236
- PySequenceMethods.sq_contains (*miembro C*), 236
- PySequenceMethods.sq_inplace_concat (*miembro C*), 236
- PySequenceMethods.sq_inplace_repeat (*miembro C*), 236
- PySequenceMethods.sq_item (*miembro C*), 236
- PySequenceMethods.sq_length (*miembro C*), 236
- PySequenceMethods.sq_repeat (*miembro C*), 236
- PySet_Add (*función C*), 128
- PySet_Check (*función C*), 128
- PySet_Clear (*función C*), 129
- PySet_Contains (*función C*), 128
- PySet_Discard (*función C*), 129
- PySet_GET_SIZE (*función C*), 128
- PySet_New (*función C*), 128
- PySet_Pop (*función C*), 129
- PySet_Size (*función C*), 128
- PySet_Type (*variable C*), 127
- PySetObject (*tipo C*), 127
- PySignal_SetWakeupFd (*función C*), 32
- PySlice_AdjustIndices (*función C*), 143
- PySlice_Check (*función C*), 142
- PySlice_GetIndices (*función C*), 142
- PySlice_GetIndicesEx (*función C*), 142
- PySlice_New (*función C*), 142
- PySlice_Type (*variable C*), 142
- PySlice_Unpack (*función C*), 142
- PyState_AddModule (*función C*), 140
- PyState_FindModule (*función C*), 140

- PyState_RemoveModule (función C), 140
- PyStatus (tipo C), 177
- PyStatus_Error (función C), 177
- PyStatus_Exception (función C), 177
- PyStatus_Exit (función C), 177
- PyStatus_IsError (función C), 177
- PyStatus_IsExit (función C), 177
- PyStatus_NoMemory (función C), 177
- PyStatus_Ok (función C), 177
- PyStatus.err_msg (miembro C), 177
- PyStatus.exitcode (miembro C), 177
- PyStatus.func (miembro C), 177
- PyStructSequence_Desc (tipo C), 122
- PyStructSequence_Field (tipo C), 122
- PyStructSequence_GET_ITEM (función C), 123
- PyStructSequence_GetItem (función C), 123
- PyStructSequence_InitType (función C), 122
- PyStructSequence_InitType2 (función C), 122
- PyStructSequence_New (función C), 122
- PyStructSequence_NewType (función C), 122
- PyStructSequence_SET_ITEM (función C), 123
- PyStructSequence_SetItem (función C), 123
- PyStructSequence_UnnamedField (variable C), 122
- PySys_AddAuditHook (función C), 43
- PySys_AddWarnOption (función C), 42
- PySys_AddWarnOptionUnicode (función C), 42
- PySys_AddXOption (función C), 43
- PySys_Audit (función C), 43
- PySys_FormatStderr (función C), 42
- PySys_FormatStdout (función C), 42
- PySys_GetObject (función C), 42
- PySys_GetXOptions (función C), 43
- PySys_ResetWarnOptions (función C), 42
- PySys_SetArgv (función C), 160
- PySys_SetArgv(), 156
- PySys_SetArgvEx (función C), 159
- PySys_SetArgvEx(), 12, 156
- PySys_SetObject (función C), 42
- PySys_SetPath (función C), 42
- PySys_WriteStderr (función C), 42
- PySys_WriteStdout (función C), 42
- Python 3000, 257
- Python Enhancement Proposals
 - PEP 1, 257
 - PEP 7, 4, 6
 - PEP 238, 22, 251
 - PEP 278, 260
 - PEP 302, 251, 254
 - PEP 343, 249
 - PEP 353, 10
 - PEP 362, 248, 257
 - PEP 383, 109, 110
 - PEP 384, 15
 - PEP 393, 100, 108
 - PEP 411, 257
 - PEP 420, 251, 256, 257
 - PEP 432, 189, 190
 - PEP 442, 232
 - PEP 443, 252
 - PEP 451, 138, 251
 - PEP 483, 252
 - PEP 484, 247, 251, 252, 259, 260
 - PEP 489, 138
 - PEP 492, 248, 250
 - PEP 498, 251
 - PEP 519, 257
 - PEP 523, 166
 - PEP 525, 248
 - PEP 526, 247, 260
 - PEP 528, 155
 - PEP 529, 110, 155
 - PEP 538, 187
 - PEP 539, 172
 - PEP 540, 187
 - PEP 552, 182
 - PEP 578, 43
 - PEP 585, 252
 - PEP 587, 176
 - PEP 590, 67
 - PEP 617, 184
 - PEP 623, 100
 - PEP 3116, 260
 - PEP 3119, 65, 66
 - PEP 3121, 135
 - PEP 3147, 46
 - PEP 3151, 36
 - PEP 3155, 258
- PYTHON*, 155
- PYTHONCOERCECLOCALE, 187
- PYTHONDEBUG, 154
- PYTHONDONTWRITEBYTECODE, 154
- PYTHONDUMPREFS, 214
- PYTHONHASHSEED, 155
- PYTHONHOME, 12, 155, 160, 182
- Pythónico, 257
- PYTHONINSPECT, 155
- PYTHONIOENCODING, 157
- PYTHONLEGACYWINDOWSFSENCODING, 155
- PYTHONLEGACYWINDOWSSSTDIO, 155
- PYTHONMALLOC, 192, 195, 197
- PYTHONMALLOCSTATS, 192
- PYTHONNOUSERSITE, 156
- PYTHONOLDPARSER, 184
- PYTHONOPTIMIZE, 156
- PYTHONPATH, 12, 155, 183
- PYTHONUNBUFFERED, 156
- PYTHONUTF8, 187

- PYTHONVERBOSE, 156
 PyThread_create_key (función C), 173
 PyThread_delete_key (función C), 173
 PyThread_delete_key_value (función C), 173
 PyThread_get_key_value (función C), 173
 PyThread_ReInitTLS (función C), 173
 PyThread_set_key_value (función C), 173
 PyThread_tss_alloc (función C), 172
 PyThread_tss_create (función C), 173
 PyThread_tss_delete (función C), 173
 PyThread_tss_free (función C), 172
 PyThread_tss_get (función C), 173
 PyThread_tss_is_created (función C), 173
 PyThread_tss_set (función C), 173
 PyThreadState, 160
 PyThreadState (tipo C), 163
 PyThreadState_Clear (función C), 165
 PyThreadState_Delete (función C), 165
 PyThreadState_DeleteCurrent (función C), 165
 PyThreadState_Get (función C), 163
 PyThreadState_GetDict (función C), 166
 PyThreadState_GetFrame (función C), 165
 PyThreadState_GetID (función C), 165
 PyThreadState_GetInterpreter (función C), 165
 PyThreadState_New (función C), 165
 PyThreadState_Next (función C), 171
 PyThreadState_SetAsyncExc (función C), 166
 PyThreadState_Swap (función C), 163
 PyTime_Check (función C), 149
 PyTime_CheckExact (función C), 149
 PyTime_FromTime (función C), 150
 PyTime_FromTimeAndFold (función C), 150
 PyTimeZone_FromOffset (función C), 150
 PyTimeZone_FromOffsetAndName (función C), 150
 PyTrace_C_CALL (variable C), 170
 PyTrace_C_EXCEPTION (variable C), 170
 PyTrace_C_RETURN (variable C), 171
 PyTrace_CALL (variable C), 170
 PyTrace_EXCEPTION (variable C), 170
 PyTrace_LINE (variable C), 170
 PyTrace_OPCODE (variable C), 171
 PyTrace_RETURN (variable C), 170
 PyTraceMalloc_Track (función C), 198
 PyTraceMalloc_Untrack (función C), 198
 PyTuple_Check (función C), 121
 PyTuple_CheckExact (función C), 121
 PyTuple_GET_ITEM (función C), 121
 PyTuple_GET_SIZE (función C), 121
 PyTuple_GetItem (función C), 121
 PyTuple_GetSlice (función C), 121
 PyTuple_New (función C), 121
 PyTuple_Pack (función C), 121
 PyTuple_SET_ITEM (función C), 121
 PyTuple_SetItem (función C), 121
 PyTuple_SetItem(), 8
 PyTuple_Size (función C), 121
 PyTuple_Type (variable C), 121
 PyTupleObject (tipo C), 121
 PyType_Check (función C), 87
 PyType_CheckExact (función C), 87
 PyType_ClearCache (función C), 87
 PyType_FromModuleAndSpec (función C), 89
 PyType_FromSpec (función C), 89
 PyType_FromSpecWithBases (función C), 89
 PyType_GenericAlloc (función C), 88
 PyType_GenericNew (función C), 88
 PyType_GetFlags (función C), 88
 PyType_GetModule (función C), 88
 PyType_GetModuleState (función C), 89
 PyType_GetSlot (función C), 88
 PyType_HasFeature (función C), 88
 PyType_IS_GC (función C), 88
 PyType_IsSubtype (función C), 88
 PyType_Modified (función C), 88
 PyType_Ready (función C), 88
 PyType_Slot (tipo C), 90
 PyType_Slot.PyType_Slot.pfunc (miembro C), 90
 PyType_Slot.PyType_Slot.slot (miembro C), 90
 PyType_Spec (tipo C), 89
 PyType_Spec.PyType_Spec.basicsize (miembro C), 89
 PyType_Spec.PyType_Spec.flags (miembro C), 90
 PyType_Spec.PyType_Spec.itemsize (miembro C), 89
 PyType_Spec.PyType_Spec.name (miembro C), 89
 PyType_Spec.PyType_Spec.slots (miembro C), 90
 PyType_Type (variable C), 87
 PyTypeObject (tipo C), 87
 PyTypeObject.tp_alloc (miembro C), 229
 PyTypeObject.tp_as_async (miembro C), 218
 PyTypeObject.tp_as_buffer (miembro C), 220
 PyTypeObject.tp_as_mapping (miembro C), 219
 PyTypeObject.tp_as_number (miembro C), 218
 PyTypeObject.tp_as_sequence (miembro C), 218
 PyTypeObject.tp_base (miembro C), 227
 PyTypeObject.tp_bases (miembro C), 231
 PyTypeObject.tp_basicsize (miembro C), 216
 PyTypeObject.tp_cache (miembro C), 231
 PyTypeObject.tp_call (miembro C), 219
 PyTypeObject.tp_clear (miembro C), 224

- PyTypeObject.tp_dealloc (*miembro C*), 216
- PyTypeObject.tp_del (*miembro C*), 231
- PyTypeObject.tp_descr_get (*miembro C*), 228
- PyTypeObject.tp_descr_set (*miembro C*), 228
- PyTypeObject.tp_dict (*miembro C*), 227
- PyTypeObject.tp_dictoffset (*miembro C*), 228
- PyTypeObject.tp_doc (*miembro C*), 223
- PyTypeObject.tp_finalize (*miembro C*), 231
- PyTypeObject.tp_flags (*miembro C*), 220
- PyTypeObject.tp_free (*miembro C*), 230
- PyTypeObject.tp_getattr (*miembro C*), 217
- PyTypeObject.tp_getattro (*miembro C*), 220
- PyTypeObject.tp_getset (*miembro C*), 227
- PyTypeObject.tp_hash (*miembro C*), 219
- PyTypeObject.tp_init (*miembro C*), 229
- PyTypeObject.tp_is_gc (*miembro C*), 230
- PyTypeObject.tp_itemsize (*miembro C*), 216
- PyTypeObject.tp_iter (*miembro C*), 226
- PyTypeObject.tp_iternext (*miembro C*), 226
- PyTypeObject.tp_members (*miembro C*), 227
- PyTypeObject.tp_methods (*miembro C*), 226
- PyTypeObject.tp_mro (*miembro C*), 231
- PyTypeObject.tp_name (*miembro C*), 215
- PyTypeObject.tp_new (*miembro C*), 230
- PyTypeObject.tp_repr (*miembro C*), 218
- PyTypeObject.tp_richcompare (*miembro C*), 225
- PyTypeObject.tp_setattr (*miembro C*), 218
- PyTypeObject.tp_setattro (*miembro C*), 220
- PyTypeObject.tp_str (*miembro C*), 219
- PyTypeObject.tp_subclasses (*miembro C*), 231
- PyTypeObject.tp_traverse (*miembro C*), 223
- PyTypeObject.tp_vectorcall (*miembro C*), 232
- PyTypeObject.tp_vectorcall_offset (*miembro C*), 217
- PyTypeObject.tp_version_tag (*miembro C*), 231
- PyTypeObject.tp_weaklist (*miembro C*), 231
- PyTypeObject.tp_weaklistoffset (*miembro C*), 226
- PyTZInfo_Check (*función C*), 149
- PyTZInfo_CheckExact (*función C*), 149
- PyUnicode_1BYTE_DATA (*función C*), 101
- PyUnicode_1BYTE_KIND (*macro C*), 102
- PyUnicode_2BYTE_DATA (*función C*), 101
- PyUnicode_2BYTE_KIND (*macro C*), 102
- PyUnicode_4BYTE_DATA (*función C*), 101
- PyUnicode_4BYTE_KIND (*macro C*), 102
- PyUnicode_AS_DATA (*función C*), 103
- PyUnicode_AS_UNICODE (*función C*), 103
- PyUnicode_AsASCIIString (*función C*), 117
- PyUnicode_AsCharmapString (*función C*), 117
- PyUnicode_AsEncodedString (*función C*), 112
- PyUnicode_AsLatin1String (*función C*), 117
- PyUnicode_AsMBCSString (*función C*), 118
- PyUnicode_AsRawUnicodeEscapeString (*función C*), 116
- PyUnicode_AsUCS4 (*función C*), 107
- PyUnicode_AsUCS4Copy (*función C*), 107
- PyUnicode_AsUnicode (*función C*), 108
- PyUnicode_AsUnicodeAndSize (*función C*), 108
- PyUnicode_AsUnicodeCopy (*función C*), 109
- PyUnicode_AsUnicodeEscapeString (*función C*), 116
- PyUnicode_AsUTF8 (*función C*), 113
- PyUnicode_AsUTF8AndSize (*función C*), 113
- PyUnicode_AsUTF8String (*función C*), 113
- PyUnicode_AsUTF16String (*función C*), 115
- PyUnicode_AsUTF32String (*función C*), 114
- PyUnicode_AsWideChar (*función C*), 111
- PyUnicode_AsWideCharString (*función C*), 111
- PyUnicode_Check (*función C*), 101
- PyUnicode_CheckExact (*función C*), 101
- PyUnicode_Compare (*función C*), 120
- PyUnicode_CompareWithASCIIString (*función C*), 120
- PyUnicode_Concat (*función C*), 119
- PyUnicode_Contains (*función C*), 120
- PyUnicode_CopyCharacters (*función C*), 107
- PyUnicode_Count (*función C*), 120
- PyUnicode_DATA (*función C*), 102
- PyUnicode_Decode (*función C*), 112
- PyUnicode_DecodeASCII (*función C*), 117
- PyUnicode_DecodeCharmap (*función C*), 117
- PyUnicode_DecodeFSDefault (*función C*), 111
- PyUnicode_DecodeFSDefaultAndSize (*función C*), 110
- PyUnicode_DecodeLatin1 (*función C*), 117
- PyUnicode_DecodeLocale (*función C*), 109
- PyUnicode_DecodeLocaleAndSize (*función C*), 109
- PyUnicode_DecodeMBCS (*función C*), 118
- PyUnicode_DecodeMBCSStateful (*función C*), 118
- PyUnicode_DecodeRawUnicodeEscape (*función C*), 116
- PyUnicode_DecodeUnicodeEscape (*función C*), 116
- PyUnicode_DecodeUTF7 (*función C*), 115
- PyUnicode_DecodeUTF7Stateful (*función C*), 115
- PyUnicode_DecodeUTF8 (*función C*), 113
- PyUnicode_DecodeUTF8Stateful (*función C*), 113
- PyUnicode_DecodeUTF16 (*función C*), 114
- PyUnicode_DecodeUTF16Stateful (*función C*), 115
- PyUnicode_DecodeUTF32 (*función C*), 113

- PyUnicode_DecodeUTF32Stateful (función C), 114
- PyUnicode_Encode (función C), 112
- PyUnicode_EncodeASCII (función C), 117
- PyUnicode_EncodeCharmap (función C), 118
- PyUnicode_EncodeCodePage (función C), 118
- PyUnicode_EncodeFSDefault (función C), 111
- PyUnicode_EncodeLatin1 (función C), 117
- PyUnicode_EncodeLocale (función C), 109
- PyUnicode_EncodeMBCS (función C), 119
- PyUnicode_EncodeRawUnicodeEscape (función C), 116
- PyUnicode_EncodeUnicodeEscape (función C), 116
- PyUnicode_EncodeUTF7 (función C), 115
- PyUnicode_EncodeUTF8 (función C), 113
- PyUnicode_EncodeUTF16 (función C), 115
- PyUnicode_EncodeUTF32 (función C), 114
- PyUnicode_Fill (función C), 107
- PyUnicode_Find (función C), 119
- PyUnicode_FindChar (función C), 119
- PyUnicode_Format (función C), 120
- PyUnicode_FromEncodedObject (función C), 106
- PyUnicode_FromFormat (función C), 105
- PyUnicode_FromFormatV (función C), 106
- PyUnicode_FromKindAndData (función C), 105
- PyUnicode_FromObject (función C), 109
- PyUnicode_FromString (función C), 105
- PyUnicode_FromString(), 125
- PyUnicode_FromStringAndSize (función C), 105
- PyUnicode_FromUnicode (función C), 108
- PyUnicode_FromWideChar (función C), 111
- PyUnicode_FSConverter (función C), 110
- PyUnicode_FSDecoder (función C), 110
- PyUnicode_GET_DATA_SIZE (función C), 102
- PyUnicode_GET_LENGTH (función C), 101
- PyUnicode_GET_SIZE (función C), 102
- PyUnicode_GetLength (función C), 107
- PyUnicode_GetSize (función C), 109
- PyUnicode_InternFromString (función C), 120
- PyUnicode_InternInPlace (función C), 120
- PyUnicode_IsIdentifier (función C), 103
- PyUnicode_Join (función C), 119
- PyUnicode_KIND (función C), 102
- PyUnicode_MAX_CHAR_VALUE (macro C), 102
- PyUnicode_New (función C), 105
- PyUnicode_READ (función C), 102
- PyUnicode_READ_CHAR (función C), 102
- PyUnicode_ReadChar (función C), 107
- PyUnicode_READY (función C), 101
- PyUnicode_Replace (función C), 120
- PyUnicode_RichCompare (función C), 120
- PyUnicode_Split (función C), 119
- PyUnicode_Splitlines (función C), 119
- PyUnicode_Substring (función C), 107
- PyUnicode_Tailmatch (función C), 119
- PyUnicode_TransformDecimalToASCII (función C), 108
- PyUnicode_Translate (función C), 118
- PyUnicode_TranslateCharmap (función C), 118
- PyUnicode_Type (variable C), 101
- PyUnicode_WCHAR_KIND (macro C), 102
- PyUnicode_WRITE (función C), 102
- PyUnicode_WriteChar (función C), 107
- PyUnicodeDecodeError_Create (función C), 33
- PyUnicodeDecodeError_GetEncoding (función C), 33
- PyUnicodeDecodeError_GetEnd (función C), 33
- PyUnicodeDecodeError_GetObject (función C), 33
- PyUnicodeDecodeError_GetReason (función C), 34
- PyUnicodeDecodeError_GetStart (función C), 33
- PyUnicodeDecodeError_SetEnd (función C), 34
- PyUnicodeDecodeError_SetReason (función C), 34
- PyUnicodeDecodeError_SetStart (función C), 33
- PyUnicodeEncodeError_Create (función C), 33
- PyUnicodeEncodeError_GetEncoding (función C), 33
- PyUnicodeEncodeError_GetEnd (función C), 33
- PyUnicodeEncodeError_GetObject (función C), 33
- PyUnicodeEncodeError_GetReason (función C), 34
- PyUnicodeEncodeError_GetStart (función C), 33
- PyUnicodeEncodeError_SetEnd (función C), 34
- PyUnicodeEncodeError_SetReason (función C), 34
- PyUnicodeEncodeError_SetStart (función C), 33
- PyUnicodeObject (tipo C), 101
- PyUnicodeTranslateError_Create (función C), 33
- PyUnicodeTranslateError_GetEnd (función C), 33
- PyUnicodeTranslateError_GetObject (función C), 33
- PyUnicodeTranslateError_GetReason (función C), 34
- PyUnicodeTranslateError_GetStart (función C), 33
- PyUnicodeTranslateError_SetEnd (función C), 34
- PyUnicodeTranslateError_SetReason (función C), 34

ción C), 34
PyUnicodeTranslateError_SetStart (*función C*), 33
PyVarObject (*tipo C*), 202
PyVarObject_HEAD_INIT (*macro C*), 203
PyVarObject.ob_size (*miembro C*), 215
PyVectorcall_Call (*función C*), 69
PyVectorcall_Function (*función C*), 69
PyVectorcall_NARGS (*función C*), 69
PyWeakref_Check (*función C*), 144
PyWeakref_CheckProxy (*función C*), 144
PyWeakref_CheckRef (*función C*), 144
PyWeakref_GET_OBJECT (*función C*), 144
PyWeakref_GetObject (*función C*), 144
PyWeakref_NewProxy (*función C*), 144
PyWeakref_NewRef (*función C*), 144
PyWideStringList (*tipo C*), 176
PyWideStringList_Append (*función C*), 176
PyWideStringList_Insert (*función C*), 176
PyWideStringList.items (*miembro C*), 176
PyWideStringList.length (*miembro C*), 176
PyWrapper_New (*función C*), 141

R

realloc(), 191
rebanada, 259
recolección de basura, 252
releasebufferproc (*tipo C*), 240
repr
 función incorporada, 65, 218
reprfunc (*tipo C*), 239
retrollamada, 249
richcmpfunc (*tipo C*), 239
ruta de importación, 253

S

saltos de líneas universales, 260
stderr
 stdin stdout, 157
search
 path, module, 12, 156, 158
secuencia, 258
sentencia, 259
sequence
 objeto, 97
set
 objeto, 127
set_all(), 9
setattrfunc (*tipo C*), 239
setattrofunc (*tipo C*), 239
setswitchinterval() (*in module sys*), 160
SIGINT, 31
signal
 módulo, 31

SIZE_MAX, 93
special
 method, 259
ssizeargfunc (*tipo C*), 240
ssizeobjargproc (*tipo C*), 240
staticmethod
 función incorporada, 206
stderr (*in module sys*), 168
stdin
 stdout stderr, 157
stdin (*in module sys*), 168
stdout
 stderr, stdin, 157
stdout (*in module sys*), 168
strerror(), 27
string
 PyObject_Str (*C function*), 65
sum_list(), 9
sum_sequence(), 10, 11
sys
 módulo, 12, 156, 168
SystemError (*built-in exception*), 134, 135

T

ternaryfunc (*tipo C*), 240
tipado de pato, 250
tipo, 259
tipos genéricos, 252
traverseproc (*tipo C*), 244
tupla nombrada, 255
tuple
 función incorporada, 76, 124
 objeto, 121
type
 función incorporada, 66
 objeto, 6, 87

U

ULONG_MAX, 93
unaryfunc (*tipo C*), 240

V

variable de clase, 249
variable de contexto, 249
variables de entorno
 exec_prefix, 4
 PATH, 12
 prefix, 4
 PYTHON*, 155
 PYTHONCOERCECLOCALE, 187
 PYTHONDEBUG, 154
 PYTHONDONTWRITEBYTECODE, 154
 PYTHONDUMPREFS, 214
 PYTHONHASHSEED, 155

PYTHONHOME, 12, 155, 160, 182
PYTHONINSPECT, 155
PYTHONIOENCODING, 157
PYTHONLEGACYWINDOWSFSENCODING, 155
PYTHONLEGACYWINDOWSTDIO, 155
PYTHONMALLOC, 192, 195, 197
PYTHONMALLOCSTATS, 192
PYTHONNOUSERSITE, 156
PYTHONOLDPARSER, 184
PYTHONOPTIMIZE, 156
PYTHONPATH, 12, 155, 183
PYTHONUNBUFFERED, 156
PYTHONUTF8, 187
PYTHONVERBOSE, 156
vectorcallfunc (*tipo C*), 68
version (*in module sys*), 159
visitproc (*tipo C*), 244
vista de diccionario, 250

Z

Zen de Python, 260